EFFICIENT TRANSLATION IN COEXISTENT IPV4-IPV6 NETWORKS

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree of

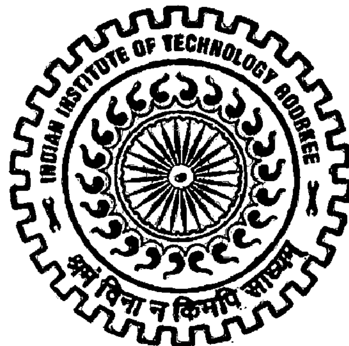
MASTER OF TECHNOLOGY

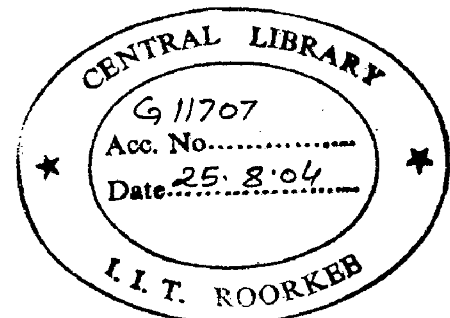
in

INFORMATION TECHNOLOGY

By

PREM DASARI




CENTRAL LIBRARY
G 11707
Acc. No.
Date 25.8.04
I. I. T. ROORKEE

IIT Roorkee - CDAC, NOIDA,
c-56/1, "Anusandhan Bhawan"
Sector 62, Noida-201 307

JUNE, 2004

CANDIDATE'S DECLARATION

I hereby declare that the work presented in this dissertation titled " **EFFICIENT TRANSLATION IN COEXISTENT IPV4-IPV6 NETWORKS",** in partial fulfillment of the requirements for the award of the degree of **Master of Technology** in **Information Technology**, submitted in **IIT-Roorkee - CDAC campus, Noida,** is an authentic record of my own work carried out during the period from June 2003 to June 2004 under the guidance of **Dr. Poonam Rani Gupta**, Associate Professor, CDAC, Noida.

I have not submitted the matter embodied in this dissertation for the award of any other degree or diploma.

Date: 26-06-2004
Place: Noida

prem
(PREM DASARI)

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date: 26/6/2004
Place: Noida

(Dr. Poonam Rani Gupta),
Associate Professor,
CDAC, Noida.

// ACKNOWLEDGEMENTS

I hereby take the privilege to express my deep sense of gratitude to **Prof. PREM VRAT**, Director, Indian Institute of Technology, Roorkee, and **Mr. R.K.VERMA**, Executive Director, CDAC, Noida for providing me with the valuable opportunity to carry out this work. I am very grateful to **Prof. A.K.AWASTI**, Programme Director, **Prof. R.P. AGARWAL**, course coordinator, M.Tech (IT), IIT, Roorkee and **Mr. V.N.SHUKLA**, course coordinator, M.Tech (IT), CDAC, NOIDA for providing the best of the facilities for the completion of this work and constant encouragement towards the goal.

I express my sincere thanks and gratitude to my Guide **Dr. POONAM RANI GUPTA**, Associate Professor, CDAC, Noida, for her inspiring guidance and sustained interest throughout the progress of this dissertation.

I am thankful to **Mr. R.K.SINGH** and **Mr. MUNISH KUMAR**, project engineer, CDAC Noida, for providing necessary infrastructure to complete the dissertation in time.

I owe special thanks to my friends, all of my classmates and other friends who have helped me formulate my ideas and have been a constant support.

I thank my family members for their moral support.

prem

(PREM DASARI)

Enroll. No. 029016

ABSTRACT

Internet system is a collection of hosts, connected to each other. Communication among the hosts needs to have a protocol, which can be understood by each host in the Internet. IPv4 and the later version IPv6 protocols are designed for this purpose. Though IPv6 has begun to be deployed, IPv4 is still very widely used and it is impossible to expect a complete switch to IPv6 in the near future. It is therefore expected that IPv4 and IPv6 will co-exist for quite some time to come. Hence, an IPv4 host may need to communicate with an IPv6 host and vice-versa. This will necessitate translating an IPv6 packet to an IPv4 packet and vice-versa at network boundaries. The router sitting at the end of the network does translation of an IPv4 packet to an IPv6 packet and vice-versa at the network boundaries. We first designed and developed a mechanism in which a router whenever receives a packet from the IPv4 subnet intended for the IPv6 subnet, the router will translate the packet. The same thing happens when the router is forwarding a packet from the IPv6 network to the IPv4 network. To translate a packet from IPv4 to IPv6, each field in the IPv4 header is converted to a functionally equivalent field in the IPv6 header following the conversion scheme designed in the present work. Since ICMP is considered an integral part of IP, a mechanism for converting an ICMPv4 packet to an ICMPv6 packet and vice-versa has also been developed and implemented. This mechanism has been deployed on a Linux-based router and tested rigorously. The test results have been analyzed for correctness and estimated the time required for translating the packets.

List of Figures

2.1 IPv4 Primary address class format......................... 6

2.2 Format of CIDR address 7

2.3 IPv4 Internet header format..7

2.4 IPv4 Type of Service field..8

2.5 IPv4 TLV format 10

2.6 IPv4 LSSR option format 11

2.7 IPv4 SSRR option format 11

2.8 IPv4 RR option format.. 12

2.9 IPv4 Security Identifier option format 12

2.10 IPv4 Internet Timestamp option format...13

2.11 IPv4 Router Alert option format... 13

2.12 IPv6 Unicast Address 16

2.13 IPv6 Anycast Address 17

2.14 IPv6 Multicast Address..19

2.15 IPv4-Compatible IPv6 Address..17

2.16 IPv4-Mapped IPv6 Address 20

2.17 IPv6 header format..20

2.18 Example packet containing an IPv6 header as IP header 22

4.1 IPv4 header format... 29

4.2 IPv6 header format 30

4.3 IPv6 Pseudo header format...36

4.4 IPv4 Pseudo header format 37

4.5 ICMPv4 general header format 38

4.6 ICMPv6 Message general format..40

4.7 IPv4-to-IPv6 ICMP error message translation.. 42

4.8 IPv6-to-IPv4 ICMP error message type translation....................................44

5.1 Processing of a packet in a host.. 52

5.2 Processing of a packet in a host... 55

6.1 kernel Configuration options...57

6.2: Compiling the kernel...58

6.3: IP header and data fields of the packet generator..60
6.4: TCP header and data fields of the packet generator....................................61

List of Tables

2.1 IPv4 Address Classes.. 6

2.2 IPv4 Type of Service precedences..8

2.3 IP Router Alert option Values..14

2.4 IPv6 Multicast scope field values...17

2.5 IPv6 allocated addresses and their values. 19

4.1 ICMPv4 message types.. 39

4.2. ICMPv6 message types... 41

4.3. ICMPv4 to ICMPv6 query type translation. 43

4.4 ICMPv4 to ICMPv6 Error Reporting type translation. 43

4.5: ICMPv6 to ICMPv4 Query type translation...45

CONTENTS

Candidate's Declaration

Acknowledgements

Abstract

1. INTRODUCTION.. 1

1.1 Report Organization. 3

2. LITERATURE SURVEY...5

2.1 IPv4 Addressing scheme...5

2.1.1 Primary address classes..5

2.1.2 Classless Inter Domain Routing (CIDR)................................6

2.2 IPv4 header format...7

2.3 Need to migrate from IPv4 to IPv6..7

2.4 Addressing schemes in IPv6..8

2.4.1 IPv6 Address Types...15

2.5 IPv6 Header Format...19

2.5.1 IPv6 Extension Headers...21

2.6 IPv6 Research...23

3.ANALYSIS...26

3.1 Fragmentation Extension Header...26

3.2 Pseudo headers...27

3.3 Translation of ICMP...27

3.4 Mapping...27

4. DESIGN...29

4.1 IPv4 to IPv6 header translation... 29

4.2 IPv6 to IPv4 header translation... ...33

4.3 Upper Layer protocol checksum calculation................................36

4.4 Translating ICMP..37

4.4.1 ICMP version 4 (ICMPv4)...38

4.4.2 ICMP version 6 (ICMPv6). 39
4.4.3 ICMPv4 to ICMPv6 translation......................................42
4.4.4 ICMPv6 to ICMPv4 translation. 44
4.4.5 ICMP checksum calculation......................................45

5.IMPLEMENTATION......................................49
5.1 Processing of a packet at a receiving host 52
5.2 Files changed 55
5.3 Testing......................................56

6.RESULTS......................................57
7.CONCLUSION AND FUTUTRE WORK......................................69
REFERENCES

Chapter 1

INTRODUCTION

Internet system is a collection of hosts, connected to each other. Hosts may not be directly connected to one another (one-to-one connection). To identify a host in the Internet system, that host needs to have some identity. This identity is called the IP **[1, 16]** address of the host. Each host in the Internet system needs to have at least one IP address. In the current version of IP, version 4, an IP address is 32 bits only. This address is unique and can be broken up into a network part and a host part.

