Master thesis : Web-scale Network Telemetry for live user traffic: In-band OAM for the Linux Kernel

Auteur : Francart, Jonathan
Promoteur(s) : Donnet, Benoît
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "computer systems and networks"
Année académique : 2017-2018
URI/URL : https://github.com/FrancartJ/iOAM-IPv6-LinuxKernel; http://hdl.handle.net/2268.2/4582

UNIVERSITY OF LIÈGE

MASTER THESIS

In-band OAM in the Linux Kernel

Author:
Jonathan FRANCART

Supervisor:
Benoit DONNET
Frank BROCKNERS

Graduation Studies conducted for obtaining the Master's degree in Computer Science and Engineering

Academic Year 2017-2018

UNIVERSITY OF LIÈGE

Abstract

Faculty of Applied Sciences

Computer Science and Engineering

In-band OAM in the Linux Kernel

by Jonathan FRANCART

The measurement methods are important in the network to monitor it. Companies having a hypergiant CDN or a big data center need some mechanisms to detect an issue and quickly find its origin to be able to fix it. Indeed, due to some issue, it is possible that a whole part of the service proposed by a company will be unavailable during a certain amount of time. Consequently, it is normal to see the apparition of new methods which fix some issue like Paris traceroute for traceroute, that are more efficient or allow to collect more information. One of them which allows to collect more information, is the subject of this thesis. It is iOAM.

This work tempts to put the basics needed by iOAM in the Linux kernel. Indeed, not all the possibilities bring by this method are implemented during this thesis but now, there is a starting point to continue the development of it in the Linux kernel. The code of the implementation made during this master thesis can be retrieved on Git Hub [1]. It allows a node having Linux as an operating system to understand and perform the appropriate actions when it receives an IPv6 packet containing an iOAM pre-allocate trace inside the Hop-By-Hop options header.

The performances of this implementation are quite promising. They highlight the link between the percentage of packet which will receive the iOAM trace and the throughput. Also they put on the way of possible improvements for the recording of data.

Acknowledgements

I would like to express my acknowledgements to the following people:

- My two supervisors Prof. Donnet and Frank Brockners which have taken the time to guide and help me during this work. I am grateful for their assistance and the means made available in order to realize this master thesis.
- My family and friends for helping me and supporting me during my studies. Without them, it would not have been possible.

Contents

List of Figures

Chapter 1

Introduction

Nowadays, the system administrators need to constantly monitor their network. This can be done through measurements. Thanks to those measurements, if a problem occurs on their network, they can investigate, possibly find the cause and, finally, apply the appropriate actions to fix it. For the moment, there are mainly two ways for performing measurement on the network: the passive and the active measurement.

The passive measurement is a process which allows somebody to get some data about a specific node of the network without injecting new packets or modifying existing ones. Some software or hardware are placed on the network devices to examine the traffic. It is like in a hospital when a doctor uses an electrocardiogram to see how is the hearth of a patient listening to its pulse. One software tools to perform some passive measurement is tcpdump [2]. It records all the packets going through a specific interface of a network device. With those records, a system administrator can see, for example, the bit or the packet rate. It can also apply some filter to only see packets using some specific protocol. Tcpdump is not the only tool which allows performing such measurements indeed, there are other tools like wireshark[3].

But the system administrator can also ask the hardware to get some specific information. Indeed, there are tools, like snmp (simple network management protocol) which allow an administrator to speak with its network devices like routers, to retrieve some specific information or change their behavior. He can also get some BGP dumps to retrieve all the exchanged BGP messages to know for example if a new route has appeared or if an existing one has disappeared.

The active measurement defines techniques where the system administrator injects new packets into the network to get certain information about its state. There is a lot of tools which allow someone to do that. One of this tools is ping which allows the system administrator to send an ICMP echo packet to a specific host which will respond with an ICMP echo reply. It is used to test the reachability of a host. But it can also be used at the same time to compute the round trip time taken by the packet. Indeed, one can compute the RTT by subtracting the timestamp at which the reply has arrived by the timestamp at which the request has been sent.

Another tool which is used to perform some active measurement is traceroute. This tool allows knowing the number of hops crossed by a packet

on its path to reach the destination and the address of the ingress interface of each hop. It also allows detecting on which link there is an issue if any.

The problem with the active measurement techniques is that they generate new packets, therefore, they add new traffic in the network. Certainly, it is not always a good idea to inject new traffic in a sick network which has already some difficulties to deal with the existing packets because it will only increase the difficulties. But also the information given by those methods are not always rightly interpreted due to a lack of understanding.

Now, there is a new way to perform those measurements thanks to iOAM. In-band OAM describes techniques to collect and carry OAM which is the abbreviation for Operations Administration and Maintenance, or telemetry information within the existing packets while their journey in the network. Indeed, if a packet carries an iOAM pre-allocated trace in an Hop-By-Hop options header, each hop crossed by the packet must add the asked data or if they are not able to do, they must fill the data field with a specific value to notify it. It is like the record route option of IP which asks to each node on the path of the packet to put its IP address in the appropriate field.

Since iOAM adds new data in the existing packet, it can not be considered as a passive measurement method but because it does not create new packet and so increase the existing traffic, it can not either be taken as an active measurement method. There are three different iOAM options allowing to collect some specific information and they can be used together.

The first one is the trace option. This option like for the following one is placed in an Hop-By-Hop options header. Since it is placed in this particular extension header, each hop on the path of the packet must add if they are able to do it the asked information. Thanks to the collected information, the system administrator can know the time needed to go from one node to another, the time the packet has spent in the node and other useful information. All the possible information which can be collected thanks to it are described in section 3.2.

The second one is the proof of transit option. In that case, this options carry a specific value at its creation. Each node on the path if they are capable to do it, will update this value thanks to a piece of information that they have. Each piece of information must be known only by the node. Also, the operation performed must make the discovery of the value difficult. Otherwise, anyone could simulate the passage of its packet into some specific nodes which in fact are not crossed by it.

When the packet arrives in the iOAM egress node, it is possible to determine if the packet has crossed all the nodes it should have or not. Indeed, because the system administrator knows the value in each node and the operation performed, it can know what should be the final value and compare it to the value present in the packet. So, if the value in the option is the expected one, he is sure that the packet has gone through those nodes, and if it is not the case, it means that the packet has not crossed all the nodes which are normally on its path.

The last one is the edge to edge option. It is used to carry data added by the node which inserts the extension header in an existing IPv6

packet to the node which will extract it. This data can be a sequence number or a timestamp. It can be used to see packet loss, reordering or duplication for example.

In this master thesis, an implementation in the Linux kernel of the first option which is the trace option, will be depicted. The code of this implementation can be retrieved on Git Hub [1]. The rest of this documents will be organized as follow :

- Chapter 2 is a discussion about the IPv6 to bring back in mind important features which are used in this master thesis and also to see the improvement bring by this new version of the internet protocol compared to its predecessor IPv4. Moreover, it shows the evolution of the deployment of this new version in the world.
- Chapter 3 will depict in more detail the motivation behind this method which is iOAM and shows a comparison between iOAM and two existing tools which are ping and traceroute. After the motivational aspects, comes a description of the format of this technique which is at this moment still in discussion at the IETF.
- Chapter 4 shows the implementation made in the Linux kernel of the iOAM trace option and more particularly, the four main part of it which are the creation of the extension header, its insertion in an existing packet, its extraction and how the asked data are recorded in the packet.
- Chapter 5 will firstly depict the methodology used to asses the performances of this implementation. Then, those performance are highlighted by a discussion.
- Finally, the last chapter which is the sixth one, consists of a conclusion and a discussion about the possible improvements.

Chapter 2

IPv6

In the early 90's, it becomes evident that there will be a lack of available IPv4 addresses. On April 19, 2011, the RIR (Regional Internet Registry) APNIC (Asia-Pacific Network Information Center) has exhausted its general use pool of addresses [4] then it is the turn of RIPE NCC (RIPE Network Coordination Centre) on September 14, 2012. A few years later, on June 10, 2014, this time it is the LACNIC (Latin America and Caribbean Network Information Centre) which has exhausted its general use pool of addresses followed about one year later by the ARIN (American Registry for Internet Numbers). There is at this time one RIR which has not exhausted all its general use pool of IPv4 addresses. It is AFRINIC (African Network Information Centre) but it is projected to the date of May 18, 2019. Seeing that the pool of IPv4 addresses will be exhausted, the IETF had started, among others, to design a new version of the internet protocol which is the IPv6. The specification of this protocol was finalized in 1998[12]. But since some extension has been added to it.

One can think of why there is no mention about the fifth version of the internet protocol. The name of this version is the stream protocol. It is a very different protocol compared to the two version previously cited.

In this section, the issue coming from the fourth version of the internet protocol will be described first. Then the major difference between these two versions of the header will be described. After that, the advantages of this new version against the previous one (IPv4) will be highlighted. And finally, a review of the IPv6 deployment nowadays will be exposed.

2.1 Issue of IPv4

The main problem with IPv4 is the range of available addresses. Indeed, in this version of the internet protocol, the IP address has a size of 32 bits. So there are about 4.2 billions different addresses with this version.

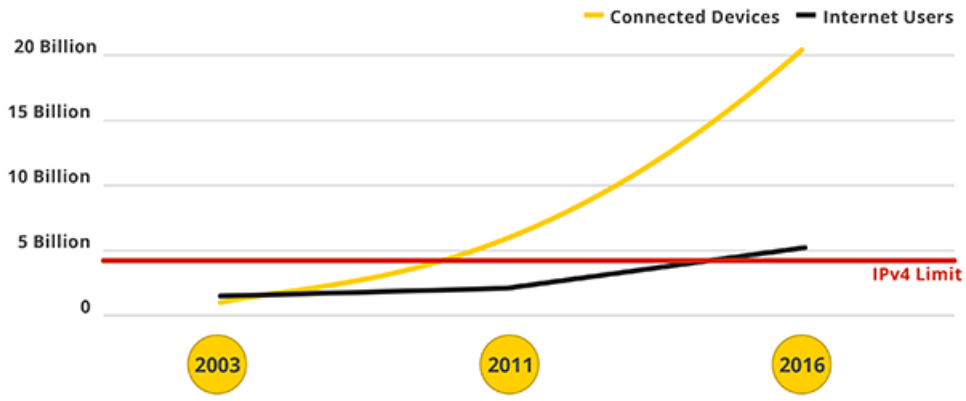


FIGURE 2.1: Exhaustion of IPv4 addresses (from Google)

As seen on Figure 2.1 since some years the number of IPv4 addresses are exhausted. At the end of the 80's, there are more human on earth than IPv4 addresses but at this time not every people has a device which was able to handle a connection on the network so this was not an issue. But, nowadays, most humans in developed countries have at least 2 connected devices.

Thanks to the NAT (network address translator) it is possible to deal with this number of connected devices which is higher than the number of possible IPv4 addresses. There are two types of IPv4 addresses: the public and private IP addresses. The private addresses can not be routed on the network since they are not unique on the whole network. Indeed, those addresses are behind a NAT which will change the source address of the packet sent by those devices having a private address, to replace it with its public address.

In the network, a flux can be identified by its source and destination address and its source and destination port. So the NAT must sometimes also change the source port because if for example, two different hosts want to connect to the same HTTP server they will have the same destination address and port and since the NAT changes the source address with its own, they will also have the same source address. Consequently, if they used the same source port, the NAT will change one of them to be able to make the distinction between the two and so when the server will send the response, be able to correctly forward it to the right host.

This NAT breaks one main law of the network which is the end to end principle. But thanks to it we are able to still add new devices on the network and also, it offers some protection. Indeed, if its is a dynamic NAT, it is not possible for someone which is not behind the NAT to initiate a connection or contact a host which is behind it. The connection must be started by the host behind it. But if it is a static NAT this is not true, it is possible to preconfigure some connection which allows some specific host in the network to contact a certain host which is behind the NAT. Now with the number of possible IPv6 addresses, there is no any more need of NAT but it will maybe still be used for security reasons.

Another advantage of the NAT is that it simplified the routing table because a lot of hosts shared the same public IP address. So it is maybe also

a good reason to sill use it.

2.2 IPv4 and IPv6 Headers format

As one can suppose, due to the issue noticed in the previous section, the length of IPv6 addresses is much larger than the length of IPv4 addresses. In this new version of the internet protocol, an address has a length of 128 bits which four times bigger than in IPv4. Consequently, it takes more space in the header.

The size of the address is not the only difference in the header format of this two versions of the internet protocol. Indeed, during the evolution of the internet some new need have appeared and the option field of the IPv4 protocol was not always convenient for it. So this new version has also been designed having that in the mind. But before going too much in detail of those differences, it is preferable to put back in mind the two versions of the header of the internet protocol. On the following Figure 2.2 is depicted the header of an IPv4 packet [14].

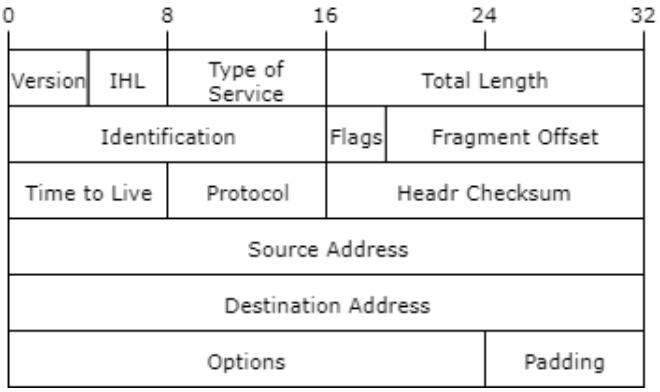


FIGURE 2.2: IPv4 header

- Version field : indicates the format of the internet header which is 4 in this case.
- IHL field : is the length of the internet header in 32 bits words.
- Type of service field : allows to give the adapted quality of service to the packet.
- Total length field : the length of the full packet in octets.
- Identification field : a value helping to assemble the fragments of the datagram.
- Flags field : a field of 3 bits :
  - bit 0 : reserved, must be zero
  - bit 1 : 0 may Fragment, 1 don't Fragment

  - bit 2 : 0 last fragment, 1 more fragment

- Fragment Offset : the offset of the fragment in the full datagram.
- TTL : number of hops before dropping the packet if it has not reached its destination.
- Protocol : indicates the next level protocol.
- Header Chekcsum : checksum of the header only to detect errors on it.
- Source Address : IPv4 source address which is 32 bits long.
- Destination Address : IPv4 destination address which is 32 bits long.
- Options : field to add options if needed.
- Padding : if the options field has not a length which is a multiple of 32 bits.

Now that it is done for the fourth version, it will also be done for the sixth version. The format of the header of an IPv6 packet [12] is displayed on Figure 2.3.

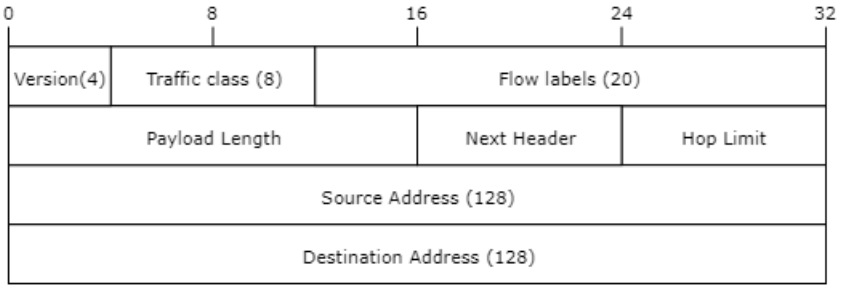


FIGURE 2.3: IPv6 header

- Version field : indicates the version of the internet protocol which is six in this case.
- Traffic class field : allows giving a specific service to the packet.
- Flow label field : allows giving a specific service to a flow of packets.
- Payload length : length of the payload in octets including the extension headers length.
- Next header field : contains the value according to some standard [5] to know the type of the following header which can be an extension header or a transport protocol header.
- Hop limit field : is set by the sender and decremented each time the packet is going through a hop. Like that, a packet will not be forever in the network.

