Implementation of NAT44 and NAT64 using TC-BPF and eXpress Data Path (XDP)

Shashank D
Wireless Information Networking Group
Dept. of CSE, NITK Surathkal
Mangalore, India
shashankd.181co24@nitk.edu.in

Bhaskar Kataria
Wireless Information Networking Group
Dept. of CSE, NITK Surathkal
Mangalore, India
bhaskar.181co213@nitk.edu.in

Aditya Sohoni
Wireless Information Networking Group
Dept. of CSE, NITK Surathkal
Mangalore, India
adityasohoni.181co203@nitk.edu.in

Mohit P. Tahiliani
Wireless Information Networking Group
Dept. of CSE, NITK Surathkal
Mangalore, India
tahiliani@nitk.edu.in

***Abstract*—Large number of new devices connecting to the Internet has overwhelmed the available IPv4 address space. With devices that are IPv6 enabled, there is a need to translate their addresses to IPv4 so that they can communicate with servers that use IPv4. Network Address Translation (NAT) solves this problem by mapping IPv6 addresses to IPv4 and performing the translation at the router between the IPv4-enabled client and IPv6-enabled server. This is called NAT64. NAT is currently used by most of the Internet Service Providers (ISPs) around the world. However, most of the existing implementations involve a lot of kernel overhead. eXpress Data Path (XDP) is a relatively new concept that lets packets be processed faster than the normal network stack. It requires a modification to the kernel and allows packets to move through an integrated fast path in the kernel stack. XDP-NAT is being treated as a feasible alternative to the traditional NAT implementations, owing to its advantages such as low processing overhead and easy implementation. This work focuses on using the packet processing capabilities of XDP to perform address translation. This paper describes the design and a proof-of-concept implementation of NAT64 using XDP.**

***Index Terms*—Address Translation, NAT64, eXpress Data Path**

I. Introduction

The increasing number of devices in the Internet has led to depletion of IPv4 addresses. IPv6 resolves this issue by providing a much larger address space. Internet is now shifting towards IPv6 from IPv4. The latest data show that India has 60% of its current traffic using IPv6 [1]. However, migrating existing networks to IPv6 is a tedious process. Hence, not all networks can shift to IPv6 at once. This leaves us with the problem of connecting IPv4 and IPv6 networks. Network Address Translation (NAT) and NAT64 [2], [3] were introduced to solve such problems. NAT64 is an IPv6 transition mechanism that helps communicate between IPv6 and IPv4 hosts by using NAT. It is an address translator between IPv4 and IPv6. It needs at least one IPv4 address and an IPv6 network segment having a 32-bit address space. For an IPv6 client to communicate with an IPv4 client on the Internet, an internal DNS system should be set up that provides an IPv6 address containing the information about the external target IPv4 address. This IPv6 address is obtained by appending the IPv4 address to a fixed IPv6 prefix usually of 96 bits. The client sends its packets to the obtained IPv6 address. The NAT64 gateway connecting to the public Internet creates a mapping between IPv6 and IPv4 addresses, which may be manually configured or determined automatically.

This paper makes three key contributions towards the implementation of: (i) stateless NAT44 using Traffic Control - Berkeley Packet Filter (TC-BPF) (ii) stateless NAT64 using TC-BPF, and (iii) stateless NAT64 using eXpress Data Path (XDP) [4]. XDP is a high-performance data path for packets based on the eBPF technology [5] in the Linux kernel. The main idea behind XDP is to add an early hook at the lowest point in the software stack in the receive (RX) path of the kernel and let a user-supplied eBPF program handle the packet. This hook is placed in the Network Interface Controller (NIC) driver just after the interrupt processing and even before the network stack allocates any memory because memory allocation can be expensive. XDP requires support in the NIC driver, but as not all drivers support it, it can fall back to a generic implementation which performs the eBPF processing in the network stack, although with slower performance.

Since XDP allows packets to move through an integrated fast path in the kernel, it requires much less overhead than traditional kernel calls [6], thus allowing faster processing of packets while performing NAT. The time and resource savings due to lesser overhead makes XDP a viable technology to use for NAT implementations. However, the larger effort of evaluating the performance of NAT using XDP is a work-in-progress. This paper mainly focuses on the implementation of different types of NAT (for example, NAT44, NAT64) using XDP. It does not discuss the performance benchmarking of these implementations.

II. Background

A. Network Address Translation (NAT44 and NAT64)

NAT maps local private network addresses to public ones. Usually, NAT is used by organizations that want to advertise a small set of public IPv4 addresses for all their outbound traffic. NAT gained popularity due to the limited number of IPv4 addresses that are available. The NAT configuration requires at least one interface on a router (NAT outside), and another interface connecting the router with internal devices (NAT inside), along with specified rules to translate the IP addresses of incoming packets. When a device in the unregistered internal network needs to communicate with the outside world, the router translates those internal IP addresses to registered public IP addresses. However, the server that the internal devices are trying to communicate with might understand only IPv4 and may not be compatible with IPv6. So for the internal devices that use IPv6, their address is mapped to a valid IPv4 address and then the packet is sent to the server. This is commonly known as NAT64. The two types of NAT implementations are:

Stateless NAT: It involves mapping IP addresses from one subnet to another. An internal IPv6 or IPv4 address is mapped to an IPv4 address by translating the IP layer fields. Each translation is done without considering previous translations.

Stateful NAT: It maps many internal IP addresses to a few public IP addresses. Here, the translation requires manipulating the transport layer of the packets. The state is stored, i.e., the router is aware of the previous translations. A single public IP address can be used for all private IPs with different ports.