The Internet Protocol (IP) is a part of TCP/IP Protocol Suite, which is developed as a part of Defense Advanced Research Projects Agency (DARPA) Internet project. IP has the functionality similar to Open Systems Interconnection (OSI) Model Connection Less Network Protocol (CLNP) layer (which is the 3 rd layer). IP resides within layer 2 (network layer) of TCP/IP Protocol Suite, which has the functionality of routing packets and controlling congestion. IP is the lowest layer that deals with end-to-end (host to host) transmission of packets through connectionless services.

Since IPv4 address is of 32 bits length, Internet Assigned Numbers Authority (IANA) can accommodate a maximum of 2^{32}. This number is big, however, due to the rapid explosion of the Internet and the wastage of addresses in Class A and Class B types, the Internet system has been running out of addresses. Even though Classless Inter Domain Routing (CIDR) **[17]** protocol provided a temporary solution, still it was not sufficient to solve this problem completely. The new version of IP, IPv6 was designed to accommodate 128 bit addresses, thereby increasing the address space vastly. IPv6 also addresses some other shortcomings of IPv4 such as variable header length etc., enabling faster processing of IP packets in modern day high-speed networks.

Though IPv6 has begun to be deployed, IPv4 is still very widely used and it is impossible to completely switch to IPv6 [3] immediately. It is therefore expected that IPv4 and IPv6 will co-exist for quite some time to come. Hence, an IPv4 host may need to communicate with an IPv6 host and vice-versa. This will necessitate translating an IPv6 packet to an IPv4 packet and vice-versa at network boundaries.

Translation of an IPv4 packet to an IPv6 packet and vice-versa at the network boundaries is the main aim of this work. In this work we have first developed a scheme to convert an IPv4 packet to an IPv6 packet and vice-versa. Following that the proposal is implemented on a system, in which the router, when forwarding a packet from an IPv4 network to an IPv6 network, will translate the IPv4 packet into the IPv6 packet. The same thing happens when the router is forwarding a packet from the IPv6 network to the IPv4 network. Since ICMP is considered an integral part of IP, a mechanism has been developed and implemented for converting an ICMPv4 [17] packet to an ICMPv6 [16] packet and vice-versa. The router, which is sitting at the end of the network boundary of the IPv6, will translate an IPv4 packet coming from the Internet system to an IPv6 packet and vice-versa. Although the proposals for this method have been specified, modern day operating systems do not provide an integrated solution that can be used to support IPv4-IPv6 communication. The latest Windows server 2003 server provides dual IPv4/IPv6 stacks and support tunneling, but protocol translation is not supported. Thus an IPv4 host to communicate with an IPv6 host and viceversa cannot use it. Linux provides both IPv4 and IPv6 stacks, but router configuration can be IPv4-only or IPv6- only no interoperation is supported. The proposed mechanism helps us to achieve interoperation between both the IPv4 and IPv6 stacks in Linux.

The proposed mechanism has been implemented and deployed on a Linux-based router and tested rigorously. The test results have been analyzed for correctness and for estimating the time required for translating the packets. The translation time has been found to be quite low in our implementation.

1.1 Report Organization

Chapter 2 describes the IPv4 and IPv6 protocol in detail such as their header formats and addressing schemes. Chapter 3 analyzes the issues of translation such as handling fragmentation extension header, mapping issues etc. Chapter 4 explains the translation mechanism design of an IPv4 packet to an IPv6 packet and vice-versa. It also discusses issues concern about the packets having Internet Control Message Protocol (ICMP) in its upper layer and also describes translation mechanism of these packets from one protocol to another. Chapter 5 explains implementation of the translation mechanism, testing. This Chapter includes a brief explanation about the packet traversal in an end system (a host which is receiving packets). Chapter 6 speaks about the results of the work, about the kernel recompilation, taking statistics etc. Chapter 7 concludes the work done till now and explains extension to this mechanism, which can be dealt as future works.

Chapter 2

LITERATURE SURVEY

In this Chapter we first discuss the addressing scheme and the header format of IPv4. The shortcomings of this protocol are discussed next. Then the need for a new protocol has been discussed which will fit in the TCP/IP Protocol Suite network layer. The new Internet Protocol provides more efficiency than IPv4. The new addressing schemes, header format and extension headers of the new protocol have been discussed and finally the work done in the current area has been presented. Most of the current document has been produced by referring to the different Requests for Comment (RFC) documents regarding IPv4 and IPv6 protocols.

2.1 IPv4 Addressing Schemes

Internet system is a collection of hosts, which are connected to each other. Hosts may not be directly connected to one another (one-to-one connection). To identify a host in the Internet system, that host needs to have some identify. This identity is called the IP address of the host. Each host in the Internet system needs to have at least one IP address.

Internet Assigned Numbers Authority (IANA) **[13]** is the governing body to assign addresses to any host. Any host connected to the Internet system should have a 32-bit long unique address. Hosts can be grouped together into a network and the 32-bit address is usually broken up into two parts - a network part to indicate the network to which the host belongs, and a host part to identify the host in this network. There are two schemes for allocation of addresses to hosts.

2.1.1. Primary address classes

Addresses belonging to these classes have the format shown in the Figure 2.1.To have the flexibility of assigning numbers to each host, address is been divided into two halves as shown in

0 31

Figure 2.1: IPv4 Primary address class format

Figure 2.1. Network Address and Host Address are divided in a way such that there are a small number of networks with a large number of hosts, a moderate number of networks with a moderate number of hosts, and a large number of networks with a small number of hosts. Each of this type can be specified as a class and the address classes are shown in Table 2.1.

High Order Bits	Format	Class
0	7 bits of net, 24 bits of Host	A
10	14 bits of net, 16 bits of Host	B
110	21 bits of net, 8 bits of Host	C
1110	24 bits for multicast	D
11110	Reserved for future use	E

Table 2.1: IPv4 Address Classes

2.1.2 Classless Inter Domain Routing (CIDR)

Class A [19] networks may not use all of the 2^{24} addresses allocated to it. If a network has small number of hosts it can have a Class C [19] network address. If a network administrator wants to expand a network with some more hosts, it may so happen that all hosts may not be within the same network. In this case, the administrator may need to get one more Class type network. If this happens everywhere in the world then the router routing table will be exploded with specific network masks (which is useful in finding the network address and route the packet to the appropriate route).

To avoid the above problem, a new scheme is devised which is called as CIDR. In this scheme, contiguous Class C type network addresses are allocated to a geographical region.

0	31
Network number (m bits)	Host Number (32-m) bits

Figure 2.2: Format of CIDR address

Who ever want to expand their network will ask for a new network address based on their geographical region. With this scheme, doing a general network masking on the packet rather than applying specific network mask can minimize routing table expansion. If a router identifies a packet as belonging to the same geographical region as it is connected, then that router again use more specific network mask to route the packet. In this way explosion of routing tables can be solved. The format of this scheme is shown in Figure 2.2

2.2 IPv4 Header Format

This section describes IPv4 header format. Format of the IPv4 header is shown in Figure 2.3. The IPv4 header contains the following fields:

Version (4 bits): This field indicates the version of the Internet Protocol (which is 4 here).