- Source Address : IPv6 source address which is 128 bits long.
- Destination Address : IPv6 destination address which is 128 bits long.

Now that the two versions have been put back in mind, the comparison can be done. One can see some difference between the two headers format. Indeed, given that now the options are not anymore directly placed in the header of the internet protocol, and also due to the constant size of the IPv6 header (40 bytes), the IHL field has disappeared. But because the options are not anymore in the header once need to know what is the next header type to correctly process it since this next header can be a next layer header like a TCP one, but also an extension header. So the protocol field has disappeared to be replaced by the next header field. The different extension headers are described in more detail in the following subsection.

Also, the flags and fragment offset field have vanished because an IPv6 packet usually can not be fragmented since each node must execute a path MTU (maximum transfer unit) discovery to detect what is the maximum size allowed for a packet on the path which will be used to reach the destination. So it is the upper layer which must deal with it to correctly shape the packet. But if it is not possible, the sender host still can fragment an IPv6 packet thanks to the fragment extension header. However it is the sender host which must do the fragmentation not like for the previous version. Indeed this modification removes some work for the routers given that previously, if they received a packet which had a bigger size than the MTU of the link on which the packet should be forwarded, the router had two options. The first one is to drop the packet and to send and ICMP message with the packet too big code. The second one is to fragment the packet. The last option requires the router to perform some computation because it needs to recreate the header of the packet with the new parameters and to recompute the checksum of the packet.

2.2.1 IPv6 extension header

There is a list of possible extension headers which can be added after the IPv6 one. They must be in a certain order and one can know which is the type of the following header thanks to the next header field. In the following Table 2.1, the possible extension headers are listed [11].

As seen in the previous Table 2.1, one of the extension headers is the Hop-by-Hop options. It is used when someone wants that every host which are on the path of a packet perform some actions on it. The format of this extension header [12] is displayed on Figure 2.4 :

Extension Header	Code	Description
Hop-by-Hop Options	0	Need to be examined by each hop on the path
Routing Header	43	Methods to specify the route of a packet
Fragment Header	44	Contains parameters for the fragmentation of a datagram
Encapsulating Security Payload	50	Carries encrypted data for secure communication
Authentication Header	51	Contains information used to verify the authenticity
Destination Options	60	Need to be examined only by the destination
Mobility Header	135	Parameters used with Mobile IPv6 [10]
Host Identity Protocol	139	Used for Host Identity Protocol version 2 [20]
Shim6 Protocol	140	Used for Shim6 [18]
Use for experimentation and testing	253	Used for experimentation and testing
Use for experimentation and testing	254	Used for experimentation and testing

TABLE 2.1: IPv6 extension header code

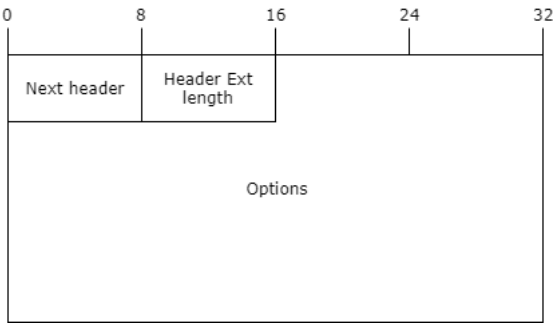


FIGURE 2.4: Hop-by-Hop Options header

The first field is the same than the fifth field of the IPv6 header. The second one is the length of the Hop-by-Hop options header in 8 octets without taking into account the first 8 octets.

Then there are the options. This field has variable length and the data in it have the TLV format. This format [12] is described in Figure 2.5 :

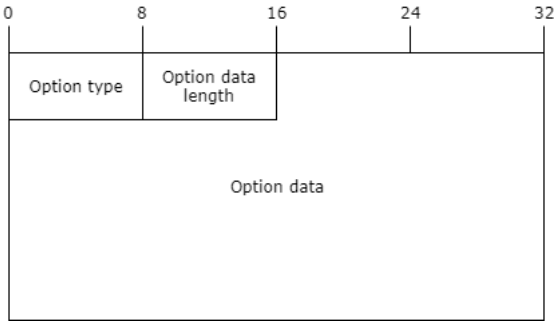


FIGURE 2.5: Options format

The TLV abbreviation is for type-length-value. So it is normal that the first field of this format is for the type of the option, then its length and finally the data. Every extension headers must have a length which is a multiple of 8 octets so if the length of the options present in the header does not full fill this criterion, a padding must be added and its format [12] is depicted on Figure 2.6.

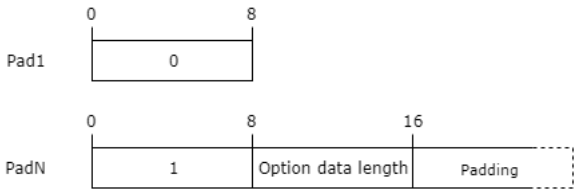


FIGURE 2.6: Padding format

There is two types of padding. The first one is when it misses only one octet and the second one is of course, when more are missing. In the first case, there is only an octet filled with 0 but in the last case, the padding is also in the TLV format. The type of the padding is 1 and its length must be smaller or equal to 7. The data for a padding must only be composed of 0.

2.3 Stateless Address Autoconfiguration

Prior to IPv6, when a host wanted to connect on the network, an IPv4 address could be assigned dynamically through DHCP (Dynamic Host Configuration Protocol). This protocol defined the following steps to allow an host to get its IPv4 address:

1. The host must broadcast a message to ask if there is a DHCP server.
2. If there is some DHCP server, they will respond to the host and propose to it an available IPv4 address.
3. Then the host chooses an IPv4 address in the ones offered and broadcasts its respond saying that it will takes this particular address.
4. The DHCP server which has offered the address, answers to the host to notify it that this address belongs to it for a certain time period.

5. If the host is still connected to the network some instant before the end of the period during which the IP address belongs to it, it must send a message to the DHCP server to notify it that it will keep this address for the next period.

The last step must be done until the host disconnects itself from the network. When the DHCP server provides the address to the host, in the message it also gives some important information to it like the address of the gateway router.

With IPv6, address assignment works differently as it is based an SLAAC (Stateless Address Autoconfiguration) [21].

When an interface activates, it must get a link-local address. This address is used by a host to exchange packet within the network segment and so will not be forwarded by the router. Those addresses are unique only in the network segment, it is possible to find the same address in another segment. To get a link-local address, the interface uses the specific prefix which is the fe80::0 one. It appends at the end an identifier of N-bit. This identifier can be its MAC address so 48 bits or an EUI64 which has as its name suggests, 64 bit. After that, the interface sends a multicast message to all neighbors to detect if its address is unique in this network segment. If its a duplicate address, the process stops and an administrator is needed to get ones. In the other case, the interface can now try to get its global address to be able to send and receive packet beyond its network segment. For that, it is needed to get the prefix of this area in the network. This information is contained in router advertisement messages. The routers periodically sen those messages but, a host can receive it more rapidly by sending a router advertisement solicitation. So when it will receive the wanted message, it just needs to add its interface identifier to the prefix obtained to get its global address. In the router advertisement, there are some useful information like the MTU of the link, the lifetime of an address with this prefix.

2.4 Deployment

The IPv6 protocol was defined in 1998 but nowadays it is still not fully deployed. On Figure 2.7 the evolution of deployment since January 2009 to nowadays is displayed. On May 26, 2018, 23.33% of the network has IPv6 connectivity. But as it can be seen, even 10 years after its creation the new protocol was practically not in use. Indeed, on December 31, 2008, only 0.06% of the network has IPv6 connectivity. It is only since 2014 that the usage of IPv6 has an appreciable increase.

Figure 2.8 exposed the IPv6 deployment per country. Surprisingly, some powerful and well developed countries like Russia and China have nearly not deployed IPv6. 1.69% for Russia and 3.48% for China. The country which has the most deployed the IPv6 is Belgium with 52.1%. And as it can be expected, third World countries have nearly or not deployed IPv6.

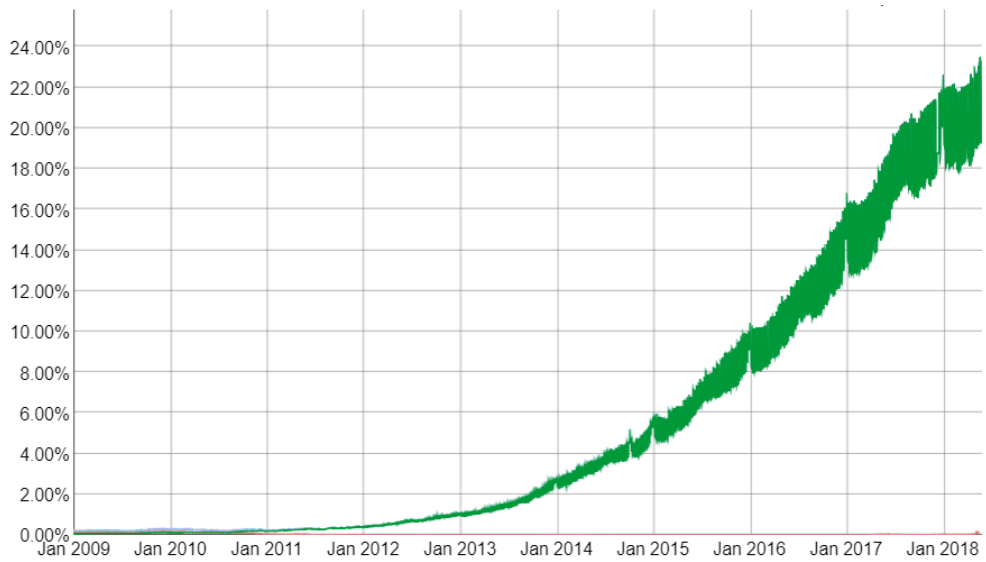


FIGURE 2.7: Evolution of the IPv6 deployment [6]

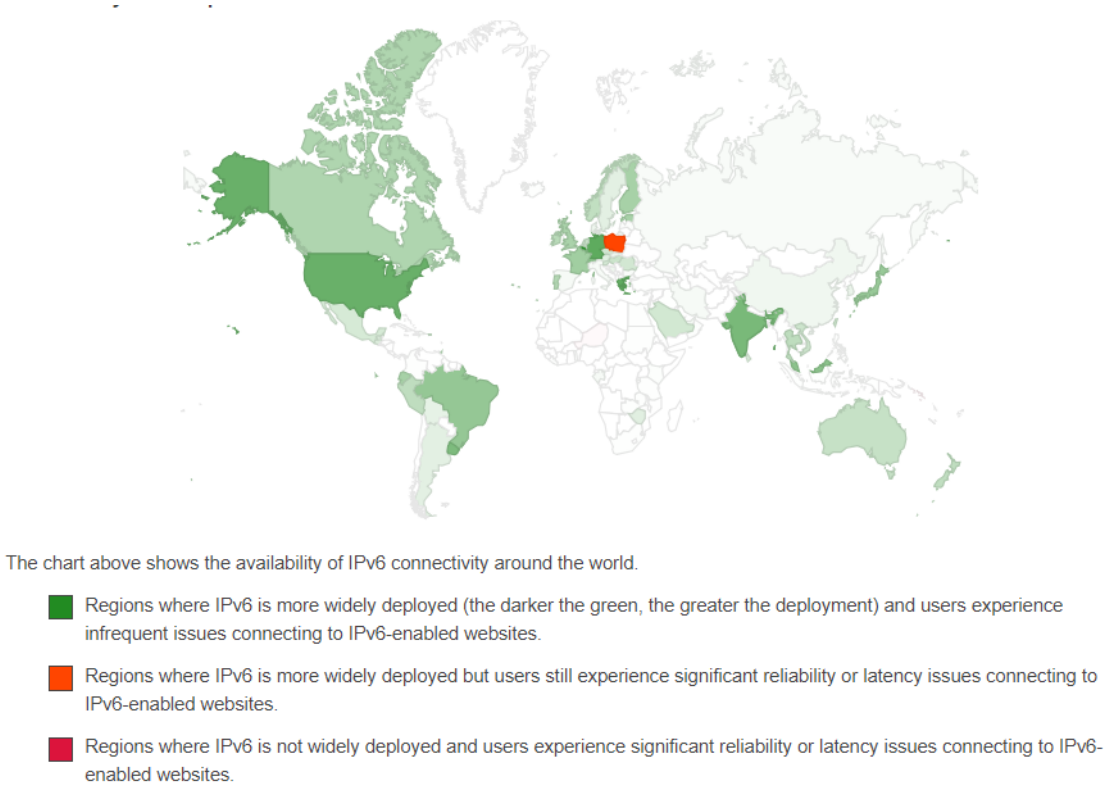


FIGURE 2.8: Deployment of IPv6 per country [6]

The main reason for this slow deployment of IPv6 is its cost due to material needs. In fact, all network devices must be changed to understand and correctly process this new version of the internet protocol.

Because it is impossible to fully replace the oldest version of the protocol by the new one in a single moment, several techniques have been developed to make the coexistence of the two versions of the protocol feasible. Indeed, if a host wants to connect to a specific server to retrieve some data and it uses the sixth version of the protocol to perform this exchange but in the path between the host and the server, there are several routers which only understand the fourth version of the protocol, something must be done otherwise those routers will just drop the packet.

One of those tricks uses the tunneling. To explain the trick in a more visible way, Figure 2.9 will be used to illustrate an example. So in this case, the user sends a packet with an IPv6 header to the server but only the routers R1 and R4 understand IPv6. Consequently, the router R1 knowing that the following router which understands the sixth version of IP is R4, will put the packet of the user as a payload of an IPv4 packet which has as destination address, the address of the left interface of R4. Like that, R2 and R3 which do not understand IPv6 will forward the packet to R4 which will unwrap the packet to retrieve the original packet and then will forward this packet to the server.

Of course, it is also possible in the other way so if there are routers which only understand IPv6 on the path. In this case, if a packet using IPv4 must go through those routers, the router which understands the two versions of the protocol will put the packet as the payload of an IPv6 packet which will be unwrapped later.

Thanks to this trick, the two versions of the protocol can coexist and one can use the desired version of the protocol since he can get the address of the destination host in the wanted version and its network administrator allows both versions of the protocol.

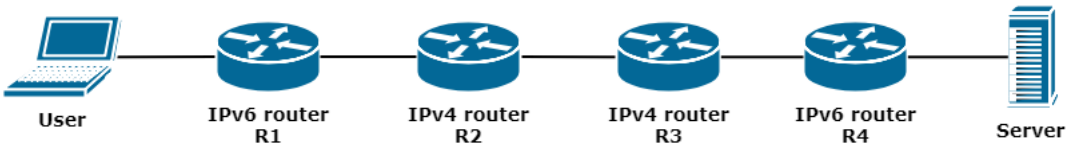


FIGURE 2.9: Tunnel example

Another trick maps an IPv4 address with an IPv6 address [7]. It exists two kinds of mapping. The first one is called the IPv4-Compatible IPv6 Addresses. It is used when the devices on the path can understand the IPv6. In this case, the IPv6 address will start with 96 bits set to 0 followed by the IPv4 address. For example, if the IPv4 address is 192.168.0.5, the IPv6 address will be 0::192.168.0.5

The second one is called the IPv4-Mapped Embedded IPv6 Addresses. It is used when the devices on the path do not understand IPv6. In this case, the IPv6 address will start with 80 bits set to 0 then 16 bits set to

1 and finally the IPv4 address. Taking the previous example, in this case the IPv6 address will be 0::FFFF:192.168.0.5.