B. Extended Berkeley Packet Filter (eBPF)

eBPF technology is used to create sandboxed programs in Linux OS. It is a way to extend the functionality of the Linux kernel. The kernel is not very customizable and is hard to evolve due to the requirement of stability for all kernel users. Thus, eBPF is a very useful tool that helps add more functionality and innovation to the kernel capabilities. Today, eBPF is used for a lot of applications like load balancing on modern data centers, extracting fine-grained data from the kernel at a low overhead, enabling code tracing and debugging in the kernel, and many more. eBPF allows the programmers to write their own code to run in the Linux kernel, thus offering a lot of flexibility when it comes to creating applications.

eBPF allows placing bits of code into the kernel at various places for multiple purposes. One of them is processing the network packets and enabling the building of fully programmable data planes. Linux kernel already supports NAT, but eBPF offers better performance in terms of faster packet processing and the ability to customize the NAT features.

eBPF programs are event-driven and are triggered when the kernel or application satisfies a check condition or passes a certain hook point. These conditions could be anything from a system call to a certain network event. These are called eBPF hooks, and act as trigger points after which the eBPF program is given control. The eBPF program passes through 2 checks before being attached to the requested hook.

The first stage is the *Verification* stage, where the program is checked for many conditions like (i) Loading the eBPF program has the required permissions (ii) It does not access memory out of score or crash or harm the system in any way (iii) It is able to execute completely, i.e., run to completion.

The second stage is the *Just In Time (JIT) compilation* phase. The LLVM compiler converts the high-level eBPF program to raw byte code using the JIT compiler. This bytecode is attached to the kernel instructions during runtime. This makes the eBPF program run as effectively as compiled kernel code.

Another important aspect of eBPF is the ability to share collected information and to store state. This is done by eBPF Maps. eBPF maps use a variety of data structures to help store information in memory for fast lookup during runtime.

C. TC-BPF

Traffic control (tc) is a tool that is used to configure the kernel packet scheduler. *tc* can be used to simulate packet delay, limit bandwidth, and modify queue disciplines (qdiscs) of networking interfaces. This helps in testing network performance under various conditions. The *tc* configurations are to be specified for an interface. tc supports classifiers (cls) and actions (act) as BPF programs which can be attached to hook points provided by ingress and egress qdiscs. We use the *cls* hook point of the kernel to attach the eBPF program. The eBPF program gets attached to the ingress or egress interface of the device. The advantage of eBPF is that it provides a generic framework, wherein users can specify their highly specialized use cases. Besides, the availability of eBPF map data structures implies that statistics or other information can easily be stored and accessed with low overhead.

D. XDP

XDP is a relatively new concept that lets packets be processed faster than the normal network stack. It requires a modification to the kernel and allows packets to move through an integrated fast path in the kernel stack. It does not replace TCP/IP and is not a kernel bypass. An analogy that helps understand XDP is that if the traditional network stack is a freeway, then kernel bypass is a proposal to build an infrastructure for high-speed trains and XDP is a proposal for adding carpool lanes to the freeway. So XDP is like a faster path through the kernel stack.

XDP is a path for packets in the mainline Linux kernel, that allows for programmable packet processing. In the ordinary kernel stack, the packet passes through the networking layers and the future path of the packet is based on the destination address, the correctness of the packet, etc. However, there are many use cases where we might want to personally customize the path of the packet based on its internals. In today's industry, high-performance packet processing in software requires very tight bounds on the time spent processing each packet.

There are multiple solutions to the above problem. Special-purpose toolkits are available for software packet processing, such as the Data Plane Development Kit (DPDK). Such toolkits generally bypass the operating system completely, instead of passing control of the network hardware directly to the network application and dedicating one, or several, CPU cores exclusively to packet processing. On the other hand, XDP is a system that adds programmability directly to the operating system networking stack in a cooperative way. This makes it possible to perform high-speed packet processing that integrates seamlessly with existing systems, while selectively leveraging functionality in the operating system.

XDP works by defining a limited execution environment in the form of a virtual machine running eBPF code, an extended version of the original BSD Packet Filter (BPF) byte code format. This environment executes custom programs directly in the kernel context before the kernel itself touches the packet data, which enables custom processing (including redirection) at the earliest possible point after a packet is received from the hardware. The kernel ensures the safety of the custom programs by statically verifying them at load time, and programs are dynamically compiled into native machine instructions to ensure high performance.

III. Implementation of NAT44 and NAT64

A simple topology consisting of one internal host, one public host and one NAT router, as shown in Fig. 1, is used to validate all the implementations discussed in this section. This setup is created using Network Stack Tester (NeST) [7], which internally uses Linux network namespaces to emulate these nodes on a single physical machine. All the implementations discussed in this section are available in the GitHub repository[1]. The filenames used in the following sub-sections refer to the files that are available in this repository.

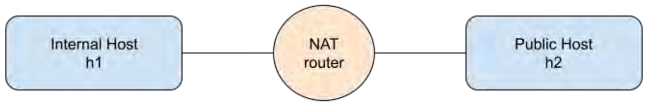


Fig. 1: Testbed Topology

A. NAT44 using TC-BPF

NAT44 performs address translation from IPv4 to IPv4. The use case for this type of NAT is when a local network is IPv4 enabled and clients from this network want to reach an IPv4-enabled endpoint outside the network, but one which uses a different subnet. A great example of this is the way campus Internet translates the IPv4 addresses of connected internal devices to a public IPv4 address, in a one-to-one fashion. We are using eBPF maps to store the state while performing NAT. eBPF Maps are data structures that allow fast lookup in the kernel. Although we use the term 'state' in the above explanation, the NAT that we have performed is classified as 'stateless'. The state that the implementation is talking about is merely checking if a translation is already done for the given input address, and if the timer for this state has expired.