0 3	4 7	8 15	16	31
Version	IHL	Type of Service	Total Length	
Identification			Flags	Fragment Offset
Time to Live		Protocol	Header Check sum	
Source Address				
Destination Address				
Options				Padding

Figure 2.3: IPv4 Internet header format

IHL (4 bits): Internet Header Length (IHL) **[17]** is the length of the Internet Header in 16 bit (4 octets) words. Minimum value of this field for a correct header is 5 (for the header which does not have any options).

Type of Service (8 bits): This field provides precedence of the packet as well as some of the desired

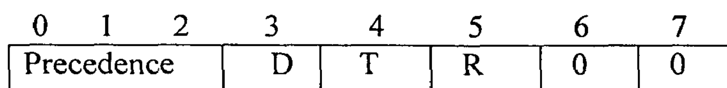


Figure 2.4: IPv4 Type of Service field

features, like throughput(T),delay(D),and reliability(R). These features will give an idea about the handling of the packet in a particular network. Figure 2.4 explains how the 8 bits of this field are used in handling the packet.

- *Bits 0 to 2*: Indicates the precedence of the packet (from low to high). A high value has high precedence and a low value has low precedence.
- Bit - 3 (D): Indicates whether the packet can be delayed or not. If this bit is not set (which is zero) means normal Delay, if set (which is 1) means low delay.
- *Bit - 4 (T)*: Indicates throughput of the connection. If set high Throughput else normal Throughput.
- *Bit - 5 (R)*: Indicates reliability of the packet. If set high Reliability else normal Reliability.
- *Bits 6 and 7*: Reserved for future use (always set to zero).

Table 2.2 gives some of the defined precedences. Note that it is upto the individual ISP's to decide whether to honor these precedences or not.

Precedence Bits	Description
111	Network Control
110	Internet work Control
101	CRITIC/ECP
100	Flash Override
011	Flash
010	Immediate
001	Priority
000	Routine

Table 2.2: IPv4 Type of Service precedences

Total Length (16 bits): This field specifies the length of the packet, measured in octets, including the IPv4 header and data in the packet. This field allows packet length up to 65,535 octets (64K Bytes).

Identification (16 bits): This field is used to specify an unique identifier for the combination of source host, destination host, and applications on the two sides. This field is useful in assembling fragmented packets at the destination. (Packets that are larger than Path Maximum Transmission Unit (PMTU) will be fragmented. PMTU is the maximum packet size that can be sent along a path).

Flags (3 bits): These are useful in reassembling fragmented packets. The field has the following bits.

0	1	2
0	DF	MF

- Bit 0: Reserved for future use, and should be zero.
- Bit 1: If DF is 0 then this packet can be fragmented else don't fragment this packet.
- Bit 2: If MF is 0 then this is the last fragment else more fragments are going to follow this packet.

Fragment Offset (13 bits): This field specifies the offset of this packet data in the original packet (if the packet is fragmented). The fragment offset is measured in units of 8 octets (64 bits). The first fragment has offset zero.

Time to Live (TTL) (8 bits): This field indicates the maximum lifetime of a packet (datagram) in the Internet system. Each time a packet passes through a router, its TTL**[1]** field is decremented by 1. If the TTL field reaches zero, the packet is discarded. The intention is to not allow undeliverable packets to circulate in the network forever.

Protocol (8 bits): This field indicates the next higher-level protocol, such as TCP, UDP etc that is using this packet.

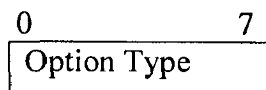
Header Checksum (16 bits): This option contains checksum on the header only, including any options that may be present. Since some header fields will be changed (e.g., TTL) by the routers in transmission of the packet, this field value is recomputed and verified at each point at which the header is processed.

Source Address (32 bits): This field indicates the IPv4 address of the source host of this packet (generator of this packet).

Destination Address (32 bits): This field indicates the IPv4 address of the destination host.

Options (Variable length): This field may appear or may not appear in a packet. This field contains any optional information that needs to be carried by the packet, which will be helpful to the router. This Options field has two formats to specify the options.

✓ *Format 1*: Contains only a single octet, which is called the Option Type. This type of options is helpful in padding.



✓ *Format 2*: In this format an Option Type octet, an Option Length octet, and the actual Option Data octets will be present. By looking at the values of Option Type and Option Length, we can know the length of the Option Data in octets and the data type present in the Option Data field. The number of data octets will be two less than the Option Length field. This type of format can be viewed as shown in Figure 2.5 and is called the Type Length Value (TLV) format.

Figure 2.5: IPv4 TLV format

The Options field contains TLV encoded options. The various possible TLV encoded options are discussed below:

1. *End of Option List* (Option Type - 0): This is of type Format 1, which is explained in the previous paragraph. The option length is one octet and this has the Option Type value as zero (0x00). This option indicates the end of the Options field in the IPv4 header.

2. *No Operation* (Option Type - 1): This option may be used between two TLV encoded options, for example, for the alignment of 32-bit boundary of IPv4 header. This option is of type Format 1 and the value of Option Type field is 1 (0x01).

3. *Security* (Option Type - 130): This option is of type Format 2 and has three fields specifying TLV. This option has the value 130 as Option type and 11 as Option Length. So there will be 9 octets of data, which is specific to this option. This option is useful in security applications.

4. *Loose Source and Record Route* (LSSR) (Option Type – 131): This option provides a mean for the source of a packet to supply routing information to be used by the routers in forwarding the packet to the destination, and to record the route information. The option begins with the Option Type field in Figure 2.6, which has the value of 131. The second octet is the Option Length in Figure 2.6, which specifies length of the data including the first two octets. The third octet is the pointer into the route data indicating the octet, which begins the next source address to be processed. The pointer is relative to this option, and the smallest legal value for the pointer is 4. The general Format of this option is shown in Figure 2.6

Figure 2.6: IPv4 LSSR option format

5. *Strict Source and Record Route* (SSRR) (Option Type - 137): This option is similar to LSSR. However, any intermediate host should strictly follow the route registered, and is not allowed to go through other routers. A router should discard the packet if can't reach the next address. Option

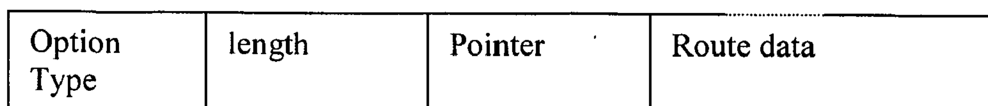


Figure 2.7: IPv4 SSRR option format

Type field will have the value 137. The format of this option is shown in Figure 2.7.

6. *Record route (RR)* (Option Type - 7): This option provides a means to record the route of an IPv4 packet. When a router routes a packet it checks to see if the RR option is

present. If it is present, then it inserts its own IPv4 address into the recorded route, beginning at the octet indicated by the pointer, and increments the pointer by four. If the route data area is already full (the pointer field value exceeds the length field value) then the packet is forwarded without inserting the address into the recorded route. If there is some room but not enough room for a full address to be inserted, then the original packet is considered to be in error and is discarded. In either case an Internet Message Control Protocol (ICMP) [16] parameter problem message may be sent to the source host of this packet. The value of the Option Type field (Opt Type in Figure 2.8) is 7. The format of this option is shown in Figure 2.8.

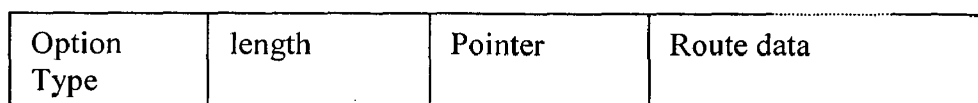


Figure 2.8: IPv4 RR option format

7. *Stream Identifier* (Option Type - 136): This option provides a way for the 16-bit SATNET stream identifier to be carried through networks that do not support the stream concept [9]. This is a very rarely used option. The value of the Option Type field is 136 (Opt Type in Figure 2.9). The format of this option is shown in Figure 2.9.