They are not the only two tricks which allow the coexistence of IPv6 and Ipv4. For example there is also a method using MPLS to forward the IPv6 packet inside a network which only understands IPv4 [15].

Chapter 3

iOAM

In this chapter, the motivation behind In-band OAM will be developed first. Then its format will be displayed. During this master thesis, not all possible options of iOAM have been implemented so some of them will not be described with a lot of detail.

3.1 Motivation

As said in the introduction, the purpose of In-band OAM is to have a different way to perform the measurements. Indeed, thanks to it, one can get information given by an active measurement without injecting new traffic. But it cannot be considered as a passive measurement since an extension header is added in the user packet and so the size of the packet is bigger than before.

To see more clearly the purpose behind the development of this new method some comparison between iOAM and existing measurement tools will be made.

3.1.1 Ping vs iOAM

Before doing the comparison between this two methods, a quick reminder of ping will be made.

Ping is a tool used to know if a host is reachable and it allows to measure the round trip time taken by the packet to reach the destination and come back. It uses ICMP which is a protocol used by the network devices like hosts and routers to get some feedback when an error occurs. It allows some communication between devices. This protocol has a field for the type of the message, ping uses two of those possible types. The first message has a type value equal to 8. This type notifies the destination that the sender wants a response. This message is called an echo request. The second message has a type value equal to 0. It is used by the destination to notify the source that this message is the response of the echo request. This message is called the echo reply. The following Figure 3.1 represents this exchange.

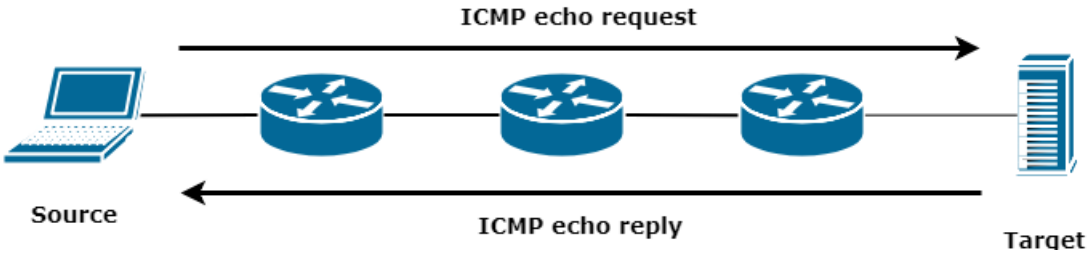


FIGURE 3.1: Ping exchange

And on the following Figure 3.2, there is an example of the result of a ping with IPv6. It can be seen the RTT taken by a packet sent through the interface `wlp3s0` to reach the host which has thee IPv6 address 2001:4860:4860::8888 and comes back to the source.

```
$ ping6 -c 3 -I wlp3s0 2001:4860:4860::8888
PING 2001:4860:4860::8888(2001:4860:4860::8888) from 2a02:2788:104:17ee:d05b:d4a3:1
a35:e33 wlp3s0: 56 data bytes
64 bytes from 2001:4860:4860::8888: icmp_seq=1 ttl=56 time=25.8 ms
64 bytes from 2001:4860:4860::8888: icmp_seq=2 ttl=56 time=24.5 ms
64 bytes from 2001:4860:4860::8888: icmp_seq=3 ttl=56 time=26.6 ms

--- 2001:4860:4860::8888 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 24.507/25.683/26.671/0.893 ms
```

FIGURE 3.2: Result of ping

Now that this reminder is done, the comparison can be made. For that, a point which is really important to understand, is that with ping it is the round trip time which is obtained. So it is the time taken by the packet to go from the source to the target plus the time taken by the response to go from the target to the source. It is a mistake to think that one can always divide by two the RTT to get the true time needed by a packet to go from the source to the target. Indeed, in the network, the path is not always symmetrical which means that the reply may not follow the same path than the request.

With iOAM, this problem disappears. Now it is possible to get the real time taken by a packet to go from a node to another. Indeed, there is an option in the iOAM trace option which allows getting the timestamp at which the packet is treated by the node so once just need to compute the difference between two timestamps to know this time. This option is described in the section 3.2. And it is also possible by setting a flags to notify to each hop that they must send back the iOAM trace to the source after they have modified it. On the following Figure 3.3 is displayed an iOAM trace in hexadecimal which contains the following information in the order of the list:

- the timestamp in second
- the timestamp in nanosecond

```
[IOAM] TRACE START
30  0 10  0
node size : 8
node 0:
5b  e 87 87
33 f6  4 ea
node 1:
5b  e 87 86
f2 3d ce a4
node 2:
5b  e 87 86
 8 57 1c f7
[IOAM] TRACE END
```

FIGURE 3.3: Result of an iOAM trace with the timestamp in seconds and nanoseconds

So as seen on Figure 3.3, there are three nodes and they are presented in the reverse order. The packet arrives first to the last node of the list, then to the second and finally to the first.

Moreover, since ping is mostly used to test the accessibility of and host, it is does not take into account the path followed by the packet. It is possible that the request does not always follow the same path because the packet cross some node on its path to reach the destination which performs some load balancing. So the value obtained must be treated carefully.

With iOAM, it is also possible to get the identifier of each node crossed by the packet and so know the path followed by the packet and the accessibility of those hosts. This node identifier must be unique in the scope of the network of the system administrator. So now if the packet can follow different paths to reach the same destination, it is possible to see it and also notice which node performs load balancing. On the following Figure 3.4 is displayed an iOAM trace in hexadecimal which collects the following information:

- the TTL value and the node identifier
- the timestamp in second
- the timestamp in nanosecond

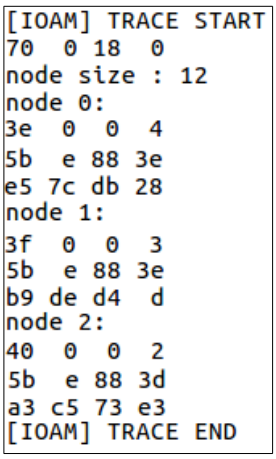


FIGURE 3.4: Result of an iOAM trace with TTL, node identifier and timestamp in second and nanoseconds

As previously said, the nodes are presented in the reverse order but here the first line gives the TTL value of the packet when it is ready to be forwarded by the nodes and its identifier. The first two characters are for the TTL and the following ones for the node identifier as described on Figure 3.10. So this packet has firstly encountered the node having as node identifier value 2. Then the node having as node identifier value 3 and finaaly, the node having as node identifier value 4.

When someone uses ping, he injects new packets in the network which have a length equals to 64 bytes in the case of IPv6. With iOAM it will add an Hop-By-Hop options header which has a size depending on the data asked and the number of hops in the trace. In the case where only the two timestamps are asked, this size will be equal to $2+4+4+8*n+p$ bytes where n is the number of hops in the trace and p the size of the padding to get a size which is a multiple of height. The first 2 bytes are for the next header field and the size field of the Hop-By-Hop options header, the second 4 ones for the type field and the length field of the option in TLV format plus the two reserved bytes, and the following bytes are described in the section 3.2 and on Figure 3.9. If there is also the option to get the node identifier, the size will be equal to $2+4+4+12*n+p$ bytes with n and p which are the same than previously. So in the first case, with less than 6 hops in the trace the size added is smaller than the size of a ping, for 6 the size is equal to the one of a ping and with more than 6 hops, the size needed is greater than the size of a ping. In the second case, it is smaller with less than 4 hops in the trace, equals if there are exactly 4 hops and greater if there are more than 4 hops. Of course, it is not mandatory to ask the two timestamp, it is totally possible to only ask the timestamp in nanosecond or in seconds. But if someone uses the timestamp in nanosecond, it must be aware that it is then totally possible to get a timestamp value in a node which is smaller than in the previous one since this timestamp has not enough space to record the full epoch time.

3.1.2 Traceroute vs iOAM

Before comparing this two methods, a quick reminder of traceroute will be made.

Traceroute is a tool used to know how many hops are crossed by the packet from the source to the target. It also allows knowing the RTT taken by the packet to reach each node on the path and for the response to go back to the sender. Moreover, it gives the needed information to infer the location of the nodes. Traceroute works in the following ways:

- At first the host sends a packet to the target but this packet has a TTL value equals to 1.
- The first hop on the path will receive the message and decrements the TTL value. Because this value will be equal to zero, it will send back a time to live exceeded ICMP message to the source of the packet.
- The source will receive the time to live exceed message and it will then redo those steps 2 more times to finally get three RTT values.
- The source will increment the TTL value by one and redo the three previous step.
- The i^{th} hop on the path will drop the packet because the packet has an initial TTL value equal to i. So when the packet will reach this specific node on the path, the current TTL value after the decrementing will be equal to 0.
- At some point, the TTL value will be large enough to reach the destination which will not discard the packet. But because traceroute uses expressly a UDP port which is not valid, the target will respond to the source with a port unreachable ICMP message.
- After that, the traceroute stops.
- Generally after 30 hops, if the destination is not reached, the traceroute also stops.

This process is represented on Figure 3.5.

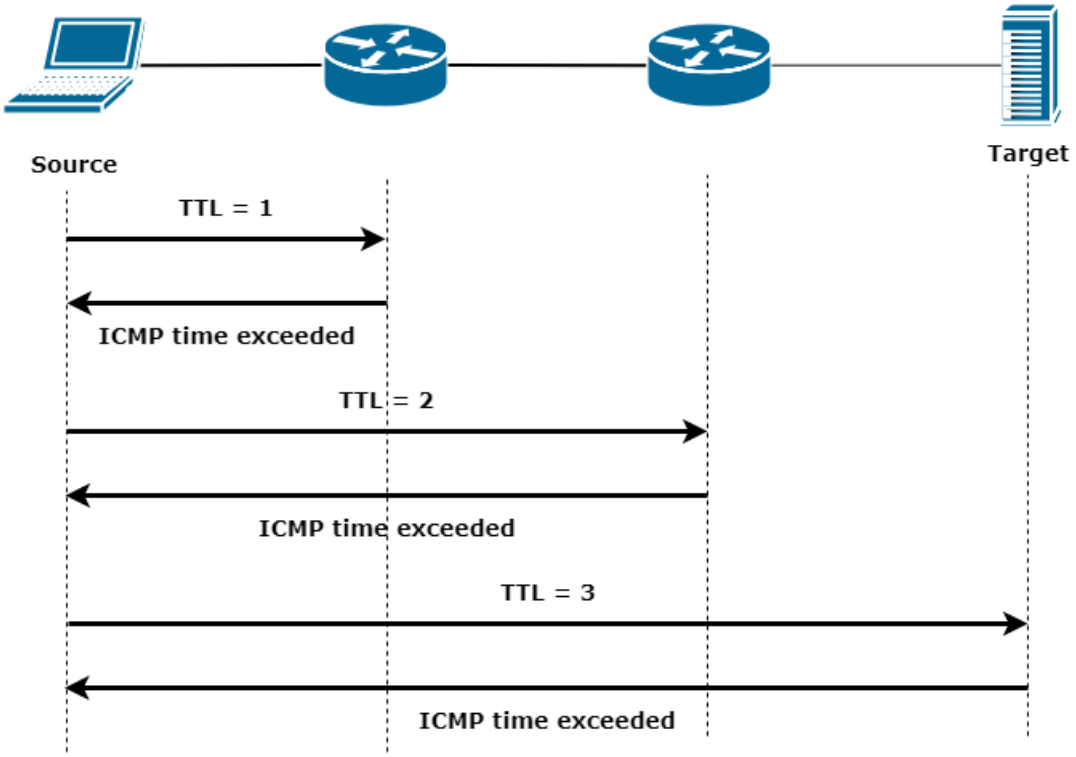


FIGURE 3.5: Traceroute process

On Figure 3.6 is displayed the result of a traceroute towards the device which has as IPv6 address 2001:4860:4860::8888. The ingress interface of each device can be seen with the RTT taken by the packet for each of it.

```
$ traceroute6 -n 2001:4860:4860::8888
traceroute to 2001:4860:4860::8888 (2001:4860:4860::8888) from 2a02:2788:104:17ee:d
05b:d4a3:1a35:e33, 30 hops max, 24 byte packets
 1  * * *
 2  2a02:2788:100::1  13.095 ms  13.579 ms  8.37 ms
 3  2a02:2788:100:3::1  8.699 ms  12.33 ms  9.211 ms
 4  2a02:2788:ffff:10::1  12.35 ms  16.939 ms  11.796 ms
 5  2001:7f8:26::a500:6939:1  11.122 ms  11.538 ms  13.141 ms
 6  2001:470:0:393::2  16.556 ms  25.01 ms  25.215 ms
 7  2001:7f8:4c::3b41:1  22.764 ms  23.571 ms  24.147 ms
 8  2001:4860:0:11e0::1  40.349 ms  35.565 ms  35.262 ms
 9  2001:4860:0:1::204d  21.409 ms  26.642 ms  22.96 ms
10  2001:4860:4860::8888  22.657 ms  21.92 ms  21.595 ms
```

FIGURE 3.6: Result of a traceroute

Here the tool uses UDP packets but it is also possible to send ICMP echo request with increasing TTL value. Like previously, the nodes will reply with time to live exceed ICMP message but when the packet will reach its destination, the source will not anymore receive an port unreachable ICMP message but instead an ICMP echo reply message.

Another version of this tool uses TCP packets. The port used in this case is the port 80 which is for HTTP. And in this case the source knows that it has reached the target because it will receive a SYN + ACK packet since it tries to start a connection with its destination.

With this tools, the same problems than with the ping occur since it is also the RTT which is measured. Moreover, the path given by the traceroute can be a path which does not exist in the network. On the following Figure 3.7, a small network is represented. This small network will be used to illustrate the problem announced previously.

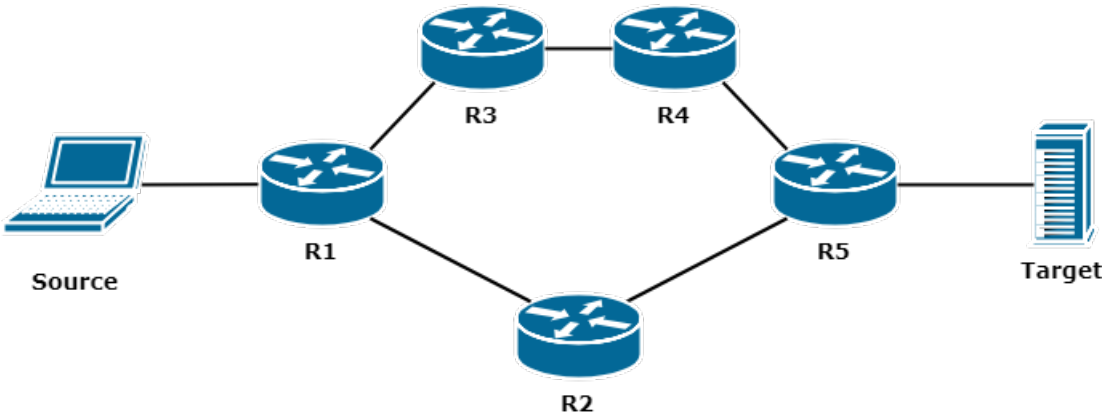


FIGURE 3.7: Example of multipath

So in this scenario, the source starts a traceroute towards the target. There is 2 possible paths to reach the target and in some cases it is conceivable that R1 can forward the packet towards R2 or R3 if the destination of

this packet is the address of the target. So traceroute can give different results since R1 will not always forward the packet having the same source and destination address and port, on the same egress interface. For example, it can give the following result which is not correct :

1. R1
2. R2
3. R4
4. Target

So with this result, one can think that there is a path between the source and the target which is composed of 3 hops where those hops are R1, R2 and R4 which is totally false. It is obvious when seeing the network 3.7 than the path source -> R1 -> R2 -> R4 -> target does not exist. But given that when the source sends a message to the target with a TTL of 2 it can reach R2 or R3, with a TTL of 3 it can reach R4 or R5, with a TTL of 4 it can reach R5 or the target and with a TTL of 5 or bigger it will always reach the target. So since all packets will not necessarily be forwarded in the same way by R1, it is normal to obtain a path which is not necessarily an existing one. Paris traceroute [9] which is an update of traceroute solves this issue.