We set the IPv4 addresses for the interfaces connecting 'h1' to the 'router' and those connecting the 'router' to 'h2'. The IPv4 addresses for each of the pairs belong to different subnets. We have 4 main components in our implementation of NAT44:

1) The eBPF program loader: This is implemented in the file *nat44.c*. This file has methods to parse and load the eBPF program into the kernel.

2) The IPv4 internal packet handler: This is done by the method *static int nat64_handle_ingress(struct __sk_buff *skb, struct hdr_cursor *nh)*. This method receives the IPv4 packet, checks its validity, and then checks if a state is already allocated for the packet's IPv4 source address. If there is no state, then it allocates a new state for this address. Otherwise, it checks if the state is valid. Subsequently, it changes the packet's fields to the new state details, thereby changing the packet's IP layer, including the destination address. It performs the IPv4 to IPv4 address translation by a simple map between IPv4 to IPv4 address space.

3) The IPv4 packet handler: This is done by the method *static int nat64_handle_v4(struct __sk_buff *skb, struct hdr_cursor *nh)*. This method receives the IPv4 internal packet, checks its validity, and checks if a state exists already. If a state does not exist, it immediately drops the packet. Otherwise, it changes the IP layer fields in the packet, thereby updating the destination address.

4) The state allocator: It is done by a method *static struct v4_addr_state *alloc_new_state(__u32 *internal_src_v4)*. It is used to allow a new state to the IPv4 handler and returns a new state for IPv4 addresses that have expired state.

B. NAT64 using TC-BPF

NAT64 performs address translation from IPv6 to IPv4 by mapping many IPv6 addresses used internally to one or few IPv4 addresses. The use case for this type of NAT is when a local network is IPv6 enabled and clients from this network want to reach an IPv4-enabled endpoint outside the network. Like in case of NAT44 implementation, we use eBPF maps to store the state while performing NAT. As mentioned in the previous sub-section, the usage of the word 'state' is only about checking if a translation is already done for the given input address, and if the timer for this state has expired. The NAT64 that we have implemented is actually 'stateless'.

We set the IPv6 addresses for the interfaces connecting 'h1' to the 'router' and IPv4 addresses to those connecting the 'router' to 'h2'. Similar to the NAT44 implementation, we have 4 main components in our implementation of NAT64:

1) The eBPF program loader: This is implemented in the file *nat64.c*. It has methods to parse and load the eBPF program into the kernel.

2) The IPv6 packet handler: This is done by the method *static int nat64_handle_v6(struct __sk_buff *skb, struct hdr_cursor *nh)*. It receives the IPv6 packet, checks its validity, and then checks if a state is already allocated for the packet's IPv6 source address. If there is no state, then it allocates a new state for this address. Otherwise, it checks if the state is valid. Subsequently, it changes the packet's fields to the new state details, thereby changing the packet's IP layer, including the destination address. For the IPv6 to IPv4 translation, a hash type eBPF map is used to store state. State refers to the IPv4 address, a last seen to keep track of expired addresses and additional configurations, if required. For the reverse process too, i.e., IPv4 to IPv6, a hash type eBPF map is used. We use a *Trie* type eBPF map to store the allowable

[1]https://github.com/steps-to-reproduce/nat-and-nat64

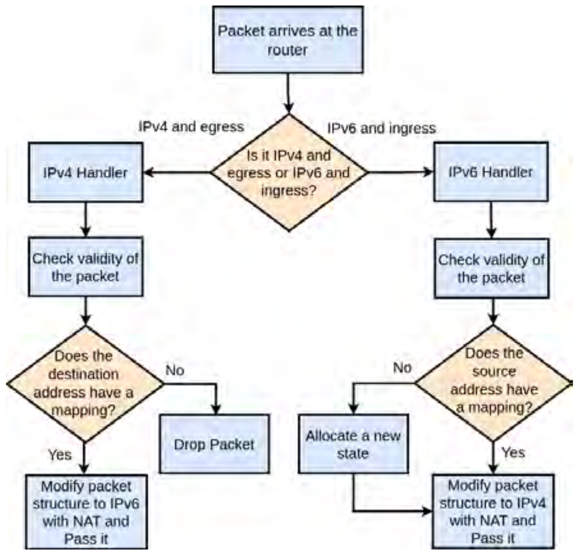

Packet arrives at the router
IPv4 and egress
IPv6 and ingress
Is it IPv4 and egress or IPv6 and ingress?
IPv4 Handler
IPv6 Handler
Check validity of the packet
Check validity of the packet
Does the destination address have a mapping?
No
Drop Packet
Yes
Modify packet structure to IPv6 with NAT and Pass it
Does the source address have a mapping?
No
Allocate a new state
Yes
Modify packet structure to IPv4 with NAT and Pass it

Fig. 2: Path of a packet in TC-BPF NAT router

source IPv6 addresses, and another queue type map to store the reclaimed addresses.

3) The IPv4 packet handler: This is done by the method *static int nat64_handle_v4(struct __sk_buff *skb, struct hdr_cursor *nh)*. It receives the IPv4 packet, checks its validity, and whether a state exists already. If it does not exist, the packet is dropped. Otherwise, it changes the IP layer fields in the packet, thereby updating the destination address.