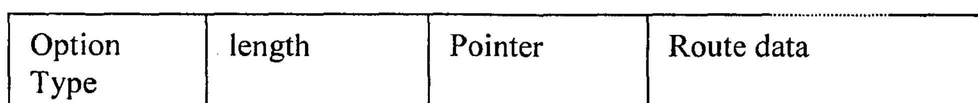


Figure 2.9: IPv4 Security Identifier option format

8. *Internet Timestamp* (Option Type - 68): This option is useful in getting the timestamp of various intermediate hosts, if required. The pointer field value is the number of octets from the beginning of this option to the end of timestamps plus one (i.e., it points to the octet beginning the space for next timestamp). The smallest legal value is 5. The timestamp area is full when the pointer is greater than the length. The Overflow (oflow in Figure 2.10) (4 bits) is the number of routers (intermediate hosts processing this

packet) that cannot register timestamps due to lack of space. This option has 68 as the value of the Option Type field. The format of this option is shown in Figure 2.10. The possible values of the field flg in Figure 2.10, which is 4 bits wide, are:

Opt type	Length	Pointer	oflw	Flg
IPv4 address				
Timestamp				

Figure 2.10: IPv4 Internet Timestamp option format

0 - time stamps only, stored in consecutive 32-bit words.

1 - each timestamp is preceded with IPv4 address of the registering entity.

3 - the IPv4 address fields are pre-specified.

An IPv4 router only registers its timestamp if it matches its own address with the next specified IPv4 address. The timestamp [9] field is a right-justified, 32-bit timestamp in milliseconds since midnight Universal Time (UTC) (00:00:00, January 1, 1970)). If the time is not available in milliseconds or cannot be provided with respect to midnight UTC, then any time may be inserted as a timestamp provided the high order bit of the timestamp field is set to one to indicate the use of a non-standard value. If there is some room but not enough room for a full timestamp to be inserted, or the over count itself over then the original packet is considered to be in error and is discarded. In either case an ICMP parameter problem message may be sent to the source host.

9. *Router Alert (RA)* (Option Type - 148): The goal of this option is to provide a mechanism whereby routers can intercept packets not addressed to them directly, without incurring any significant performance penalty. This option has the semantic that routers should examine this packet more closely. The format of this option is shown in Figure

Type	Length	Value (2 octet)

Figure 2.11: IPv4 Router Alert option format

2.11.This option has 148 as the value of the Option Type field and 4 as the value of the Option Length field. The Option Data field is referred as the Value field, which is 2-octet length, and Table 2.3 gives possible values of this field

Value	Description
0	Router shall examine packet
1-65535	Reserved

Table 2.3: IP Router Alert option Values

Padding (Variable Length): This is used to ensure that the IPv4 header ends on a 32-bit boundary.

2.3 Need to migrate from IPv4 to IPv6

IPv6 is the new Internet Protocol, which is designed to overcome some of the shortcomings of IPv4. New IP is called as IPv6 because its Version field has the value 6. Version value 5 is being used for the experimental real-time protocol. In Section 2.1, we discussed about the various address classes in IPv4. As an IPv4 address is of 32 bits length, IANA can accommodate only 2^{32} addresses to hosts. This number is big, but due to wastage of addresses in Class A and Class B types, the Internet system has been running out of addresses. Even though CIDR provided a temporary solution, still it was not sufficient to solve this problem. For this reason there is a need to devise a new protocol, which is more efficient than the existing IPv4 and also should co-exist with present Internet Protocol. In the process of creating that which will overcome all of the shortcomings of IPv4, IPv6 has been devised.

IPv6 maintains good features of IPv4 and discarded bad ones, and adds some new ones. Description below gives some of the features of IPv6.

- *Larger address*: Provided larger address bits so as to give more scope for the accommodation of the addresses. IPv6 has 128 bits as its address field.

- *Static header fields*: In contrast to IPv4 varying number of header fields, the number of fields in IPv6 is set to constant and Options are moved to a special area called as Extension Headers **[1]**. By having constant number of header fields intermediate hosts can process the header very fast and can route the packet as soon as possible.
- *Better support for Options*: In IPv4 options are resided in the main header, so router (intermediate host) can not skip them, because some options may be useful to the router, so it needs to process them and it is waste of time, to overcome this problem, options in this new protocol moved to special area called as Extension Headers. Extension Headers will appear in a pre-specified order (if they exist), so router can know up to which point it has to go to see for the useful information, which is also speeds up the process of header processing.
- By having options as Extension Headers, can be provided authentication, data integrity and optional data confidentiality, which is not possible in IPv4 with the help of Options alone.
- Removal of checksum field further increases the speed of header processing in this new protocol, because computation of header checksum is expensive at the intermediate system (router) which is a major redundancy in IPv4 because of the presence of both the checksum headers at both the link-layer and transport layer addresses.

2.4 Addressing Schemes in IPv6

As explained earlier, each host in the Internet needs to have an IP address to identify it uniquely. This concept of a unique IP address per host remains same in IPv6. A host can also have more than one IP addresses.

2.4.1 IPv6 Address Types

IPv6 addresses are 128-bit in length to accommodate the different needs of the present world , IPv6 has the following type addresses till now.

2.4.1.1. Unicast Address

This address is an identifier to a single host (node) (which is globally unique). A packet sent to a Unicast address is delivered to the node identified by that address. IPv6

Unicast addresses are aggregatable with contiguous bit-wise masks similar to IPv4 addresses under CIDR. The format of this type is shown in Figure 2.12

Figure 2.12: IPv6 Unicast Address

2.4.1.2. Anycast Address

This is an identifier for a set of nodes (typically belonging to different devices). A packet sent to an anycast address is delivered to one of the nodes identified by that address (the nearest one, according to the routing protocol's measure of distance). Anycast addresses are allocated from the Unicast address space, using any of the defined Unicast address formats. Thus, Anycast addresses are syntactically indistinguishable from Unicast addresses. When a Unicast [3] address is assigned to more than one node, thus turning it (that address) into an Anycast address. But the nodes, to which this address is assigned, must be explicitly configured to know that it is an Anycast address. There is some limitation in assigning Anycast address to any node, an Anycast address must not be used as the source address of an IPv6 packet. An Anycast address must not be assigned to an IPv6 host, that is, it may be assigned to an IPv6 router only. A router's Anycast address is predefined and its shown in Figure 2.13. The subnet prefix is an Anycast address and it is the prefix, which identifies a specific network. This Anycast address is syntactically same as the Unicast address, for a node on the network, with the node identifier set to zero.

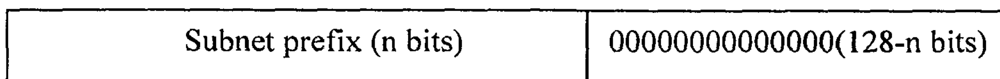


Figure 2.13: IPv6 Anycast Address

2.4.1.3 Multicast Address

This is an identifier for a group of nodes. A node may belong to any number of Multicast addresses. A packet sent to a Multicast address is delivered to all nodes identified by that address. The format of this address is shows in Figure 2.14.

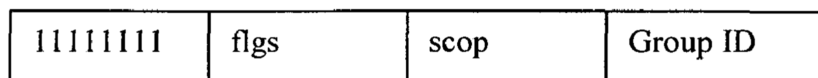


Figure 2.14: IPv6 Multicast Address

With the sequence 11111111, in Figure 2.14, at the start of the address, one can identify the address as a Multicast address. The high-order 3 bits are reserved, and must be initialized to 0. If T bit is not set it indicates a permanently-assigned Multicast address and if set, indicates a non-permanently-assigned Multicast address.

scop (4 bits) : This field is used to limit the scope of the Multicast group. Possible values of this field are shown in Table 2.4.

Group ID (Remaining bits): Identifies the Multicast group, either permanent or non-permanent (transient), within the given scope.

A source node should not use Multicast address as the source address of a packet. There are no broadcast addresses in IPv6, their function being superseded by Multicast addresses.

Value	Description
0,F	Reserved
1	Node-local scope
2	Link-local scope
3,4,6,7,9,A, B, C, D	(Unassigned)
5	Site-local scope
8	Organization-local scope
E	Global scope

Table 2.4: IPv6 Multicast scope field values

2.4.1.4 Unspecified Address

The address which has all 128 bits as zero is called as the Unspecified address. It must never be assigned to any node. It indicates the absence of an IP address. No node should use this address as the destination address of a packet.