With iOAM, this problem does not exist since there is no more need to send several packets with an incremental value of TTL. Indeed, by sending a message to the target with an iOAM trace asking for the node id and the timestamp, it is possible to get a real path between the source and the target and know the time to reach the target from the source. Therefore the number of injected packet decreases since to measure 3 times the time needed to reach each hop on the path, only 3 packets must be sent and not anymore 3 per hops present on the path. So there is an import amount of bytes saved. For example for a path with 10 hops, traceroute will send 30 IPv6 packets having a size of 80 bytes. With iOAM if the data asked are the node identifier and the timestamp in nanoseconds, the size added to 3 existing packets will be of 96 bytes. Consequently, 2112 bytes will be saved.

Also with traceroute, there is only the ingress interface of the router which is obtained. With iOAM, it is possible to obtain the node identifier, the ingress and egress interface identifier. In that case, if as previously there is 10 hops on the path and the asked data in iOAM are the two timestamps plus the node, ingress and egress interface identifier the amount of bytes saved would be equal to 1872. On the following Figure 3.8 is displayed an iOAM trace which collects the following information:

- the TTL value and the node identifier
- the ingress and egress interface identifier
- the timestamp in second
- the timestamp in nanosecond

```
[IOAM] TRACE START
f0  0 20  0
node size : 16
node 0:
3e  0  0  4
 0  1  0  2
5b  e 87 f0
b3 91 fb 4a
node 1:
3f  0  0  3
 0  1  0  2
5b  e 87 ef
87 f3 d0 8f
node 2:
40  0  0  2
 0  1  0  2
5b  e 87 ef
71 d9 82 62
[IOAM] TRACE END
```

FIGURE 3.8: Result of an iOAM trace with TTl, node and interface identifier plus timestamp in seconds and nanoseconds

On the previous Figure 3.8, the same information than the ones on Figure 3.4 are displayed plus the ingress and egress interface identifier. Here, the second line of each node in the trace notifies on which interface the packet has arrived and on which it will be forwarded. In this line, the two first column contains the identifier of the ingress and the two last the identifier of the egress as described on Figure 3.11. In this cases this line is the same for each node. It is normal since the interface identifier must be unique inside a node and they are numbered in the same way for each node.

There is another way to now the IPv6 address of a node crossed by the packet during its journey in the network. This is the Record Route Option. This option asks to each hop crossed by the pack to add their IPv6 address. But since it is the IPv6 address which is recorded, it consumes a lot of space. In iOAM, each node must be uniquely identified by a node identifier which can be placed in 24 bits. This node identifier will be collected with the TTL value and so the total place taken by those data has a size of 32 bits. So when the record route option carries one IPv6 address, iOAM can carry 4 node identifier. If also the ingress and egress interface identifier, iOAM can still carry the data collected on 2 nodes while with the record route option, only one IPv6 address is collected.

Of course here not all the possible values that can be obtained with iOAM are presented. There is much more information which can be carried in an iOAM trace and they will be described in the following section.

3.2 iOAM format

For the moment, the IETF has not decided a final version of the format for iOAM. Consequently, in this master thesis, the format given by the document [13] will be considered since it is the last at that moment.

There is three different type of iOAM options: the tracing option, the proof of transit option (POT) and the edge to edge option (E2E).

The tracing option is the one implemented in this master thesis. It is used to collect information about each node which are crossed by the packet.

The proof of transit option, as its name indicates, it is to be sure that a packet has really crossed some node. Indeed, the node on the path update some information present in the POT option and like that the final node can know if the packet has really crossed all the node that it normally should have.

The edge to edge option is used by the node which will insert the iOAM option in the packet to send some information to the node which will extract the option. It can be used to detect packet loss.

Because here it is the tracing option which is implemented, the two others formats will not be described.

For the tracing option, there is in fact two types of format. The first one is called pre-allocated trace option and the second one incremental trace option. The first one is more appropriate for the software approach and the second one for the hardware approach because as their name suggests, the pre-allocated one allocates directly the full space needed which is more convenient for the software approach since the allocation takes times. But so, the packet will carry unused bits until a node will fill this space with the asked information. And for the incremental one, each node will add some bits containing the asked information in the packet which is more convenient to the hardware approach this time. Like that the packet will not transport some unused bits.

Here the implementation is done in the Linux kernel so it is normal to choose the pre-allocated tracing options. Its format is described in the following picture :

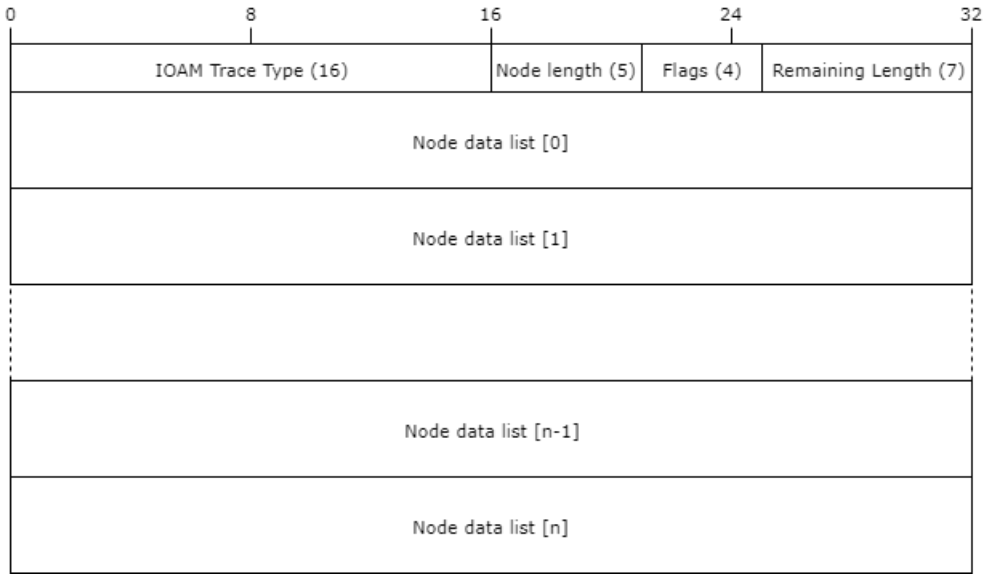


FIGURE 3.9: iOAM pre-allocated trace options

On this Figure 3.9, several fields are displayed. The first 4 octets contain all the information needed by a node to know where it must put its data in the node data list and which type of information it must record.

- IOAM trace type : it defines which information a node must add in the node data list.
- Node length : it is the sum of the data length added by a node in multiple of 4 octets without taking into account the opaque state snapshot field length (see later).
- Flags : it is a 4 bits long field but only the bits 0 and 1 are used at this moment.
  - The first bit which is the most significant one, is called the overflow bit and it is set to 1 by the node which has no more enough space to add the asked data.
  - The second bit is called the loopback one and if it is set to 1, each node crossed by the packet must send to the source a copy of the packet after having modified it.
- Remaining Length : it specifies the data space remaining in the node data list in a multiple of 4 octets.
- Node data list : it is the place where the nodes must add their data according to the iOAM trace type. The size of this part must be a multiple of 4 octets and is filled from the bottom to the top. All the information which can be put on this list are showed in the list below.

The system administrator can select which information he wants to collect thanks to the IOAM trace type field. Each bit of this 16 bits field indicates to the crossed nodes which information they must add.

The following list indicates which information is linked to which bit starting with the most significant one.

- Bit 0 : This bit indicates to the node that it must add the hop limit of the packet when it is ready for the transmission and its identifier. This identifier uniquely identifies a node in the iOAM domain and it is chosen by the system administrator. The following Figure shows the format.

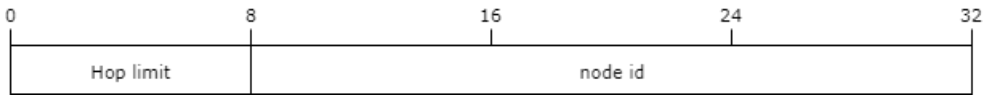


FIGURE 3.10: Hop limit and node identifier field format

- Bit 1 : This bit indicates to the node that it must add the ingress and egress interface id. Those identifiers are chosen by the system administrator. If for any reason the node can not retrieve the identifier of an interface, the field of this interface must be filled with the value 0xFFFF. The following Figure shows the format.

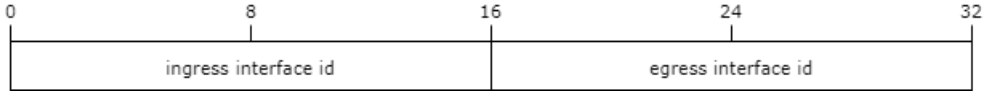


FIGURE 3.11: Interface identifier field format

- Bit 2 : This bit indicates to the node that it must add the absolute time in second at which the packet has arrived to it. If the node is not able to get the timestamp, the field must be filled with the value 0xFFFFFFFF. The following Figure shows the format.

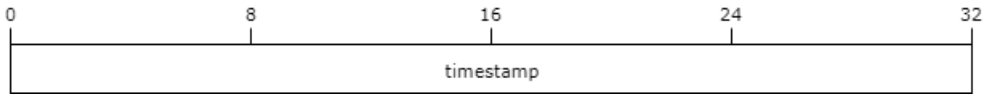


FIGURE 3.12: Timestamp field format

- Bit 3 : This bit indicates to the node that it must add the absolute time in nanosecond at which the packet has arrived to it. If the node is not able to get the timestamp, the field must be filled with the value 0xFFFFFFFF. It has the same format than the previous one, see Figure 3.12

- Bit 4 : This bit indicates to the node that it must add the time spent by the packet in the node in nanosecond. If this time is greater than $2^31 - 1$ nanoseconds, the top bit, where there is a '0' on Figure 3.13, is set to 1 to indicate an overflow and the value is set to 0x80000000. If the node is not able to retrieve the data, the field must be filled with the value 0xFFFFFFFF. The following Figure shows the format.

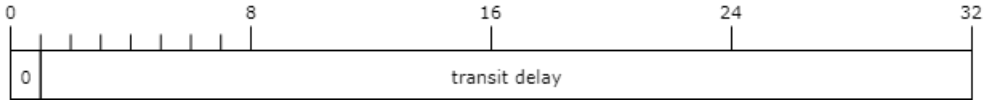


FIGURE 3.13: Transit delay field format

- Bit 5 : This bit indicates to the node that it must add some specific application data which can not be bigger than 4 octets. The following Figure shows the format.

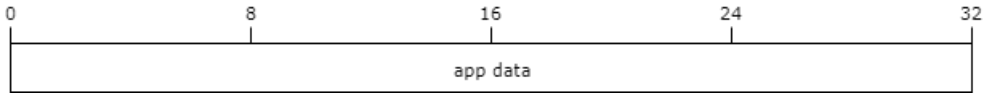


FIGURE 3.14: Application data field format

- Bit 6 : This bit indicates to the node that it must add the length of the egress interface's queue when the packet is placed in it. The length must be the number of occupied memory buffers not the length in octets. The following Figure shows the format.

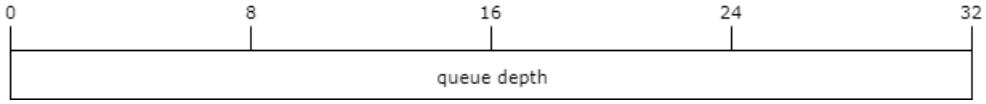


FIGURE 3.15: Queue depth field format

- Bit 7 : This bit indicates to the node that it must add some data according to the format given by the schema id field which is defined by the user. It is a TLV format which is shown on the following figure.

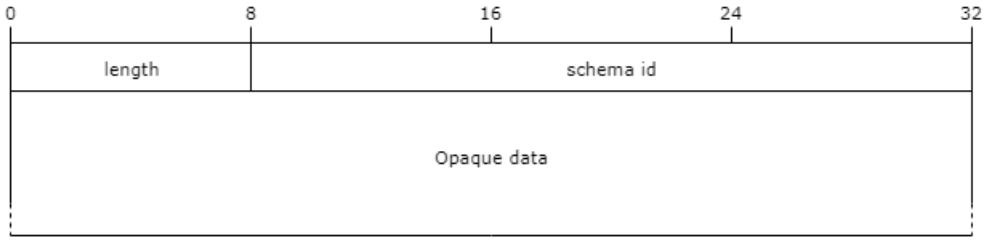


FIGURE 3.16: Opaque snapshot field format

- Bit 8 : This bit indicates to the node that it must add the hop limit and node identifier but in its large format this time. The following Figure shows the format.

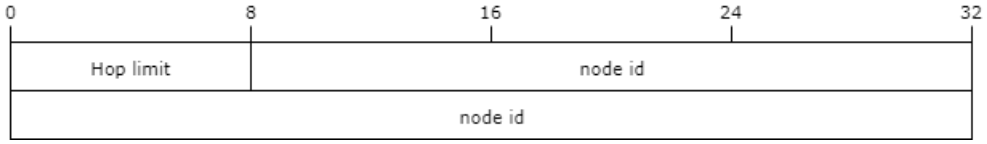


FIGURE 3.17: Hop limit and wide node identifier field format

- Bit 9 : This bit indicates to the node that it must add the ingress and egress interface id in their large format. The following Figure shows the format.

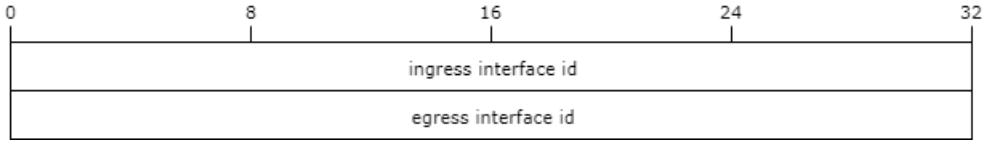


FIGURE 3.18: Wide interface identifier field format

- Bit 10 : This bit indicates to the node that it must add some application data but this time there is a place of 8 octets for it. The following Figure shows the format.

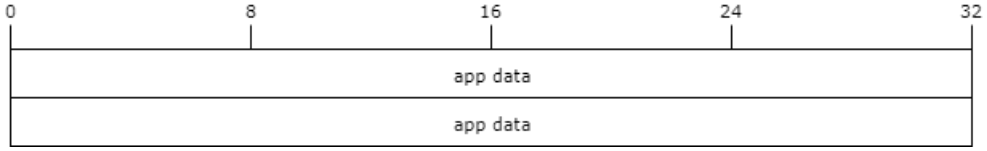


FIGURE 3.19: App data wide field format

- Bit 11 : This bit indicates to the node that it must add in the first 2 octets a checksum complement. The following Figure shows the format.