4) The state allocator: This is done by the method *static struct v6_addr_state *alloc_new_state(struct in6_addr *src_v6)*. It is used to allow a new state to the IPv6 handler. It returns a new state for IPv6 addresses that have expired state.

Fig. 2 shows the steps involved in processing a packet in the NAT router for NAT44 and NAT64 using TC-BPF.

C. NAT64 using XDP

This section provides the details about NAT64 implementation by using XDP. This is implemented in 2 major sections: the SEC("v6_side") to handle the IPv6 traffic and SEC("v4_side") to handle the IPv4 traffic:

1) IPv6 traffic: This is handled by *int xdp_nat_v6_func(struct xdp_md *ctx)* method. It receives the IPv6 packet and checks if it needs to be translated. If the packet needs to be translated from IPv6 to IPv4 then it crafts the IPv4 header from the IPv6 header and replaces the IPv6 header in the packet with the IPv4 header. In case there is an ICMPv6 header on top of IPv6 header, it parses the ICMPv6 header and crafts an ICMP header and finally replaces the ICMPv6 header with the ICMP header. After the translation, a call is made to the lookup table and the destination address on the IPv4 packet is used to find the next hop in its path. Finally, the packet is forwarded to the respective interface.

2) IPv4 traffic: This is handled by *int xdp_nat_v4_func(struct xdp_md *ctx)* method. It receives the IPv4 packet and checks for translation. If it needs to be translated from IPv4 to IPv6 then it crafts the IPv6 header from the IPv4 header and replaces the IPv4 header in the packet with the IPv6 header. If there is an ICMP header on top of IPv4 header, it parses the ICMP header and crafts an ICMPv6 header and replaces the ICMP header with the ICMPv6 header. After the translation, a call is made to the lookup table and the destination address on the IPv6 packet is used to find the next hop in its path. Finally, the packet is forwarded to the respective interface. Fig. 3 shows the steps involved in processing a packet in NAT64 router using XDP.

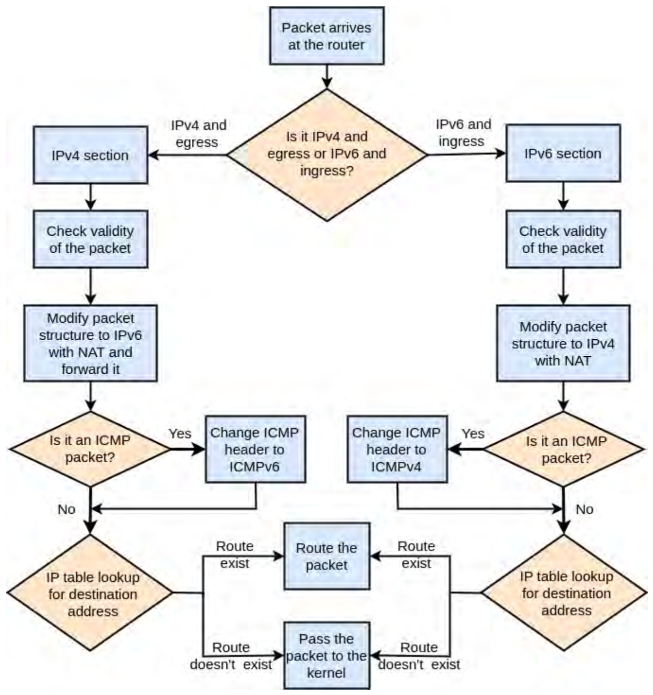

Packet arrives at the router
IPv4 and egress
IPv6 and ingress
Is it IPv4 and egress or IPv6 and ingress?
IPv4 section
IPv6 section
Check validity of the packet
Check validity of the packet
Modify packet structure to IPv6 with NAT and forward it
Modify packet structure to IPv4 with NAT
Is it an ICMP packet?
Yes
Change ICMP header to ICMPv6
Change ICMP header to ICMPv4
Yes
Is it an ICMP packet?
No
No
IP table lookup for destination address
Route exist
Route the packet
Route exist
IP table lookup for destination address
Route doesn't exist
Pass the packet to the kernel
Route doesn't exist

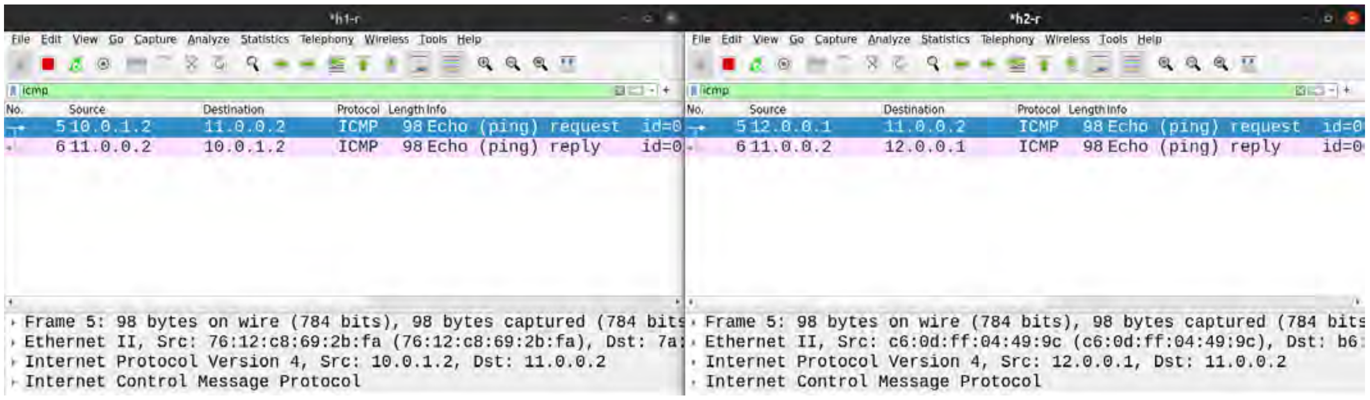
Fig. 3: Path of a packet in XDP NAT router