2.4.1.5. Loopback Address

The address, which has all higher order 127 bits as zero and lower order 1 bit as 1, then that address is called as Loopback [3] address. This is a special type of Unicast address. A node, to send an IPv6 packet to itself, may use it. This address must not be used as the source address of a IPv6 packet that is sent outside of a single node. An IPv6 packet with a destination address of Loopback must never be sent outside of a single node and must never be forwarded by an IPv6 router. One can find out the type of an address by masking the leading bits in the address. Leading bits of the address is called the Format Prefix (FP), and depending upon the prefix bits, one can decide the address type. Table 2.4 specifies the FP and its address type. IPv6 is designed to be more efficient than the protocol IPv4, but IPv4 has already been established all over the world and is also widely used. So, it is not possible to go to this new protocol in a day or two. Hence, the new protocol has to co-exist with the protocol IPv4. To co-exist, IPv4 hosts and IPv6 hosts need to have some mechanism to communicate with each other. For this reason, two addresses have been defined to be compatible with the IPv4 protocol. Those two addresses are special types of Unicast addresses. They are discussed here.

a) IPv4-*Compatible IPv6 Address*: IPv6 includes a technique for hosts and routers to dynamically tunnel (encapsulating complete packet of an IPv6 in an IPv4 packet) packets over IPv4 routing infrastructure. IPv6 nodes that utilize this technique are assigned special IPv6 Unicast addresses that carry an IPv4 address in the low-order 32 bits. This type of address is termed an IPv4-Compatible IPv6 Address [9]. When the packet comes to the IPv4 router, it will take the least 32 bits of this address and use it as the address in the IPv4 header. At the other end of the IPv4 infrastructure, the router will again build this address by taking the address from the IPv4 header. A host having IPv6 protocol at network layer can use this address. The format of this address is shown in Figure 2.15

Allocation	Prefix (binary)	Fraction of Address space
Reserved	0000 0000	1/256
Unassigned	0000 0001	1/256
Reserved for NSAP Allocation	0000 001	1/128
Reserved for IPX Allocation	0000 010	1/128
Unassigned	0000 011	1/128
Unassigned	0000 1	1/32
Unassigned	0001	1/16
Aggregatable Global Unicast Addresses	001	1/8
Unassigned	010	1/8
Unassigned	011	1/8
Unassigned	100	1/8
Unassigned	101	1/8
Unassigned	110	1/8
Unassigned	1110	1/16
Unassigned	1111 0	1/32
Unassigned	1111 10	1/64
Unassigned	1111 110	1/128
Unassigned	1111 1110 0	1/512
Link-Local Unicast Addresses	1111 1110 10	1/1024
Site-Local Unicast Addresses	1111 1110 11	1/1024
Multicast Addresses	1111 1111	1/256

Table 2.5: IPv6 allocated addresses and their values

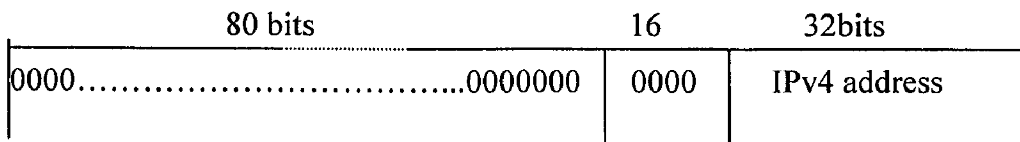


Figure 2.15: IPv4 -Compatible IPv6 address

b) *IPv4-Mapped IPv6 Address*: When a host which is in IPv4 domain want to communicate with a node in IPv6 domain, then the IPv4 host needs to have a special type of address to represent on IPv6 side. An address, in which the least 32 bits have the IPv4 host address, high order 80 bits have zeros, and remaining bits (16) have 1 as their value, is termed as an IPv4-Mapped IPv6 Address.

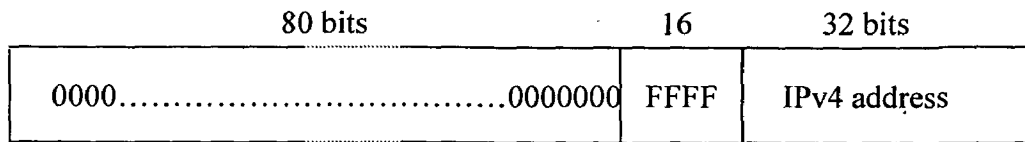


Figure 2.16: IPv4 -mapped IPv6 address

This type of address is used to represent the address of an IPv4-only node as an IPv6 address. The format of this address is shown in Figure 2.16.

2.5 IPv6 Header Format

In this section we describe header format of IPv6 and its fields. Figure 2.17 shows the format of an IPv6 header. The IPv6 header contains the following fields:

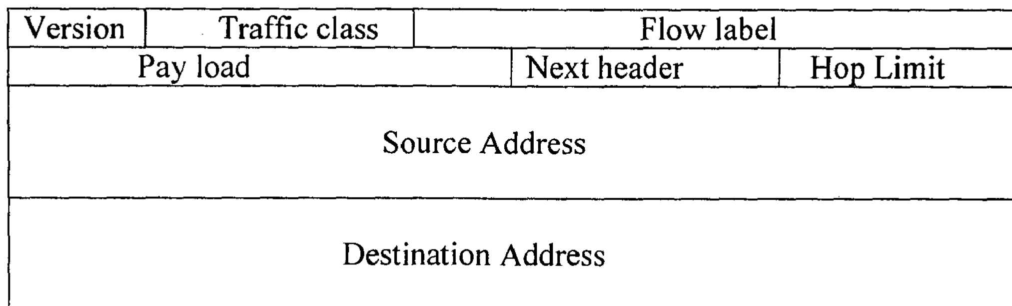


Figure 2.17: IPv6 header format

Version (4 bits): This field indicates the Internet Protocol version number, which is 6 in IPv6 header.

Traffic Class (8 bits): This field is equal to IPv4 header Type of Service field. This field provides information to the router about handling of this packet. This field contains the packet priority (high or low). Till now no specific priorities have been defined or allocated to this field

Flow Label (20 bits): This field may be used by a source to label sequences of packets for which it requests special handling by the IPv6 routers, such as non-default quality of service or real-time service. Hosts or routers that do not support the functions of the Flow

Label field are required to set the field to zero when originating a packet, pass the field on unchanged when forwarding a packet, and ignore the field when receiving a packet.

Payload Length (16 bits): This field specifies the length of the IPv6 packet in octets, which excludes IPv6 main header length (40 octets).

Next Header (8 bits): Before explaining this field, some notes about Extension Headers are in order, in an IPv4 header, if the packet source wants to send any options in the packet, it will use Options field, but this will increase the packet processing time at the router. To reduce the processing time of the packet at the router in IPv6, these Options are moved into a special area called as Extension Headers **[3]** .If a source wants to send options, it can use this area. So, a packet can have zero or more Extension Headers in the packet, the main header is a must. The Next Header field is useful in this case. To decrease the processing time of one header, this field will specify what type of data is there, after main header, whether it is an Extension Header or the header of a regular upper layer protocol, like TCP/ UDP etc. So, by looking at the current field the router will decide, whether it has to process the header further or not.

Hop Limit (8 bits): The maximum number of routers, the current packet can pass through. This is decremented by 1 by each node that forwards this packet. The packet is discarded if the field has value zero.

Source Address (128 bits), *Destination Address* (128 bits): These fields contain the IPv6 addresses of the source and destination hosts of the packet respectively.