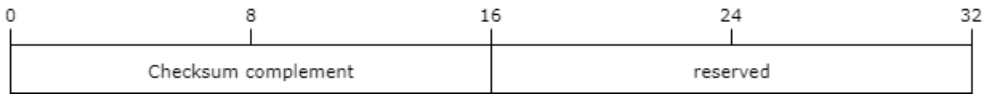


FIGURE 3.20: Checksum field format

- Bit 12-15 : Not used at this moment.

The following Figure shows an example of a node data. The value of the iOAM trace type for this node data is 0x9A40.

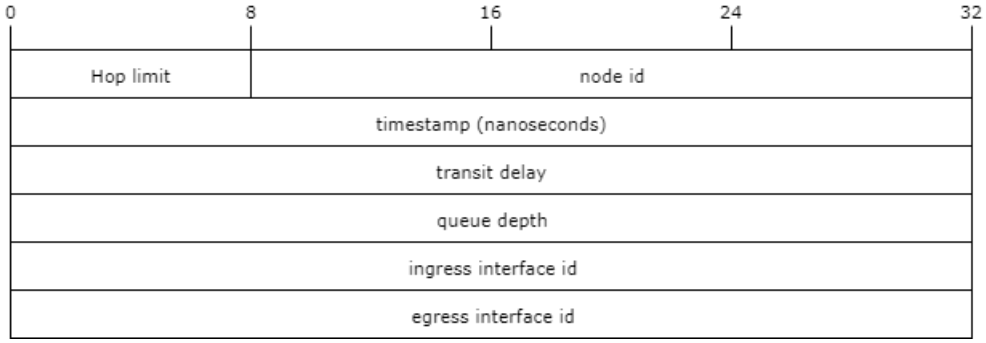


FIGURE 3.21: Node data format for iOAM trace type of 0x9A40

As seen on Figure 3.21, the data appears in the same order than in the list before.

Chapter 4

Implementation

In this chapter, the implementation of iOAM done in the Linux kernel in the context of this master thesis will be described. The code of this implementation can be retrieved on GitHub [1] This implementation will be presented in four different section. The first section will expose how the Hop-By-Hop options header containing the iOAM pre-allocated trace is created. Then the second section will show how this extension header is inserted in the existing packet going through the node. After that comes the third section which will describ how the iOAM trace is removed from an IPv6 packet. And finally, the last section will present how the asked data are recorded in the trace.

4.1 Header creation

Before being able to insert the extension header in a packet, it must be created and placed in a structure. For this purpose, a new ioctl has been created. There are several ways to set some value and do things with them in the Linux Kernel.

The first one is the `sysctl` which allows to read or write some value present in the `\proc\sys` directory. Only few types of value can be set with this method.

The second one is the system call. It is an interface which allows one to execute some specific functions with the rights of the kernel.

The third one is the ioctl system call. It is the abbreviation of input/output control. It is a specific system call used to control a device.

For the creation of an Hop-By-Hop options containing the iOAM pre-allocated trace, the `sysctl` offers too few possibilities. A system call offers enough flexibility but it is architecture dependent so, it means that it must be implemented for each possible architecture supported by the Linux kernel.

Consequently, the ioctl system call which is not architecture dependent and offers a lot of possibility in terms of exchange between the user space and the kernel space has been chosen.

One of this possibility is the read function of the ioctl. It could be implemented in a future work, to place the retrieved data coming from the trace in a ring buffer. So one could read the data placed on it thanks to this method. Or also the mmap function can be used to map some kernel space

memory with some user space memory. Which should be faster.
Figure 4.1 depicts the structure taken as argument by the ioctl.

```
struct ioam_register{
    int mode;
    int hop_nb;
    unsigned int freq;
    uint8_t pot_type;
    uint8_t e2e_type;
    uint16_t ioam_trace_type;
    uint32_t schemaID;
    int seg_nb;
    struct in6_addr *seg;
    char ifname[IFNAMSIZ];
};
```

FIGURE 4.1: Ioctl argument structure

This structure has several fields and they are described in the following list :

- mode : its value is between 0 and 7. 0 means no iOAM and so free the header if there is an existing one. 1 means there is a pre-allocated trace options, 2 there is the proof of transit option and 4 there is the edge to edge option. So for example a mode equals to 3 means that the Hop-By-Hop options header will contain an iOAM pre-allocated trace option and a proof of transit option.
- hop_nb : is the number of nodes in the node data list. This value must be chosen carefully. If this value is too big, some bits in the trace will never be used so it will consume some resources in the network for nothing. But if it is too small, some node will not be able to add their information in the trace.
- freq : the rate at which the extension header will be inserted in a packet. For example, 0 or 1 means every packet and 100 means 1 packet over 100.
- pot_type : the type of the proof of transit option.
- e2e_type : the type of the edge to edge option.
- ioam6_trace_type : the iOAM trace type of the pre-allocated trace option like seen in the section 3.2
- schemaID : is the id of the schema for the opaque state snapshot field.
- seg_nb : the number of segment in the IPv6 segment routing.
- segments : the addresses for the IPv6 segment routing.

- ifname : the name of the interface which will insert the Hop-By-Hop options header.

As seen in the structure 4.1, the Hop-By-Hop options header will be created per interface because it is logical to think that according to the incoming interface, the packet will not cross the same number of hops in the network or the number of packets going through this interface will be higher or smaller than on another one and so the rate of the header insertion will be different. It is also possible to add in a future work some useful filter. One of this filter can be applied to the destination address, so only packets which will be forwarded to this specific destination will be modified to carry the Hop-By-Hop options header.

When all the parameters have been checked and the extension header was correctly created, it is stored in a structure which has been added to the `net_device` structure. Like that it is easily accessible and it does not need to be created each time a packet arrives and full fill the condition to receive the Hop-By-Hop options header. Which speed up the process.

In this structure, there are several fields which are already present but not implemented in this work. As previously said, the proof of transit option and the edge to edge option are not implemented in this work but like that, once just need to change some specific part of the header creation to correctly add them.

4.2 Insertion of the Hop-By-Hop options

Firstly there is need to find where is the right place to insert the Hop-By-Hop options header in the incoming packet if it full fills some conditions.

After some search, it has appeared that the right place to add it was in the `ipv6_rcv` function in the Linux kernel. Like that it is inserted in the packet just before the check of the presence of an Hop-By-Hop options header in the packet. Consequently the hop which inserts the packet will also adds its information in the iOAM trace. This is represented on Figure 4.2a.

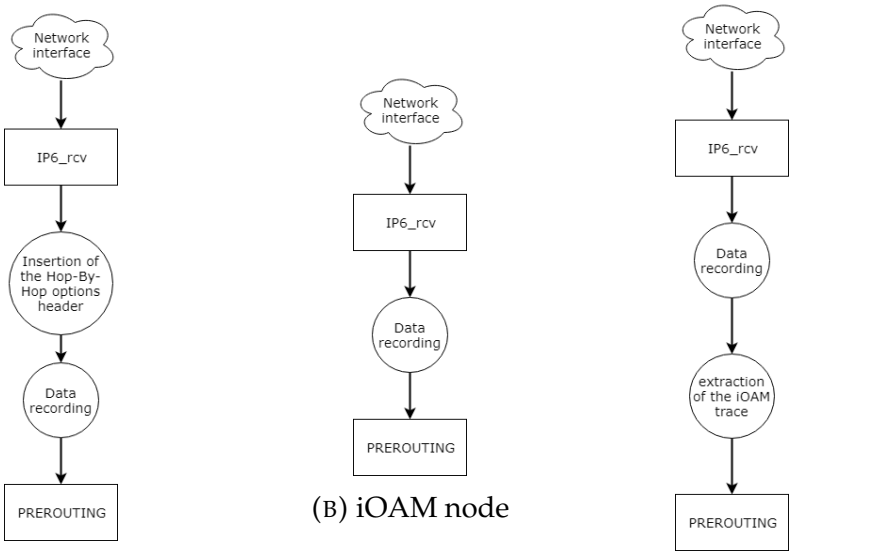
The Hop-By-Hop options header is not always inserted, some specific criterion must be full filled :

1. The ingress interface must be in iOAM mode. To put an interface in iOAM mode, once just need to use the ioctl with a value for bigger mode than 0 and a correct value for the frequency, the hop number, the iOAM trace and the interface name.

2. It must be the i^{th} packet going through this interface. This value depends on the frequency of insertion decided.

3. There must be enough remaining space in the packet to add the created Hop-By-Hop options header containing the pre-allocated iOAM trace.

4. The packet must not already carry an Hop-By-Hop options header.



(A) iOAM ingress node (B) iOAM node (C) iOAM egress node

FIGURE 4.2: Codepath per type of iOAM node

The last condition is an implementation choice. There are only three cases for a packet to have an Hop-By-Hop option header in the Linux kernel, otherwise it will be discarded.

The first one is when the packet has a jumbo frame. A packet which has a bigger size than the maximum normal size of an IP packet. But in the network most of packets are small so it does not appear often. The second one is for router alert option [19] which must be forwarded as fast as possible and the last one for common architecture label IPv6 Security Option (CALIPSO) [17].

Also because it is important to be fast in this case. If the iOAM pre-allocated trace option is inserted inside an already existing Hop-By-Hop options header, it is necessary to look at the size of the existing options. If this size plus the size of the trace is divisible by 8, the trace can be added at the end. But if it is not the case it is a mistake to think that adding padding to have a total size divisible by 8 would always be the right thing to do. Indeed, it is possible that there is already some padding option in this Hop-By-Hop options header and so if the size of the padding option already in place plus the size of the padding option required to be a multiple of 8 is greater than 7, the padding option needed can not be added otherwise the kernel will drop the packet. So the options must be parsed, the position and size of padding option recorded and if this size plus the one needed is smaller or equal to 7, the trace option must be added at the end with the padding option needed. But in the other case, a number of padding option must be removed and so a part of the Hop-By-Hop options header must be shifted which will take times.

Of course, the next header field of the IPv6 header will be copied in the next header field of the Hop-By-Hop options header and then be changed to 0 because as seen in the Table 2.1, it is the value for the Hop-By-Hop options header.

Another value must be changed and it is the payload length field of the IPv6 header. But it is not correct to just add the value of the Hop-By-Hop options header to it. Indeed, the value of this field is encoded in a particular way. To read the value, the function `ntohs` must be used. It will convert a value which is in the network byte order to a value in the host byte order. Then add to the obtained value the length of the Hop-By-Hop options header and use the `htons` function to transform it to a value in the network byte order.

Because the process of packets must be fast to not make some congestion on the ingress queue of the interface, the insertion of the extension header must be fast. So the searching of the method to allocate the space at the beginning of the packet if there is not enough space on it has been made carefully. It appears that there is two possible functions which allows adding some space if needed at the beginning of the packet.

The first one is `pskb_expand_head`. It expands the space at the head or the tail of the structure which is used to represent a packet in the Linux kernel.

The second one is `skb_cow_head`. It expands the space only at the head and it is used when someone wants to push some data at the beginning of the packet and do not touch the data of the packet.

It appeared that it is the last one which is less time consuming and so for this reason, it is the one used to allocate space at the beginning of the packet if there is not enough remaining free space.

4.3 Pulling out the Hop-By-Hop options

Firstly there must be a way to distinguish the nodes which must pull out the Hop-By-Hop options header and the others. But must be a node or some interface of it which must pull out this extension header ?

Taking for example a node which is at the border of the network of an administrator. This node will forward some packet in the network of the administrator and some other to the other network. So if the administrator wants to measure some metrics in its network, it is normal to think that this node will add the Hop-By-Hop options header containing the iOAM pre-allocated trace for packet which will go in its network. But he does not want that those packets containing the iOAM trace go to the network of other administrator so this node will pull out the header and consequently, this node will never collect information apart from its own if it this the responsibility of a node to extract the trace. So, it is normal to put this responsibility to an interface and not to a node.

But there is still need to distinguish an interface which will just add the information and another one which will add the information and then pull out the iOAM pre-allocated trace.

A new `sysctl` has been added for this purpose. Its default value is 0 like that unless the administrator specified it, no interface will pull out the trace. If the administrator wants to put a specific interface in extract mode,

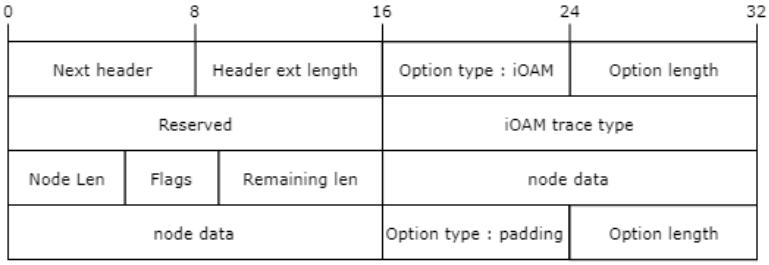


FIGURE 4.3: First extraction case

he just needs to use the following command :
`sudo sysctl -w net.ipv6.conf.ens3.ioam6_if_decap=1.`

Now that it is done, the suitable place to extract the trace must be found. Some information very essential to correctly extract it must be collected. The needed information are the place of the iOAM pre-allocated trace in the packet and the size of the padding. Those information are useful and the reason of that is explained below. It appears that those pieces of information are already calculated in the function which parses the TLV options present in the Hop-By-Hop options header. Consequently it is the a right place to extract the extension header. Just after the parsing of the options, if the iOAM pre-allocated trace is present in the packet and the interface is in removing mode, the function `ioam6_remove` will be called to extract the trace. This is represented on Figure 4.2c.

So the distinction between an interface in extract mode and the others has been made and the place where to remove the trace has been found consequently now the way in which the trace is removed will be exposed. Contrary to the insertion, the possibility that there is not only the iOAM pre-allocated trace option in the Hop-By-Hop options header is taking into account. Since some other work could possibly add an option in an existing Hop-By-Hop options header, it is normal to take that into account. Therefore there are three possible cases:

- The first case is the most simple one. Indeed if there is only the iOAM pre-allocated trace option with the needed padding option to have a size which is a multiple of eight. It is just necessary to remove the Hop-By-Hop options header and change the size present in the IPv6 header as well as its next header field. This case is represented on Figure 4.3.

- The second case is when there are others options in the extension header but after subtracting the size of the pre-allocated trace option, the size of this extension header remains a multiple of eight. In that case, there is only need to remove this option, and shift the other option if needed, so if the iOAM trace is not at the end of the Hop-By-Hop options header. After that, the size field of this extension header and the one of the IPv6 header is changed to reflect the modification made. In that case like for the following one, the next header field in the IPv6 header is not modified because the Hop-by-Hop options header is still present in the packet. This case is represented on Figure 4.4.