IV. Validation of NAT Implementations

To validate the NAT44 and NAT64 implementations proposed in this paper, we setup a topology as shown in Fig. 1 using NeST [7]. IPv4/IPv6 addresses are configured on the interfaces of all the nodes. We use Wireshark to check for the IPv4/IPv6 packets and to check the correctness of NAT.

1) Validation of NAT44 Implementation using TC-BPF: After setting the appropriate IPv4 addresses on all the interfaces of the nodes, we call the executable file *nat44* (obtained after running make on the *nat44.c* file). The *nat44.c* file has all the functions to load the eBPF code. It loads the eBPF code and calls the n*at44_kern.c* functions. These are the functions that handle the different scenarios during the translation as mentioned in detail above. Figures 4 and 5 show the packet traces from Wireshark. Fig. 4 shows that an IPv4 address from the private subnet (10.0.1.2 on the left) is translated to a public subnet (12.0.0.1 on the right), and Fig. 5 shows the reverse.

2) Validation of NAT64 Implementation using TC-BPF: After setting the appropriate IPv4 and IPv6 addresses on all the interfaces of the nodes, we call the executable file *nat64* (obtained after running make on the *nat64.c* file). The *nat64.c* file has all the functions to load the eBPF code. It loads the eBPF code and calls the *nat64_kern.c* functions. These


*h1-r
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
icmp
*h2-r
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
icmp

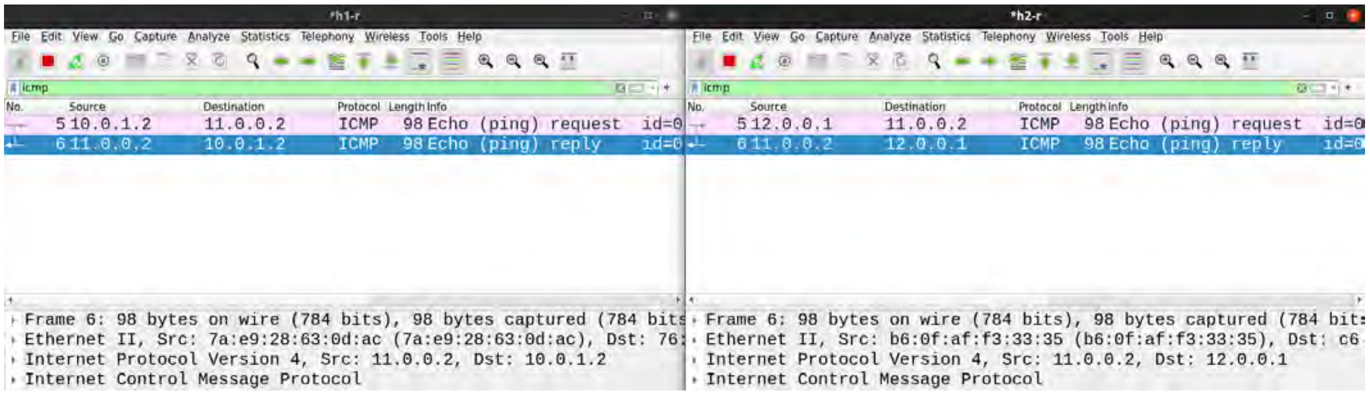
```
No.  Source      Destination  Protocol Length Info
  5 10.0.1.2    11.0.0.2     ICMP     98 Echo (ping) request  id=0
  6 11.0.0.2    10.0.1.2     ICMP     98 Echo (ping) reply    id=0

Frame 5: 98 bytes on wire (784 bits), 98 bytes captured (784 bits
Ethernet II, Src: 76:12:c8:69:2b:fa (76:12:c8:69:2b:fa), Dst: 7a:
Internet Protocol Version 4, Src: 10.0.1.2, Dst: 11.0.0.2
Internet Control Message Protocol
```

```
No.  Source      Destination  Protocol Length Info
  5 12.0.0.1    11.0.0.2     ICMP     98 Echo (ping) request  id=0
  6 11.0.0.2    12.0.0.1     ICMP     98 Echo (ping) reply    id=0

Frame 5: 98 bytes on wire (784 bits), 98 bytes captured (784 bits
Ethernet II, Src: c6:0d:ff:04:49:9c (c6:0d:ff:04:49:9c), Dst: b6:
Internet Protocol Version 4, Src: 12.0.0.1, Dst: 11.0.0.2
Internet Control Message Protocol
```

Fig. 4: Forward path of the packet with NAT44


*h1-r
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
icmp
*h2-r
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
icmp