2.5.1 IPv6 Extension Headers

As already explained the Options field in the IPv4 header is converted into Extension Headers in IPv6. As there are different types of options in IPv4, IPv6 also has different types of extension headers. In IPv6 each option will be specified as an Extension header. Each option has its own specific format but the general format of these Extension Headers is same. These Extension Headers will be placed in between the IPv6 main header and the upper-layer header in a packet (if any exist). Each such header is distinguished with the help of the Next Header field of the previous header. For example to know the following header of the IPv6 main header, we have to just see the

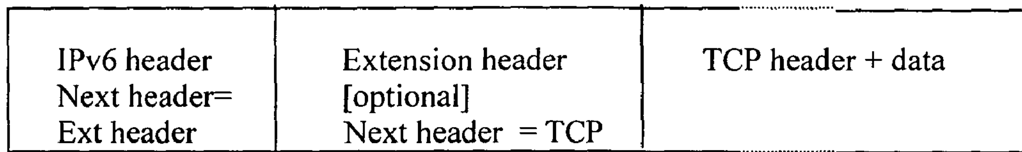


Figure 2.18: Example packet containing an IPv6 header as IP header

value of the Next Header field. Each IPv6 packet can have more than one Extension Header, but they should follow some defined rules, as discussed later in this section. Figure 2.18 shows a general IPv6 packet which having an IPv6 main header, one or more Extension Headers, and upper-layer headers. The contents of each extension header determine whether the router should examine the next header or not. To increase the processing speed of the packet, there is a pre-specified order for the Extension Headers in IPv6.

Extension Header Order in IPv6:

1. Hop-by-Hop Options Header (HbH)
2. Destination Options Header (DH)
3. Routing Header (RH)
4. Fragment Header (FH)
5. Authentication Header (AH)
6. Encapsulating Security Payload Header (ESPH)
7. Destination Options Header (DH)

Each Extension Header should occur at most once, except for the DH, which should occur at most twice (once before a RH and once before the upper-layer header). These headers may or may not occur in a packet, but if they occur they should follow the order specified above.

Every Extension Header has to follow a general format.

- o *Next Header* (8 bits): Specifies name of the header following this header.
- o *Hdr Ext Len* (8 bits): This field gives the length of this Extension Header in multiples of 64 bits (8 octets) excluding the first 64 bits. With this the router can

know the exact end point of this header and the start point of next (following) header.

- *Option Specific Data* (Variable length): This field contains header specific data. HbH and DH are currently using TLV encoded option format to specify options in their format.
- *Option Type* (8 bits): Identifier of the type of option.
- *Opt Data Len* (8 bits): This field indicates length of the Option Data field of this option, in octets. (In IPv4, TLV the length field value is excluding these two octets.)
- *Option Data* (Variable-length): This field contains data specific to the Option Type.

2.6 IPv6 Research

Commercially supported IPv6 software, including operating systems and popular applications is now available form vendors such as Cisco, Juniper, Microsoft etc. The IETF is developing IPv6 protocols for more than six years. The ngtrans-working [2] group developed many tools aimed at assisting IPv6's introduction, offering a "transitioning toolbox" for operators migrating to IPv6. More recently, v6ops' work is focusing on the operational aspects of introducing IPv6 services into IPv4 environments, and on deploying green field IPv6 networks (new networks that primarily use IPv6). In the EU's IST Fifth Framework Programme, exchange points. In 6NET, we're deploying an IPv6-only backbone to connect numerous national research and education networks (NRENs) at participating universities throughout Europe. All 6NET participants are potential IPv6 adopters, and they include all major European NRENs [35] and other universities with long-standing IPv6 interests. The 6NET project began in January 2002 and is running right now. 6NET researchers have more than 1,000 human-months of effort dedicated to:

- ✓ Network services such as IPv6 multicast, mobile IPv6, IP security protocol, quality of service, and domain name servers;
- ✓ IPv6 applications;
- ✓ Network deployment, monitoring, and management and

- ✓ IPv6 transition.

By offering IPv6 connectivity in existing academic environments (and beyond that, into the homes of staff and students), 6NET hopes to encourage the development of iunovative services and applications that are both mobile and secure, offering new solutions to old problems. There are many initiatives and projects within the Linux community contributing works on IPv6 whether it is an implementation, testing suites, or porting applications to work with IPv6. The below subsection provides a small survey of the major active projects in this area.

USAGI Project

The USAGI [7] project's mission is to deliver a production quality IPv6 implementation for Linux. The USAGI IPv6 stack for Linux relies heavily on the KAME code, which is the IPv6 stack on *BSD and most of the developers' efforts are in the direction on enhancing the stack and porting it to Linux. The current results of the project are an Open Source IPv6 implementation for the Linux kernel and an improved IPv6 API in the glibc library. The project has contributed patches and enhancements towards the Linux kernel and it seems that the current direction is to merge part of the USAGI IPv6 implementation into the Linux kernel.

TAHI Project

The TAHI [7,9] project has the objective of developing the verification technology for IPv6 through research and development of conformance and interoperability tests. The project has proven to be successful providing conformance tests, interoperability tests, and various test scenarios and test tools.

In this chapter the addressing schemes of IPv4 and IPv6 and the header formats of both of these have been specified and the need for migrating from IPv4 to IPv6 and the research done in this area has also been discussed.

Chapter 3

ANALYSIS

The proposed mechanism is implemented on a dual homed system that connects an IPv4 network and IPv6 network. It thus has atleast two network interfaces, one configured for IPv4 only connected to the IPv4 network, and one configured for IPv6 only connected to the IPv6 network. Packet translation is done whenever a packet received at the IPv4 interface has to be translated to the IPv6 network.. Many of the fields have a simple one-to-one mapping and translation is easy. However, there are several fields that require more complex handling. There are also certain subtle fields to be handled that are not clearly addressed in the RFC. Some of the interesting issues that had to be dealt in order to do packet translation have been discussed below.

3.1 Fragmentation Extension Header

Fragmentation is handled by a Fragmentation Extension Header in IPv6. An important distinction between IPv4 and IPv6 fragmentation is that in IPv6 fragmentation is done only at the source and is not allowed at any intermediate routers. In contrast, IPv4 packets can be fragmented at any intermediate router. Thus, in IPv6, path MTU discovery at the source is mandatory so that the source can fragment the packet if necessary. When an IPv4 packet comes to the translator, the translator tries to ensure that the resulting IPv6 packet will be carried properly. If the DF **[3,8]** (Don't Fragment) bit is set in the IPv4 packet header and the packet is not already a fragment, no fragmentation is done. If the DF bit is not set in the IPv4 packet header, two cases are possible. If the packet is already a fragment, the translator assumes that the IPv4 source has done a path MTU discovery, and simply adds an IPv6 fragmentation header and copies the fragment offset field form the IPv4 header. If the packet is not a fragment, the translator checks if the resulting IPv6 packet will be larger than 1280 bytes .If it is, the packet is fragmented before translation in appropriate chunks, so that the length of each packet, when converted to IPv6, will be less than 1280. These fragmented IPv4 packets are then translated as required.

3.2 Pseudo headers

The second issue is the handling of pseudo headers. Higher-level protocols such as TCP and UDP calculate a checksum in their headers that is based on the packet, as well as an imaginary header that contains, among other things the source and destination address. When an IPv4 packet is translated to an IPv6 packet and viceversa, the source and destination and hence the pseudo header used in the upper layer protocols also changes. This will necessitate recalculation of the TCP/UDP **[17]** check sum. The protocol translator checks the protocol field of the IP header and if the upper layer protocol is TCP/UDP, recreates the pseudo header with the new source and destination address after the packet translation .It then extracts the TCP/UDP header from the IP packet, recalculates the check sum in the header and puts the new TCP/UDP header back into the translated IP packet.

3.3 Translation of ICMP

The third issue that was faced is in the translation of ICMP messages. ICMP messages can be of two types. A query type ICMP message does not contain any message body (e.g. *Echo Reply)* An error type ICMP message, in addition to the ICMP header, contain a message body that includes the complete IP header and some parts of the body of the original IP packet for which this ICMP **[15]** packet is sent. An example of an error type ICMP message is *Destination Unreachable.* Translating query type ICMP message involves translating just the ICMP header. Translating error type ICMP messages includes translating the ICMP header and the hidden IP header in the body of the ICMP message. For error type ICMP messages, our system, after translating the ICMP header, extracts the IP header from the body of the ICMP message and translates it also. Length fields are adjusted accordingly after the translation.