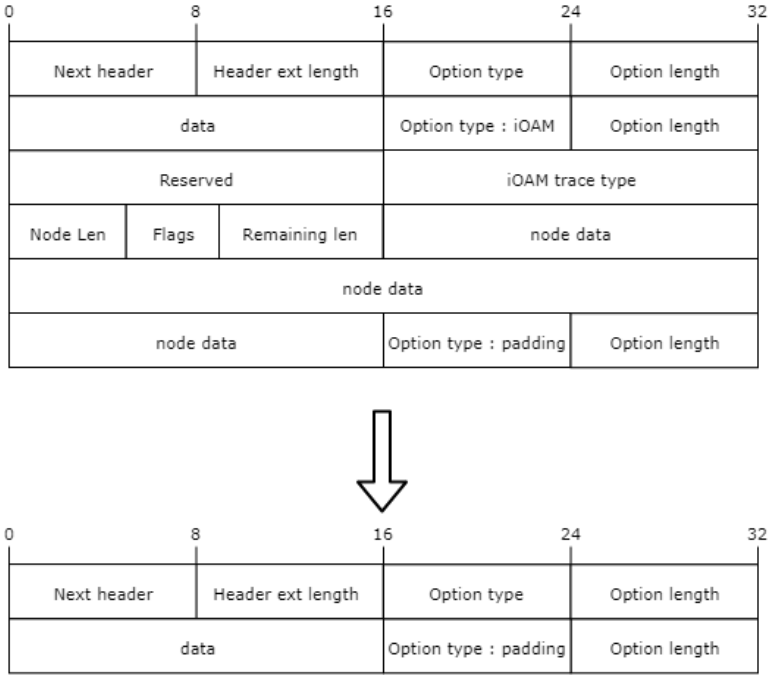


FIGURE 4.4: Second extraction case

- The last case is the most complicated one. It is when the size of the extension header after subtracting the size of the iOAM pre-allocated trace option is not a multiple of eight. In that particular case, the Hop-By-Hop options header must be parsed to find the position of padding option if any and their size. If there is no padding option, there is just need to add one with the needed size at the end. If there is already some padding options, and if their size plus the size needed to be a multiple of eight is smaller or equal to seven, the needed padding option can be added at the end. But if this is not the case the previous padding options must be removed and a new one with the right size must be added at the end. This case is represented on Figure 4.5. Of course, a shift is performed on all the options present below the removed the padding options and the pre-allocated trace option. After that, the size field of the IPv6 header and the Hop-By-Hop options header must also be changed to respect the modification.

At first it was correctly working during tests. Those tests consisting in sending some ping to a destination. The first hop encounters by the packet inserted the trace and added information, the second hop added information too and extracted the trace since its ingress interface was in extract mode and the third hop was the destination of the packet. In this scenario, the asked information was correctly collected and the packet was correctly delivered to the destination since the echo reply was received.

But then another test has been made. In this new test, the destination of the packet has its ingress interface in extract mode. In that scenario, the packet arrived at the destination, the asked information was added and the trace was removed but no echo reply was received.

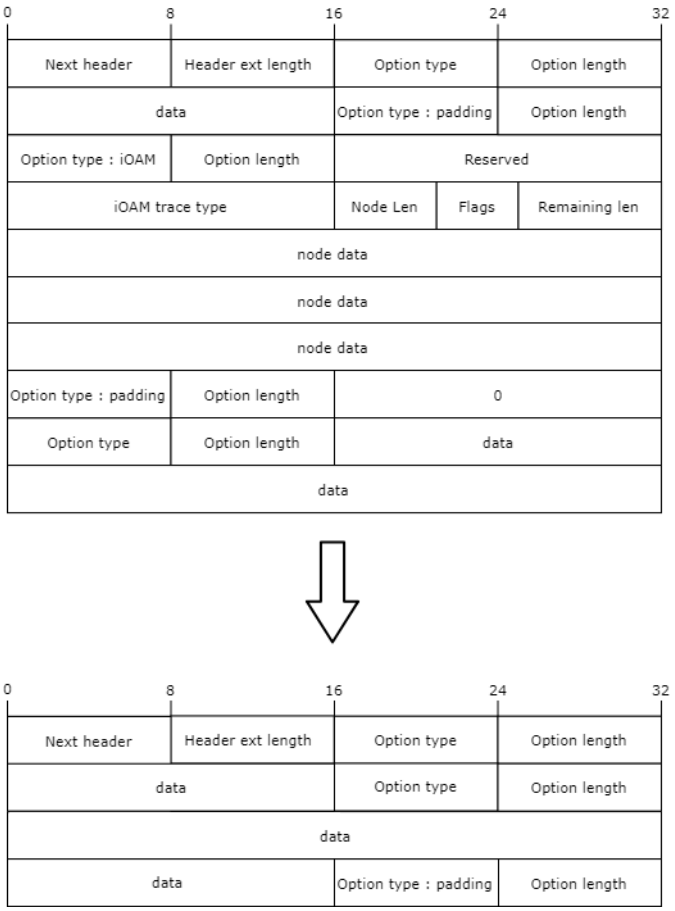


FIGURE 4.5: Third extraction case

After some search, it appeared that the problem came from a flag which was set if there is an Hop-By-Hop options header in the packet. This flag was checked at some point in the delivery process but since in this particular scenario, the Hop-By-Hop options header only contained the iOAM pre-allocated trace and the padding if they are needed, when the interface removed the trace it in fact removed all the extension header. So seeing that the value of this flag was set to true but there was no trace of an Hop-By-Hop options header in the packet, it was dropped.

To solve this issue, after removing the Hop-By-Hop options header, the flag in question is reset to false.

The information collected by the trace can be retrieved in the kernel log and they are displayed as seen on Figure 3.3. The nodes are displayed as they are present in the trace so in the reverse order. So, the node 0 is the last node which has added the information.

4.4 Adding information

So at this point, the iOAM pre-allocated trace is correctly inserted and extracted. But the information must be correctly added to the node data list field of the trace.

When an IPv6 packet arrives at a node which runs on the Linux kernel after going through the function which will handle it by the network driver, it arrives at the to the `ipv6_rcv` function. It is in this function that the check for the presence of an Hop-By-Hop options header is made. If it is present, the function `ipv6_parse_hopopts` is called which will call in turn the function `ip6_parse_tlv` with the structure `tlvprochopopt_lst` which contains the available functions to parse the different TLV according to their type. Consequently a new function was created and added to this structure to handle the TLV which has the type of iOAM pre-allocate trace. The interesting part of the path followed by a packet in the Linux kernel on a node is represented on Figure 4.2b.

Because the IETF has not already decided a final version for the format of the iOAM, there is not at this moment a number assigned by the IANA to the iOAM pre-allocate trace.

When a packet with an iOAM trace arrives at a node running on the Linux kernel, some check will be made by the function which will parse the TLV with the type of the iOAM pre-allocated trace.

The first check is to verify that the overflow bit is not set because if it is the case the parse stop directly since there is no more place in the node data list to record the data of the current node.

The following checks are made to be sure that there is no error in it since there is not anymore a checksum in the IPv6 header.

- The first check verifies that the Node Length field after being multiplied by 4 is equal to the size of a node calculated from the iOAM trace type field.

- The second check verifies that the Node Length field divides the Remaining Length field. And so that the value in the first field is not bigger than the second one.

In the following subsections, the way in which the possible data are added by a node in the trace is described.

4.4.1 Hop limit and node identifier

The first possible data which can be recorded by a node is the hop limit of the packet when it is ready to be forwarded by the node and its node identifier which must be unique in the iOAM domain. This node identifier is chosen by the system administrator.

For that a new `sysctl` has been added to make it easy for the system administrator to modify it and also to be easily retrievable to be recorded in the node data list of the trace. There is two possible types of integer value that can be passed through a `sysctl`.

The first one is an `int`. The problem with the `int`, is that it has not the same size on all architecture. Indeed on some architecture it has a size of 32 bits which is sufficient to record the node identifier but on some architecture, it has a size of 16 bits which is not enough.

The second one is an `unsigned long`. This data type has a minimum size of 32 bits which is sufficient for the need. So, for this reason the node identifier is recorded as an unsigned long. If the system administrator wants to set the identifier value of the node, he just needs to enter the following command : `sudo sysctl net.ipv6.ioam6_node_id=ID` where the ID is the desired value. As said, the node identifier is represented as an unsigned long. But only the leftmost 24 bits of this number will be placed in the trace as seen on Figure 3.10. Consequently if the system administrator tries to assign a value bigger than $2^{24} - 1$ to the node identifier through this `sysctl`, this value will not be taking into account and the old one remains. By default this value is equal to one.

The hop limit value is easily retrieved in the IPv6 header and recorded in the trace after being decremented by one because at this moment, the TTL value has not already been decremented by the node.

It is also possible to have those data in the wide format as seen on Figure 3.17. In this case, the field of the node identifier has a size of 56 bits. So there is a need of an other `sysctl` since the size of the `unsigned long` is 32 bits. Consequently another one has been created but this one have no limit in its range of value. The system administrator can modify the value thanks to the following command : `sudo sysctl net.ipv6.ioam6_node_id_cond=ID` where the ID is the desired value. This value will be recorded in the second node id field present on Figure 3.17 and has a default value of 0.

4.4.2 Ingress and egress interface identifier

The second possible data which can be recorded is the ingress and egress interface identifier. It must be at least unique at the scope of the node

to be able to distinct them. So like for the node identifier, the system administrator must have a way to modify those values. The same solution than for the node identifier is used. A `sysctl` is created for that and a system administrator just need to write the following command to set the identifier value of a specific interface `sudo sysctl net.ipv6.conf.ens3.ioam6_if_id=ID` where ID is the wanted identifier value for the specified interface which is ens3 in this case.

Usually, the egress interface is not known at this stage of the process of the packet in the Linux kernel. But to be sure of that, a test is made to know if the packet has a valid destination or not. If it is not the case, the lookup in the forwarding table is made and if there is a possible egress interface, it is set.

Thanks to that, it is possible to get the identifier of the two interfaces but if for one reason, one of them is not found, the field of the identifier value for this interface is set to 0xFFFF. Otherwise, the 16 leftmost bits of the integer value given by the `sysctl` are placed on the corresponding field present on Figure 3.11.

If its the wide format which is asked, in this case the 32 bits of the integer value given by the `sysctl` are used and placed in the right field present on Figure 3.18. In this case, if it is not possible to find the identifier value of an interface for some reasons, the field of this specific interface on Figure 3.11 is filled with 0xFFFFFFFF

4.4.3 Timestamp

The third possible data is the timestamp. There are two formats for it, the first one is the timestamp in second and the second one is in nanosecond. The timestamp must be in the Unix format so the time spent since the January first, 1970 UTC.

There is a function in the Linux kernel which gives this timestamp in nanosecond and it is : `ktime_get_tai_ns`.

So if the iOAM trace type specified the need of one of them, this function is called to get the timestamp. If it is in nanosecond, the leftmost 32 bits of this value are recorded. Otherwise this value is divided to get the timestamp in the second and this value is recorded. Since the unix timestamp in seconds is encoded on 32 bits, on January 19, 2038 at 3 hours 14 minutes 7 seconds UTC there will be a bug and the value read will be December 13, 1901.

Of course, the computation of the timestamp is an expensive process. In the network, the processing of the packets must be fast to not create some congestion problem. Consequently, it is obvious that if the two timestamps are asked so the timestamp in second and in nanosecond, the timestamp is in fact only calculated once.

4.4.4 Transit delay

The fourth possible data is the transit delay. It is the time spent by the packet in the node since the first bit has arrived until the last one has been sent. Of course, it is impossible to get the right value since when the packet is placed in the queue of the egress interface, it is no more modifiable unless each driver is modified to put the right value in it. But even before there are already several problems.

Indeed, there is a place in the structure which represents a packet in the Linux kernel which should be used to set the timestamp at which the packet has arrived. But this value is in fact never set and so always equals to 0. To change that, the code of each network driver should be changed to now set this value. But since the computation of the timestamp consumes a lot of cpu time, it is not necessarily a good idea because it will slow down the full process which handles a packet and so possibly create some congestion.

But maybe, a compromise can be found. Some accuracy can be lost for the rapidity of the packet processing. The value put in this timestamp can be placed in a structure which will be updated at a particular frequency for example. Knowing that, the person asking for the transit delay, can infer an interval of time in which is present the true time spent by the packet in this node.

There is also another problem which is to estimate the time the packet will spend in the queue of the egress interface. It will be exposed later since this issue is coming from the one in the subsection 4.4.6.

So at this moment, unless the timestamp presents in the structure representing a packet is not equal to zero, this field is filled with the value 0xFFFFFFFF. If the timestamp is not set to zero, it is this value minus the current timestamp. Of course if the current timestamp is already computed because the timestamp in second or in nanosecond is asked, it is not recalculated for the sake of performances. Moreover, the time spent between the moment when the timestamp is calculated and the moment when the transit delay is computed, is very small.

4.4.5 Application data

The fifth possible data is the application data. It is a free format where a 32 bits value set by an application can be placed.

So another `sysctl` has been created. The value of the application data can be modified through the terminal thanks to the following command : `sudo sysctl -w net.ipv6.ioam6_app_data=VALUE` where the value is the value of the application. It can also be modified by a program thanks to the `sysctl` API.

This value is per node and not per interface since the data come from an application which is running on the node.

It is also possible to have the application data in its wide format and so the value recorded can have a size of 64 bits. So another `sysctl` which is also per node, has been created. The value which will be recorded in the second part of the application data field present on Figure 3.19 can be modified

through the terminal by means of the following command : `sudo sysctl -w net.ipv6.ioam6_app_data_cond=VALUE` where the value is the value of the application. Of course it can also be modified by a program thanks to the `sysctl` API.

4.4.6 Queue depth

The sixth possible data is the queue depth. This data must record the number of occupied buffers in the queue of the interface on which the packet will be forwarded.

The problem to have this information comes from the place where it is available. Indeed, this information is stored in a structure present in the network drivers. After a long search in the Linux kernel to find a generic function which is capable of giving this information, no one was found. There is some function like the `get_ringparam` one coming from `ethtool`. But this function only gives the current parameter of the ring, not the current number of buffers used in the queue of the interface.

Of course, a function could be implemented to get the wanted value, but for that, this function should have a generic interface which should be implemented by each network driver since the structure having the desired information is driver dependent.

4.4.7 Opaque State Snapshot

The seventh possible data is the opaque state snapshot. It is a TLV encoded data. According to the schema id field present on Figure 3.16, one knows which data must be put in the data field.

For the moment, there is no support to add a schema and there is no schema already present in this implementation. There is a need to find an easy and suitable way to be able to add new schemata and correctly parse their TLV if present in the trace to correctly identify which data are asked, where to find it and how to place it in the data field of the opaque state snapshot.

4.4.8 Checksum Complement

This is the eighth and last possible data which can be asked in the iOAM trace for the moment. This value like the previous one has not been implemented in this master thesis since it has been notified during the test of the performances of the implementation. And it requires a certain amount of time to find the suitable function to recompute the checksum and test it in the appropriate scenario.

Chapter 5

Performances

In this chapter, the performance of the implementation made during the master thesis will be assessed. This chapter is broken down into several sections. The first section will present the methodology used. The second section will be composed of a presentation and a discussion about the performance obtained.

5.1 Methodology

Now that an implementation has been made in the Linux kernel, it is interesting to see how it affects the packet processing of a node when it uses iOAM. For that five hosts running on a virtual machine with the version 4.12 of the Linux kernel have been put in place. Those machines use the qemu kvm emulator. Each host has only access to one cpu core whose model is Intel Xeon E5-2620 v4 2.1Ghz and each machine has 1GB of RAM.
The following Figure shows the network topology used for the test :

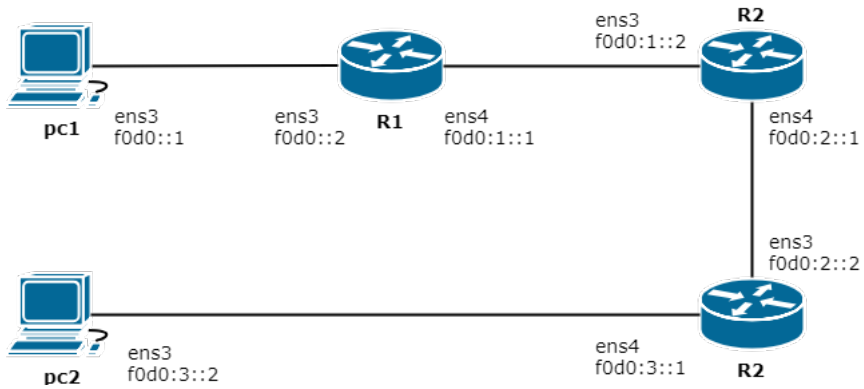


FIGURE 5.1: Network topology used for the evaluation of the performance

During the evaluation of the performance, the pc1 sends some packet having as destination the pc2. Those packets will be modified by the 3 hops present on the path. Each hop performs some specific action on the packet. Those actions are described in the following list.