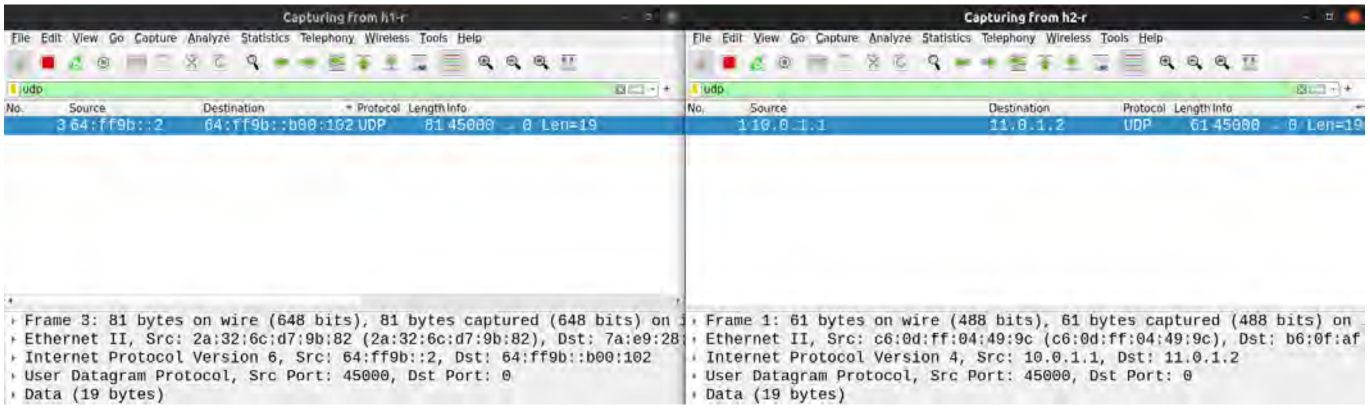
```
No.  Source      Destination  Protocol Length Info
  5 10.0.1.2    11.0.0.2     ICMP     98 Echo (ping) request  id=0
  6 11.0.0.2    10.0.1.2     ICMP     98 Echo (ping) reply    id=0

Frame 6: 98 bytes on wire (784 bits), 98 bytes captured (784 bits
Ethernet II, Src: 7a:e9:28:63:0d:ac (7a:e9:28:63:0d:ac), Dst: 76:
Internet Protocol Version 4, Src: 11.0.0.2, Dst: 10.0.1.2
Internet Control Message Protocol
```

```
No.  Source      Destination  Protocol Length Info
  5 12.0.0.1    11.0.0.2     ICMP     98 Echo (ping) request  id=0
  6 11.0.0.2    12.0.0.1     ICMP     98 Echo (ping) reply    id=0

Frame 6: 98 bytes on wire (784 bits), 98 bytes captured (784 bit:
Ethernet II, Src: b6:0f:af:f3:33:35 (b6:0f:af:f3:33:35), Dst: c6
Internet Protocol Version 4, Src: 11.0.0.2, Dst: 12.0.0.1
Internet Control Message Protocol
```

Fig. 5: Return path of the packet with NAT44


Capturing from h1-r
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
udp
Capturing from h2-r
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
udp

```
No.  Source        Destination        Protocol Length Info
  3 64:ff9b::2    64:ff9b::b00:102   UDP      81 45000 → 0 Len=19

Frame 3: 81 bytes on wire (648 bits), 81 bytes captured (648 bits) on i
Ethernet II, Src: 2a:32:6c:d7:9b:82 (2a:32:6c:d7:9b:82), Dst: 7a:e9:28
Internet Protocol Version 6, Src: 64:ff9b::2, Dst: 64:ff9b::b00:102
User Datagram Protocol, Src Port: 45000, Dst Port: 0
Data (19 bytes)
```

```
No.  Source      Destination  Protocol Length Info
  1 10.0.1.1    11.0.1.2     UDP      61 45000 → 0 Len=19

Frame 1: 61 bytes on wire (488 bits), 61 bytes captured (488 bits) on
Ethernet II, Src: c6:0d:ff:04:49:9c (c6:0d:ff:04:49:9c), Dst: b6:0f:af
Internet Protocol Version 4, Src: 10.0.1.1, Dst: 11.0.1.2
User Datagram Protocol, Src Port: 45000, Dst Port: 0
Data (19 bytes)
```

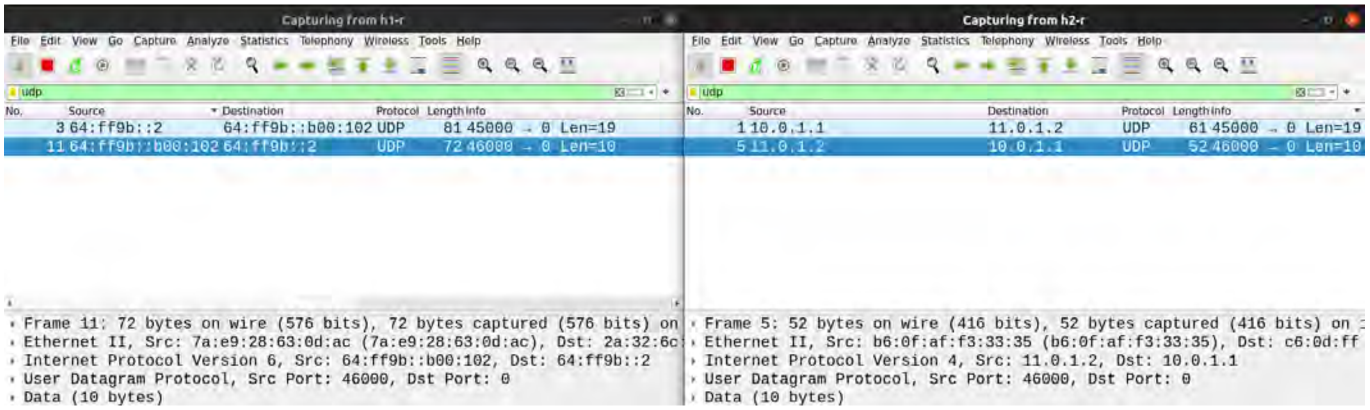
Fig. 6: Forward path of the packet with NAT64

are the functions that handle the different scenarios during the translation as mentioned in detail above. Figures 6 and 7 show packet traces from Wireshark. Fig. 6 shows that an IPv6 address is translated to IPv4, and Fig. 7 shows the reverse.