3.4 Mapping

Another interesting issue that was faced is the mapping of addresses. When an IPv6 packet is converted into an IPv4 packet, the IPv6 source and destination addresses are converted into IPv4 addresses. Thus, an IPv4 address corresponding to the

destination's IPv6 address must be known to the translator. If the IPv6 address used is an IPv4-mapped IPv6 address, the IPv4 address is extracted from the IPv6 address itself. However, the destination can have any other IPv6 address assigned to it. The translator maintains a mapping table from IPv6 to IPv4 addresses and viceversa. The mapping table is dynamically built up using DNS as packets are being translated .If no mapping is found for a destination address during IPv6-to-IPv4 translation), the packet is simply dropped. The table built up can be persisted in a file and reloaded when the router boots. The table is also changed dynamically during the operation of the system as packets are translated.

The packet translation is done in the IP layer itself. The Linux kernel is modified and recompiled. The procedure uses a kernel resident mapping table to map IPv4 addresses to IPv6 addresses and viceversa during packet translation. An application level program creates the mapping. The program runs concurrently with packet translation. The program spawns four threads. Two of these threads are listener threads that capture every IPv4 and IPv6 packet respectively that go through the router. The other two are IPv4 and IPv6 handler threads that process captured IPv4 and IPv6 packets respectively. The program captures every packet, finds the one that potentially may be translated by the packet translator and if yes, checks if an address map exists for the addresses in the packet. If no such address map exists, querying DNS **[16]** creates one, the new mapping is returned to a persistent file and also pushed down immediately to the kernel address-mapping table that the packet translator uses so that subsequent packets with the same addresses can be translated. If the mapping already exists, no action is taken, as the mapping will already exist in the kernel-mapping table. Note that all packets are also received by the IP layer and processed normally in parallel with this mapping program. Thus any packet that can be translated using the kernel-mapping table will be translated by the packet translator normally. This program checks each packet in parallel and updates the kernel-mapping table if needed. The first time a mapping is created for an address, and may be for a few packets after that, the packet may have to be resent since the packet translator may have already dropped it, since no mapping address was found. Our system employs a timestamp-based scheme to decide if a packet needs to be resent or not.

The IPv4 listener thread opens a socket using socket (PF_packet, SOCK_DGRAM, htons (ETH_P_IP)) call. Specifying SOCK_DGRAM instead of SOCK_RAW strips the Ethernet header off the packet, delivering only the IPv4 packet through the socket. The captured packet is put into a queue along with a timestamp with the time the packet was captured. Similarly the IPv6 listener thread captures all IPv6 packets (using ETH_P_IPv6 in the socket call) and puts them in a queue with the timestamp. The IPv4 and IPv6 listener threads signal the IPv4 and IPv6 handler threads respectively.

The IPv4 handler first decides if the packet is one that can potentially be translated by the packet translator in the IP layer, if not there is no need to look for any address mapping since no address translation will be done for this packet. The handler decides this by matching the source subnet to the subnet connected to the v4 interface. If the packet may be translated the handler checks if the mapping exists in the mapping file. For the source and destination addresses in the packet if a mapping exists no action is taken, if a mapping is not found, the handler sends a DNS query for a Reverse lookup of the address. The name returned is then used to make a DNS query for an AAAA record (DNS record type for IPv4 adddresses). This query will return the IPv6 address for the host. The mapping file is updated and the new mapping is immediately pushed down to the kernel mapping using *ioctl* call .The mapping is also given a timestamp to indicate when the mapping was first obtained for any packet, if the packet capture timestamp is less than the mapping timestamp foe either the source or the destination address, the packet may have been potentially dropped at the IP layer already. This is taken care of by resending the packets, the processing of the IPv6 packets by the IPv6 handler thread is similar.

The kernel-resident mapping table is kept as a red-black tree with IPv4 addresses as keys and IPv6 addresses as values. Hence the mapping is found in $O(logn)$ time for IPv4 to IPv6 mapping and in $O(n)$ time for IPv6 to IPv4 mapping, where n is the number of entries in the mapping table.

Chapter 4

DESIGN

IPv6 has been designed to overcome some of the shortcomings of IPv4. However, IPv4 is very widely used and it is impossible to completely switch to IPv6 immediately. It is therefore expected that IPv4 and IPv6 will co-exist for quite some time to come. Hence, an IPv4 host may need to communicate with an IPv6 host and vice-versa. This will necessitate translating an IPv6 packet to an IPv4 packet and vice-versa at network boundaries. This Chapter describes the protocol translation process.

4.1 IPv4 to IPv6 header translation

In this section we describe how an IPv4 header is converted to an IPv6 header. The two headers (IPv4 and IPv6) are shown here again for the sake of easy reference. We explain how each field of an IPv4 header is converted to a field of an IPv6 header. Formats of IPv4 header and IPv6 header are shown in Figures 4.1 and 4.2 respectively.

Version (4 bits): This field is retained in IPv6 and it will have the value of 6 instead of 4.

Version	IHL	Type of Service	Total Length	
Identification			Flags	Fragment Offset
Time to Live		Protocol	Header Check sum	
Source Address				
Destination Address				
Options				Padding

Figure 4.1: IPv4 header format

- *IHL* (4 bits): IPv6 header has fixed length of header as 40 octets, so there is no need to have a field to specify the length of the header. So this field is not present in the IPv6 header and is dropped.
- *Type of Service* (8 bits): In IPv6 Header there is a field called Traffic Class, which has an use similar to this field. But in Traffic Class field it does not have support

for D, T, R bits, so the high order 3 bits of Type of Service field is copied to the lower 3 bits of the Traffic Class field.

<table>
<tr><td>Version</td><td>IHL</td><td>Type of Service</td><td colspan="3">Total Length</td></tr>
<tr><td colspan="3">Identification</td><td>Flags</td><td colspan="2">Fragment Offset</td></tr>
<tr><td colspan="2">Time to Live</td><td>Protocol</td><td colspan="3">Header Check sum</td></tr>
<tr><td colspan="6">Source Address</td></tr>
<tr><td colspan="6">Destination Address</td></tr>
<tr><td colspan="5">Options</td><td>Padding</td></tr>
</table>

Figure 4.2: IPv6 header format

Total Length (16 bits) : The Payload Length field of the IPv6 main header is equivalent to this field. However, this field includes the length of the IPv4 header, where as the Payload Length field does not include IPv6 header length. This field value is first decremented by the length of IPv4 header (which is IHL * 4), and this new value is copied to the Payload Length field of the IPv6 main header.

Identification (16 bits), Flags (3 bits), Fragment Offset (13 bits): These fields are useful in carrying fragmentation information. In IPv6 this information is carried as Fragmentation header, which is one of the Extension Headers [30]. These fields are used in the construction of a Fragmentation header in the manner specified below. Addition of FH will increase the packet length by 8, hence the value 8 is added to the Payload Length field of the IPv6 header.

The Fragmentation header is constructed as follows:

- *Next Header* (8 bits): Copy the value of the field Protocol in IPv4 header to this field. This field may change if we add ESPH or AH, but we are not dealing with these headers in the translation.
- *Reserved* (8 bits): This field is reserved for future use, so these are set to zero.

- *Fragment Offset* (13 bits): Copy the value of the field Fragment Offset in IPv4 header to this field.
- *Res* (2 bits): These bits are reserved, so these are set to zero.
- *M* (1 bit): Copy the MF (more fragment) bit of the Flags field in the IPv4 header to this bit.
- *Identification* (32 bits): Copy the value of the field Identification in the IPv4 header, to this field lower order 16 bits.

Time to Live (8 bits): Hop Limit is the field in IPv6 corresponding to this field. So, this field is copied to the Hop Limit field in the IPv6 header.

Protocol (8 bits): If this packet is a fragmented [32] packet (which can be found by looking at the Fragment Offset field and Flags field of the IPv4 header), this field value is copied to FH's Next Header field, otherwise this field is copied to the Next Header field of the IPv6 main header.

Header Checksum (16 bits): Computation of this field value will increase the header processing time, so this field has been dropped from IPv6 header. So this field is ignored.