- R1: this hop inserts at a certain frequency which is given as a parameter of the ioctl, the created Hop-By-Hop options header containing the

iOAM pre-allocated trace if the ingress interface of the packet is ens3. This Hop-By-Hop options header has been made thanks to the ioctl as described in the section 4.1. It also records the information asked by the iOAM trace type if it is able to do it.

- R2 : if this hops receives a packet containing an iOAM pre-allocated trace in the Hop-By-Hop options header, it records the desired information in the trace if it is able to do it.
- R3 : if this hops receives a packet containing an iOAM pre-allocated trace in the Hop-By-Hop options header, it records the desired information in the trace if it is able to do it. If the packet containing the iOAM pre-allocated trace has as ingress interface ens3, this hop will extract the trace and display the collected information in the kernel log.

There are several measurements which have been made. The first one concerns the throughput. This test is used to see how the throughput is affected when a certain percentage of packet carries the iOAM pre-allocated trace.

To perform this measurement the tool `iperf3` [8] was used. This tool allows generating traffic with a specific throughput between two hosts using TCP packets and records at a certain frequency the observed throughput. To do that, a host which is in this case the pc2 runs the server side of the exchange. Thanks to the following command : `iperf3 -s -p 5024`, this host listens on a specific port (5024 in this case) and computes the observed throughput thanks to the amount and the size of received packets in a lapse of time. An other host which is in this case the pc1, generates a traffic towards the destination to obtain the asked throughput. This is done thanks to the following commands : `iperf3 -c f0d0:3::2 -6 -b 10m -t 60 -p 5024`. The `-c` means that it is the client, then comes the address of the server which is in this case the address of the pc2, the `-6` specifies that the test must only be done with IPv6, the `-b 10m` indicates that the desired throughput is equal to 10 Mbits/s, the `-t 60` is for the number of seconds during which the test must be performed and the `-p 5024` specifies the destination port number. Because there is no specification in this command about the frequency at which the throughput must be observed, it will be at every second.

Three tests with different throughput have been made. The first one uses a generated throughput of 10 Mbits/s the second one a throughput of 100 Mbits/s and the last one a throughput of 1 Gbits/s. For each given throughput, a test which lasts 60 seconds has been made 30 times. This number of times has been chosen due to the law of big numbers. The inserted iOAM pre-allocated trace only carries the hop limit and node identifier.

The second measurement looks at the time needed for the different operations which are the insertion of the Hop-By-Hop options header, the recording of the asked information and the extraction of the trace. Those different times are recorded for each packet using the same methodology as for the one used to perform the next measures.

The last measurement looks at the evolution of the RTT of a packet according to the number of asked data in the trace. For that the ping6 tools

was used because it gives the RTT and only uses IPv6 packets. Moreover with this tool, the size of the packet can be modified. To see if the RTT of a packet is affected by the number of asked information in the iOAM trace, the following test has been made: the pc1 sends a burst of 60 pings having as destination the pc2 for 3 different payload sizes. Thanks to the 3 different payload sizes, the sent packets have a size of 512 bits, 1 Kbits or 10 Kbits. This has been done with the following command : `sudo ping6 f0d0:3::2 -I ens3 -c 60 -i 0`. The first hop which is r1, at first does not insert the Hop-By-Hop options containing the iOAM pre-allocated trace to see the RTT of a normal ping. After that it inserts an iOAM pre-allocated trace which asks more and more information. At the most, 6 information are asked at the same time. For each possible configuration made from the previous parameters which are the number of asked data and the size of the packet, the test has been performed 30 times due to the law of big numbers.

5.2 Results and discussion

Now that the methodology has been described, the results can be showed and discussed.

5.2.1 Throughput

The first measurement made concerns the impact of iOAM on the throughput. Figure 5.2 shows how the throughput evolves according to the fraction of iOAM packet. As it can be seen on the graph, with 10% of iOAM packets for a generated throughput of 10 Mbits/s and with 1% of iOAM packets for a generated throughput of 100 Mbits/s, the observed throughput is equal to the generated one.

For a generated throughput of 1 Gbits/s, the observed throughput only reached the generated one when the percentage of iOAM packets is lower or equal to 0.01%. More the observed throughput approached 1 Gbit/s, more the confidence interval is wide. It is partly due to the fact that the machine has some difficulty to generate enough packet to reach the wanted throughput with this number of hosts even without iOAM. Consequently, this result can be pessimistic. So it is possible that if the machine has not those difficulties, the observed throughput could reach the generated throughput of 1 Gbits/s with a higher percentage of packets containing the iOAM trace than what it shows on the graph.

Of course as it is depicted on the graph, it is not possible to use iOAM on every packet since the observed throughput tends to be equal to zero when every packet carry the trace. This drastic decrease of throughput is due to the slow down of packet processing. This slow down is brought by the time needed to insert the Hop-By-Hop options header, to record the asked data and to extract the iOAM pre-allocated trace.

This graph could be an indicator of the lower bound to correctly choose the fraction of packet which will have the iOAM pre-allocated trace.

But it must be taking into account that here, it is inserted on a fraction of packets which arrives at a specific interface, of course in a real world, all the packets will not have the same destination so if a filter is applied to the destination address of the packets, the fraction of packet with iOAM could be increased.

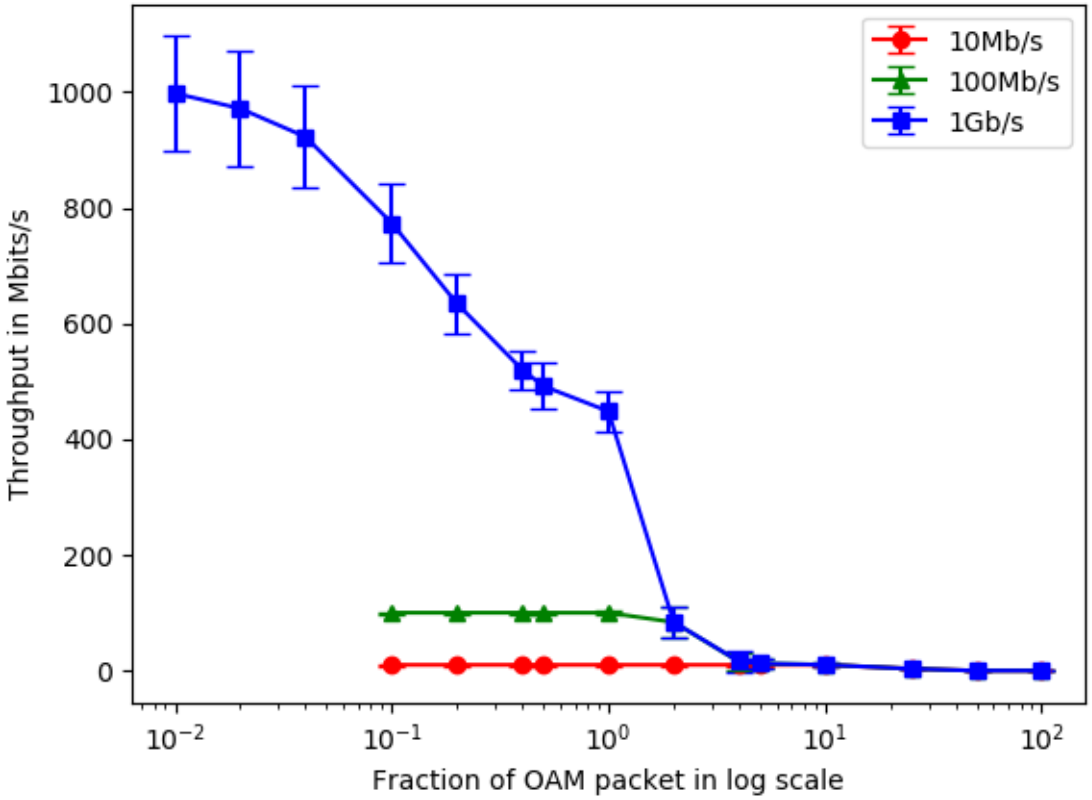


FIGURE 5.2: Mean and 95 percent confidence interval of throughput with iOAM

5.2.2 Operations time

In this part, the time needed to perform the insertion of the Hop-By-Hop options header containing the iOAM pre-allocated trace, to record the asked data and to extract the trace will be analyzed.

Figure 5.3 shows the time needed for each operation and for each node which performs them according to the number of data asked in the iOAM trace.

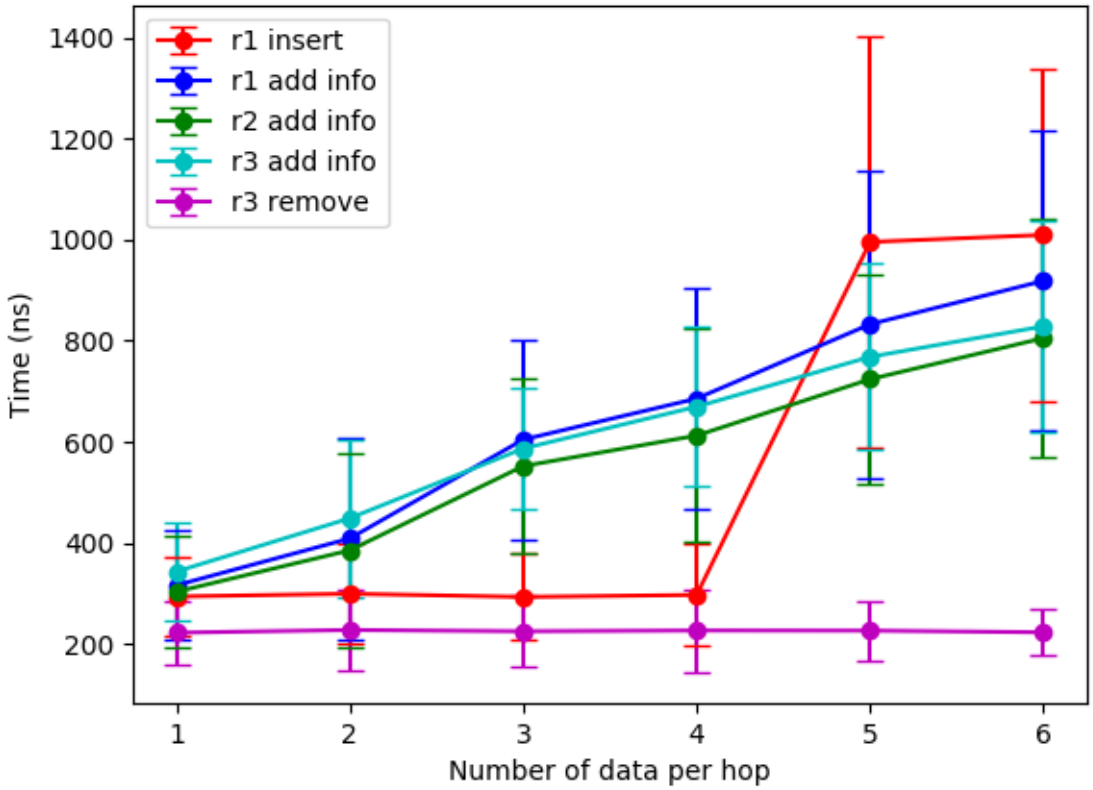


FIGURE 5.3: Mean and 95 percent confidence interval of the time needed for the operations

The list below depicts which data are asked according to the number of data present on the x axis the graph. It is a cumulative list, so each time a new data is asked in the trace, all the previous ones also remain in it.

1. The hop limit and the node identifier.
2. The ingress and egress interface identifier.
3. The timestamp in seconds.
4. The timestamp in nanoseconds.
5. The hop limit and node identifier in its large format.
6. The ingress and egress interface identifier.

At first, the red curve which is the time needed to insert the Hop-By-Hop options header in the IPv6 packet will be analyzed. So as it is shown on Figure 5.3 until the iOAM pre-allocated trace contains at most 4 data, the time of the insertion remains constant. This time is the time needed to copy the Hop-By-Hop options header containing the trace in the packet and then recopy the IPv6 header above the added extension header.

But from the moment where the number of options is greater or equal to 5, the time needed for the insertion of the header increases a lot. This jump can be explained due to the need of memory. Indeed when a packet arrives in the Linux kernel, it is stored in the `skb` structure. This structure has some unused space at the beginning but this space is not infinite so at a moment this free space is not wide enough to receive the Hop-By-Hop options header containing the iOAM pre-allocated trace. Consequently it is asked to the kernel to allocate more memory at the beginning of the structure. As usual this operation consumes a lot of time.

Now, have a look at the purple curve which represents the time need for the extraction of the iOAM pre-allocated trace. Only two operations are performed. The first one just displays the collected information in the kernel log. Since this kind of operation is costly, they are not always done directly but when it is expressly asked to be done, when the buffer used to delay those operations is full or when there is time to do it without slowing down the other processes.

The second one is in fact composed of several small operations. As said previously, the structure representing a packet in the Linux kernel has some free space at the beginning. To know where really starts the packet there is a pointer. So to remove the trace, this pointer must be shifted to represent the fact that the space taken by the trace is no more in the real packet and then the IPv6 header must be shifted at its new place. Since the IPv6 header always has the same size, this operation always takes the same amount of time. So it explains why this time is more or less constant.

In this cases, the Hop-By-Hop options header only has the iOAM pre-allocated trace so it is totally extracted from the packet. If it has not been the only option present in it, the time required for the extraction will be higher since more memory shift would be performed.

Finally, the curves representing the time needed for the recording of the data in the trace will be analyzed. It can be seen that there is a big increase in the time needed when 3 data are asked compared when only 2 are asked. This is due to the time needed to to compute the timestamp. Since the time needed to get it is value is quite expensive, it could be interesting to think of a trade-off between accuracy and performance for this particular piece of information. Because the timestamp is not recomputed if another piece of information has already asked it, it is normal to see that the increase of time needed to insert the timestamp in nanoseconds so the fourth datum, is very small compared to the previous one which is the time needed to add the timestamp in seconds.

After the fourth datum the slope of the curve increases a little. This can be explained by the size of the recorded data. Indeed until the fourth datum, each asked data have a size of 32 bits but after that they have a size of 64 bits.

5.2.3 RTT

The last measurement made concerns the modification of the RTT of a ping due to iOAM. There are three figures, one for each packet size. The first Figure 5.4 represents the percentile 0, 25, 50, 75 and 100 of the RTT of a packet which has a size of 512 bits.