3) Validation of NAT64 Implementation using XDP: After setting the appropriate IPv4 and IPv6 addresses on all the interfaces of the nodes, we load the XDP program in the *xdp_prog_kern.c* file on the ingress interface and egress interface of the router. The eBPF program to handle IPv6 packets is loaded on the IPv6 interface and the eBPF program to handle IPv4 packets is loaded on the IPv4 interface. These programs are triggered every time a packet passes through the interfaces. Figures 8 and 9 show the packet traces obtained using Wireshark. Fig. 8 shows that an IPv6 address is translated to an IPv4 address, and Fig. 9 shows the reverse.

V. Conclusions and Future work

In this paper, we focused on the implementation of NAT using newer technologies that provide a faster path for packet transmission and reception. We discussed the implementation of NAT64 using TC-BPF and XDP, and NAT44 using TC-BPF. We have also verified the translation by using the packet traces. This would serve as a good baseline for further improvements on NAT in XDP. Both TC-BPF and XDP are technologies that can be leveraged for their flexibility to perform NAT in routers. NAT implemented using XDP can save a lot of overhead, as well as offer flexibility to the programmers. eBPF has inbuilt support for eBPF maps that offer a fast lookup of data in the kernel. Owing to these advantages, NAT using XDP and eBPF can be considered for industry applications.


Capturing from h1-r
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
udp
Capturing from h2-r
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
udp

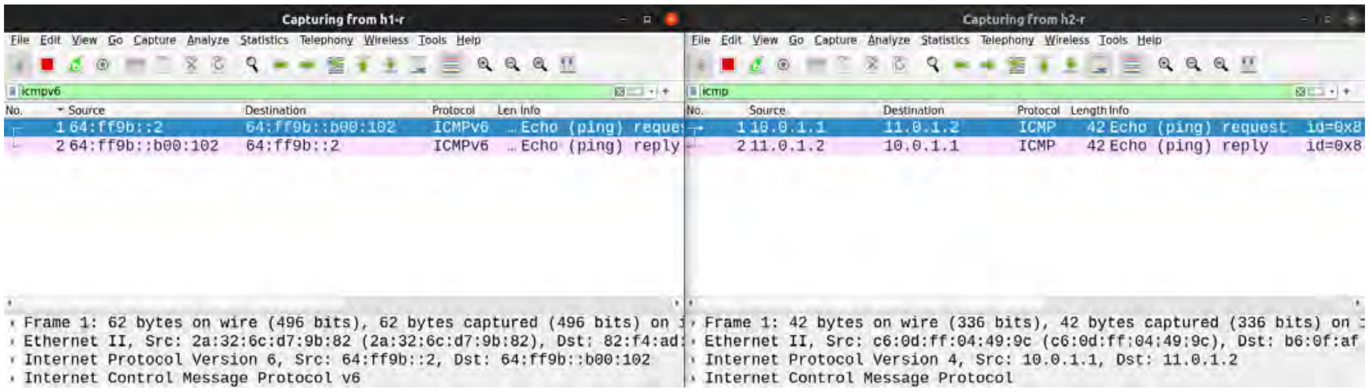
```
No.   Source              Destination         Protocol Length Info
    3 64:ff9b::2          64:ff9b::b00:102    UDP      81 45000 → 0 Len=19
   11 64:ff9b::b00:102    64:ff9b::2          UDP      72 46000 → 0 Len=10

Frame 11: 72 bytes on wire (576 bits), 72 bytes captured (576 bits) on
Ethernet II, Src: 7a:e9:28:63:0d:ac (7a:e9:28:63:0d:ac), Dst: 2a:32:6c
Internet Protocol Version 6, Src: 64:ff9b::b00:102, Dst: 64:ff9b::2
User Datagram Protocol, Src Port: 46000, Dst Port: 0
Data (10 bytes)
```

```
No.   Source      Destination   Protocol Length Info
    1 10.0.1.1    11.0.1.2      UDP      61 45000 → 0 Len=19
    5 11.0.1.2    10.0.1.1      UDP      52 46000 → 0 Len=10

Frame 5: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on
Ethernet II, Src: b6:0f:af:f3:33:35 (b6:0f:af:f3:33:35), Dst: c6:0d:ff
Internet Protocol Version 4, Src: 11.0.1.2, Dst: 10.0.1.1
User Datagram Protocol, Src Port: 46000, Dst Port: 0
Data (10 bytes)
```

Fig. 7: Return path of the packet with NAT64


Capturing from h1-r
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
icmpv6
Capturing from h2-r
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
icmp

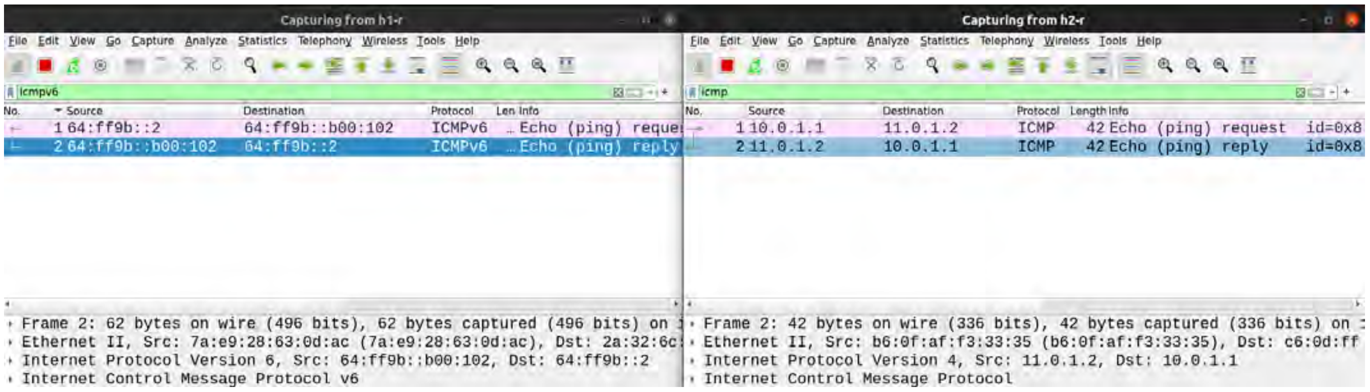
```
No.   Source              Destination         Protocol Len Info
    1 64:ff9b::2          64:ff9b::b00:102    ICMPv6   … Echo (ping) reque
    2 64:ff9b::b00:102    64:ff9b::2          ICMPv6   … Echo (ping) reply

Frame 1: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on i
Ethernet II, Src: 2a:32:6c:d7:9b:82 (2a:32:6c:d7:9b:82), Dst: 82:f4:ad:
Internet Protocol Version 6, Src: 64:ff9b::2, Dst: 64:ff9b::b00:102
Internet Control Message Protocol v6
```

```
No.   Source      Destination   Protocol Length Info
    1 10.0.1.1    11.0.1.2      ICMP     42 Echo (ping) request  id=0x8
    2 11.0.1.2    10.0.1.1      ICMP     42 Echo (ping) reply    id=0x8

Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on
Ethernet II, Src: c6:0d:ff:04:49:9c (c6:0d:ff:04:49:9c), Dst: b6:0f:af
Internet Protocol Version 4, Src: 10.0.1.1, Dst: 11.0.1.2
Internet Control Message Protocol
```

Fig. 8: Forward path of the packet with NAT64 using XDP


Capturing from h1-r
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
icmpv6
Capturing from h2-r
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
icmp

```
No.   Source              Destination         Protocol Len Info
    1 64:ff9b::2          64:ff9b::b00:102    ICMPv6   … Echo (ping) reque
    2 64:ff9b::b00:102    64:ff9b::2          ICMPv6   … Echo (ping) reply

Frame 2: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on i
Ethernet II, Src: 7a:e9:28:63:0d:ac (7a:e9:28:63:0d:ac), Dst: 2a:32:6c:
Internet Protocol Version 6, Src: 64:ff9b::b00:102, Dst: 64:ff9b::2
Internet Control Message Protocol v6
```

```
No.   Source      Destination   Protocol Length Info
    1 10.0.1.1    11.0.1.2      ICMP     42 Echo (ping) request  id=0x8
    2 11.0.1.2    10.0.1.1      ICMP     42 Echo (ping) reply    id=0x8

Frame 2: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on
Ethernet II, Src: b6:0f:af:f3:33:35 (b6:0f:af:f3:33:35), Dst: c6:0d:ff
Internet Protocol Version 4, Src: 11.0.1.2, Dst: 10.0.1.1
Internet Control Message Protocol
```

Fig. 9: Return path of the packet with NAT64 using XDP

The future work is to implement a stateful NAT using XDP and TC-BPF. Stateful NAT implementations take into account port numbers along with IP addresses and use only a few IP addresses for all the private users with different port numbers. This helps keep the IPv4 address space from running out of valid addresses. Additionally, in this work, we limited ourselves to a topology with a single host on either side of a router. A testbed with many more hosts and routers can be set up for more realistic testing and performance benchmarking to quantify the improvements offered by TC-BPF and XDP.

References

[1] "Google Networks - IPv6 Adoption by Countries." 2022. [Online]. Available: https://www.google.com/intl/en/ipv6/statistics.html#tab=per-country-ipv6-adoption

[2] M. Bagnulo, P. Matthews, and I. van Beijnum, "RFC 6146 - Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers," Tech. Rep., 2011.

[3] M. Bagnulo, A. García-Martínez, and I. Van Beijnum, "The NAT64/DNS64 Tool Suite for IPv6 Transition," *IEEE Communications Magazine*, vol. 50, no. 7, pp. 177–183, 2012.

[4] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel," in *Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies*, 2018, pp. 54–66.

[5] "eBPF Technology: An Introduction and Deep Dive into the eBPF Technology." 2022. [Online]. Available: https://ebpf.io/what-is-ebpf/

[6] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[7] Rai, Shanthanu S., G., Narayan, M., Dhanasekhar, Monis, Leslie and Tahiliani, Mohit P., "NeST: Network Stack Tester," in *Proceedings of the Applied Networking Research Workshop*, ser. ANRW '20, 2020, p. 32–37.