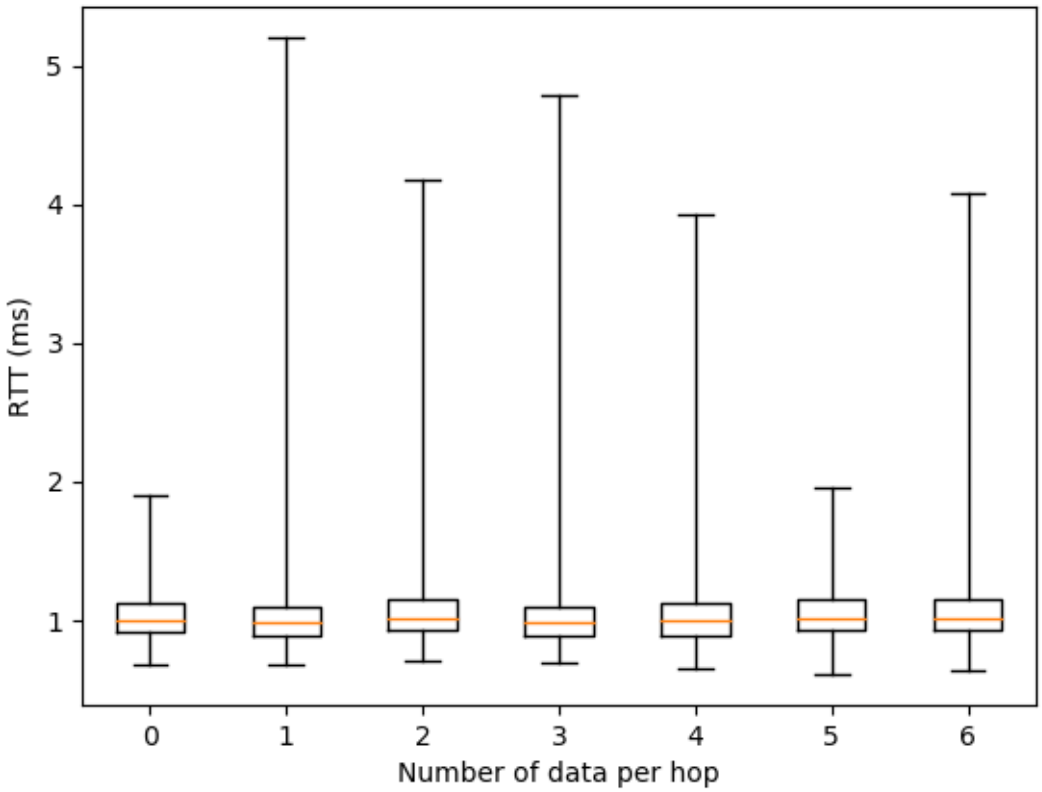


FIGURE 5.4: Percentile 0, 25, 50, 75 and 100 for the RTT of a ping of 512 bits according to the number of data

As it can be seen in the figure, there is no noticeable difference between the RTT of a packet without iOAM and a packet with iOAM. This can be explained thanks to the graph of the time needed per operations 5.3. Indeed this time is in nanosecond so the difference can not be seen since the RTT is in millisecond. Also, the node is not overloaded by the number of packets. Therefore this small amount of time needed due to iOAM is not big enough to make congestion. If the number of packets was higher, the RTT of packets with iOAM should be bigger since at this moment there would be some congestion and some packet could be discarded. Indeed, on Figure 5.2, when all the packet get iOAM the bandwidth is nearly equal to zero since the time required to operate make the packet processing more slower.

Also the figure shows that the RTT does not vary according to the number of options since the percentile 0, 25, 50 and 75 are nearly equal for

each number of options. There is some big value but they are very rare as the percentile shows it. The values are mostly centered around 1.05 ms.

Figure 5.5 is the same than the previous one apart that now the packet has a size of 1 Kbits instead of 512 bits. The same conclusions than for the RTT with packet having a size of 512 bits could be learned.

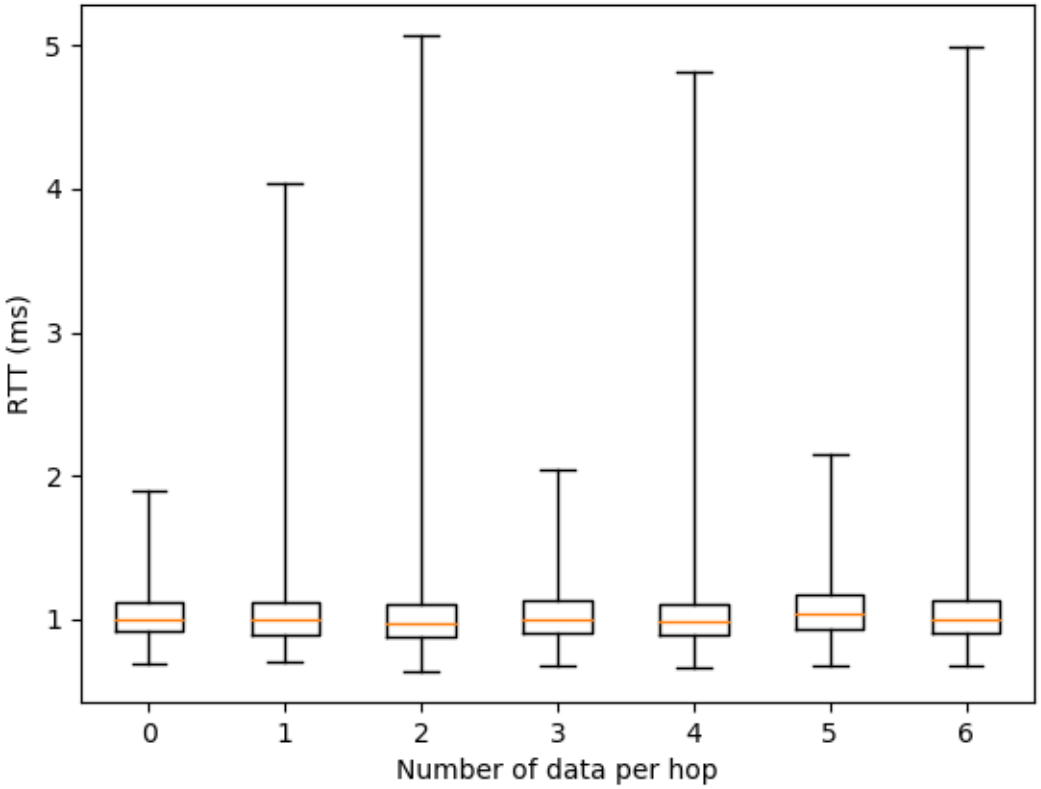


FIGURE 5.5: Percentile 0, 25, 50, 75 and 100 for the RTT of a ping of 1 Kbits according to the number of data

Figure 5.5 is the same than the previous one apart that now the packet has a size of 10 Kbits instead of 512 bits. And as previously the same conclusions than for the RTT with packet having a size of 512 bits could be learned.

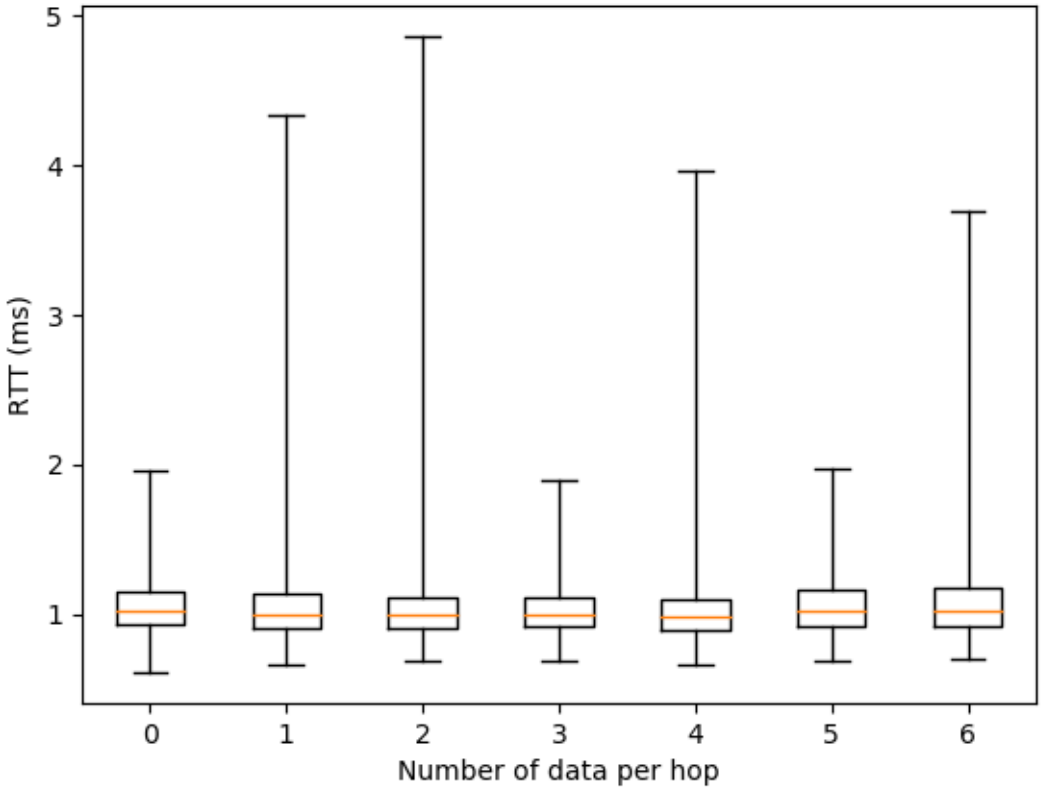


FIGURE 5.6: Percentile 0, 25, 50, 75 and 100 for the RTT of a ping of 10 Kbits according to the number of data

Chapter 6

Conclusion

In this master thesis, the base of iOAM has been implemented in the Linux kernel. The trace option of iOAM is already well developed even if there is still some work which must be done to make a node able to record all the possible data which can be collected for the moment. Of course, since the format of iOAM is still in discussion at the IETF, it is possible that there will be a need of modification to fit the future specifications and also if new collectible data appear since not all the bit of the iOAM trace type are used.

But thanks to this work there is now a way to create the Hop-By-Hop options header thanks to an `ioctl`, to insert it in an IPv6 packet and extract the trace from a packet. Thanks to the `sysctl` API, there is a way to set some value like the node identifier, the ingress and egress identifier, the application data and configure an interface of a node to make it extract any iOAM trace present in a packet. Like that it will be easier to modify like adding some filters to select on which packet the trace will be inserted or adding new data in the trace. Also since this is put in place, it will reduce the time needed to add the other iOAM options which are the proof of transit option and the edge to edge option.

Moreover the ioctl is also capable to create a segment routing header with the addresses given in parameter and insert it in the packet at the same moment as the Hop-By-Hop options header containing the iOAM pre-allocated trace. This segment routing header uses the implementation made by David Lebrun and Olivier Bonaventure in the Linux kernel [16] for its processing. But for the moment there is still a problem with it. Indeed, using `wireshark` it can be seen that the packet is well formed and respects the modification of address made but when the packet arrives at the first destination present in the routing segment it disappears. For the moment the reason why it happens has not been found.

6.1 Improvement

Some possible improvements have already been noticed in the previous section. They are resumed in the following list.

- One improvement is to reduce the time needed to get the timestamp. Indeed has seen on Figure 5.3, this operations is very costly in terms of time. So it could be interesting to explore a way to have a trade-off between accuracy and performance like placing the timestamp in

a variable which will be updated at a given frequency. Indeed thanks to that the time required to record the data in the trace will be smaller since there is just need to read a value. If someone uses the timestamp, knowing the frequency at which the value is updated, he can infer an interval of time in which the right value belongs.

- An other improvement which can be done is about the extraction of the iOAM pre-allocated trace especially how someone can retrieve the data collected by the trace during the journey of the packet in the network. Indeed at the moment those data are displayed in the log of the kernel. But thanks to the different function of the `ioctl`, it is possible to make a memory mapping between user space memory and kernel space memory. Like that a ring buffer could be put in place and when the trace is extracted from the packet, the recorded data could be placed in this ring buffer. After that, a user program would be able to get access to the data on this ring buffer. This improvement could possibly make the process more efficient and the recovery of the data by a program easier.

Bibliography

[1] URL: https://github.com/FrancartJ/iOAM-IPv6-LinuxKernel.

[2] 2018. URL: "https://www.tcpdump.org/tcpdump_man.html".

[3] 2018. URL: "https://www.wireshark.org/".

[4] June 2018. URL: "https://ipv4.potaroo.net/".

[5] Mar. 2018. URL: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml.

[6] May 2018. URL: https://www.google.com/intl/en/ipv6/statistics.html#tab=ipv6-adoption&tab=ipv6-adoption&tab=per-country-ipv6-adoption.

[7] June 2018. URL: http://www.tcpipguide.com/free/t_IPv6IPv4AddressEmbedding-2.htm.

[8] 2018. URL: "https://iperf.fr/fr/".

[9] B. Orgogozo F. Viger T. Friedman M. Latapy B. Augustin X. Cuvellier and C. Magnien. "Avoiding traceroute anomalies with Paris traceroute". In: *Proc. ACM Internet Measurement Conference (IMC)* (Oct. 2006). URL: https://conferences.sigcomm.org/imc/2006/papers/p15-augustin.pdf.

[10] Inc. D. Johnson C. Perkins Ed. Tellabs and J. Arkko. *Mobility Support in IPv6*. RFC 6275. Internet Engineering Task Force, July 2011. URL: https://tools.ietf.org/html/rfc6275.

[11] B. Carpenter and S. Jiang. *Transmission and Processing of IPv6 Extension Headers*. RFC 7045. Internet Engineering Task Force, Dec. 2013. URL: https://tools.ietf.org/html/rfc7045.

[12] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. Internet Engineering Task Force, Dec. 1998. URL: https://tools.ietf.org/html/rfc2460.

[13] C. Pignataro H. Gredler J. Leddy S. Youell T. Mizrahi D. Mozes P. Lapukhov R. Chang D. Bernier F. Brockners S. Bhandari and J. Lemon. *Data Fields for In-situ OAM draft-ietf-ippm-ioam-data-02*. Draft draft-ietf-ippm-ioam-data-02. Internet Engineering Task Force, Mar. 2018. URL: https://tools.ietf.org/html/draft-ietf-ippm-ioam-data-02.

[14] *INTERNET PROTOCOL*. RFC 791. Internet Engineering Task Force, Sept. 1981. URL: https://tools.ietf.org/html/rfc791.

[15] S. Prevost J. De Clercq D. Ooms and F. Le Faucheur. *Connecting IPv6 Islands over IPv4 MPLS Using IPv6 Provider Edge Rotuers (6PE)*. RFC 4798. Internet Engineering Task Force, Feb. 2007. URL: https://tools.ietf.org/html/rfc7045.

[16] D. Lebrun and O. Bonaventure. "Implementing IPv6 Segment Routing in the Linux Kernel". In: *Proc. ACM Applied NEtworking Research Workshop (ANRW)* (July 2017). URL: https://inl.info.ucl.ac.be/system/files/paper_10.pdf.

[17] R. Atkinson M. StJohns and G. Thomas. *Common Architecture Label IPv6 Security Option (CALIPSO)*. RFC 5570. Internet Engineering Task Force, July 2009. URL: https://tools.ietf.org/html/rfc5570.

[18] E. Nordmark and M. Bagnulo. *Shim6: Level 3 Multihoming Shim Protocol for IPv6*. RFC 5533. Internet Engineering Task Force, June 2009. URL: https://tools.ietf.org/html/rfc5533.

[19] C. Partridge and A. Jackson. *IPv6 Router Alert Option*. RFC 2711. Internet Engineering Task Force, Oct. 1999. URL: https://tools.ietf.org/html/rfc2711.

[20] P. Jokela R. Moskowitz T. Heer and T. Henderson. *Host Identity Protocol Version 2 (HIPv2)*. RFC 7401. Internet Engineering Task Force, Apr. 2015. URL: https://tools.ietf.org/html/rfc7401.

[21] T. Narten S. Thomson and T. Jinmei. *IPv6 Stateless Address Autoconfiguration*. RFC 4862. Internet Engineering Task Force, Sept. 2007. URL: https://tools.ietf.org/html/rfc4862.