Source Address (32 bits): This field is the 32-bit address of the source host. But in IPv6 header, the corresponding field is 128 bits long. This requires some mechanism to map an IPv4 address to an IPv6 address. This mechanism has been implemented in the form of a file. The file will contain addresses of IPv4 and IPv6, and there will be a mapping function, which will return an IPv6 address for an IPv4 address. Using this function and file we will get an IPv6 address and we use this address in setting up the value of the field Source Address in IPv6 main header.

Destination Address (32 bits): The same thing is done as explained for the conversion of the IPv4 Source Address field value to the IPv6 Source Address field value.

Options (Variable length): This field contains TLV [29] encoded options. Each such option is converted to an Extension Header in IPv6 if an equivalent one exists. An option is identified by seeing the value of the field Option Type in the TLV form. The conversion of each IPv4 option based on the Option Type field is described below.

1.*End of Option List* (Option Type - 0): This option indicates the end of all options in the IPv4 header. This option is simply ignored as no action is necessary for it.

2.*No Operation* (Option Type - 1): This option is used in between two options of the IPv4 header for the purpose of padding. IPv6 has its own padding options, so this option also ignored.

3. *Security* (Option Type - 130): There is no Extension Header in IPv6 corresponding to this option. This option is also ignored.

4. *Record Route* (RR) (Option Type - 7): There is no Extension Header in IPv6 corresponding to this option. This option is also ignored.

5.*Stream Identifier* (Option Type - 136): There is no Extension Header in IPv6 corresponding to this option. This option is also ignored.

6. *Internet Timestamp* (Option Type - 68): There is no Extension Header in IPv6 corresponding to this option. This option is also ignored.

7. *Loose Source and Record Route* (LSSR)(Option Type - 131), *Strict Source and Record Route* (SSRR) (Option Type - 137): Type zero Routing header **[3, 14]** is the Extension Header in IPv6 corresponding to this option. The different fields of the Type zero Routing header are filled from LSSR or SSRR option of IPv4 header in the following manner.

- *Next Header* (8 bits): If this packet is a fragmented packet, then there is already a FH, so this field value is set to 44 otherwise the value of the field Protocol in the IPv4 header is copied.
- *Hdr Ext Len* (8 bits): This is the length of the Type zero Routing header. This field value is set to the value obtained by multiplying 2 to the number of addresses in this option (LSSR or SSRR).
- *Routing Type* (8 bits): This field value is set to zero, indicating that this header is Type zero Routing header.
- *Segments Left* (8 bits): This is computed from the Pointer field of the option (LSSR or SSRR). The expression to compute the value of this field is (no. of addresses in (LSSR or SSRR)) - (value of Pointer field) / 4.

- *Type-specific data* (Variable-length): This area contains IPv6 addresses through which this packet has to be routed. Each IPv4 address in the original option is converted to an IPv6 address using the mechanism described earlier.

By adding Type zero Routing header to the IPv6 header, it increases length of the packet. So added the value, 16 multiplied by number addresses in the LSSR or SSRR option (which is computed as Option Length field value decremented by 3 and then divided it by 4) and finally add 16, to the Payload Length field of the IPv6 header. The value of the field Next Header in the IPv6 header is set to 43.

8. *Router Alert* (Option Type - 148): Hob-by-Hop Options header's Router Alert TLV encoded option of the IPv6 header is the equivalent one to this option. To add this option to the IPv6 header we have to add HbH **[9,14]** , which will increment the IPv6 packet length. So we have to adjust the length field of the IPv6 header. For that add 8 to the value of the field Payload Length in the IPv6 main header. Also the Next Header field of the IPv6 header is set to zero.

The *Next Header* field of HbH is set to 43 if there is a Routing Header, else to 43 if there is a FH, else copied the value of Protocol field of the IPv4 header. The value of Hdr Ext Len field of HbH is set to zero and the HbH is filled with the Router Alert option. The Option Type field of Router Alert is set to 5, the Option Data Len field to 0 and the Value field to 2. This completes the construction of HbH and Router Alert option in the header.

IPv6 header's *Flow Label* (20 bits) field is a new field and the value of this field is set to zero when translating.

4.2 IPv6 to IPv4 header translation

In this section details about how each field of IPv6 header is converted to corresponding IPv4 header fields are given.

Version (4 bits): This is equivalent to the Version field in the IPv4 header. The value of the Version field of the IPv4 header is set to 4.

Traffic Class (8 bits): The value of the low-order 3 bits in this field is copied to the higher order 3 bits of the field Type of Service in the IPv4 header.
Flow Label (20 bits): This is a new field in IPv6 main header. So, this field is ignored.
Payload Length (16 bits): IPv4 header's Total Length field is the equivalent one to this field. This field value is copied by subtracting the Extension Header length (if any exists in the IPv6 packet), and adding 20 to the Total Length field of IPv4 header. Total Length field value [9] in the IPv4 header will be changed again if there are any Extension Headers in the IPv6 header. Since this will necessitate adding options to the IPv4 header.
Next Header (8 bits): The Protocol field of the IPv4 header is the equivalent one to this field. The field value is copied to the Protocol field of IPv4 header. If there are any Extension Headers, then this value will be changed and set to the value copied from the final Extension header's Next Header field.
Hop Limit (8 bits): The Hop Limit field of the IPv4 header is the equivalent one to this field. So, this field value is copied to the Hop Limit field of the IPv4 header.
Source Address (128 bits): A lookup is done in the mapping le and the corresponding IPv4 address is copied to the Source Address field of the IPv4 header.
Destination Address (128 bits): A lookup is done in the mapping file and the corresponding IPv4 address copied to the Destination Address field of the IPv4 header.

The *Extension Headers* in the IPv6 packet are equivalent to the Options of the IPv4 packet. The conversion of Extension Headers to Options is explained below for each extension header.

1.*Hop-by-Hop Options Header* (HbH): This header is identified by seeing the Next Header field value of the IPv6 main header as zero. Translation of this header to IPv4 header option depends on the options of this header (TLV encoded options). This header can have more than one TLV [8] encoded options. These possible options are given below and their corresponding options in IPv4 are shown.
(a) *Jumbo Payload*: This option is used in IPv6 header to send a packet larger then 64K Bytes. But all NICs cannot handle packets larger than 64K Bytes. So, this option is ignored in IPv4.

(b) *Router Alert*: This option is converted to a Router Alert option in the IPv4 header. To add this option to the IPv4 header's Options field, add 4 octets in length at the end of IPv4 main header. The octets added are useful in filling the Router Alert option [5,9] in the IPv4 header. Field values of Router Alert option are filled as discussed below. The value of the Option Type field is set to 148, the value of the Option Data Len field is set to 4, and the value of the Value field is set to zero. By adding this option to the IPv4, header length will increase the length. For that add 4 to the value of the field Total Length in the IPv4 header. Also add 1 to the value of the field IHL in the IPv4 header, because length is incremented in IPv4 header by adding the Router Alert option.

2. *Destination Options Header* (DH): This header will not occur in IPv6 packet because till now its options are not defined by the Internet community.

3. *Routing Header* (RH): LSSR is the option in IPv4 corresponding to this extension header. The construction of the LSSR option fields from this option is shown below.

- *Option Type* (8 bits): This field value is set to 131.
- *Option Data Len* (8 bits): This field value is computed from RH of IPv6 by dividing the Option Data Len by 2, then multiplying by 4 and finally adding 3 to this value ((length of the RH / 2) * 4 + 3).
- *pointer* (8 bits) : This field value is computed from the Segments Left field of RH. The value is set to (length of the RH - Segments Left)* 4.After constructing this option the Option Data Len (of IPv4 LSSR option) value is added to the IPv4 header Total Length field. The value of the Option Data Len field divided by 4 is added to the IHL field of the IPv4 header.

4. *Fragment Header* (FH): In IPv4, the header fields carry fragmentation information itself. There is no need to have any extra option. Lower order 16 bits of the Identification field value of the header is copied to the IPv4 header's Identification field. The Fragment Offset field value of this header is copied to the IPv4 header's Fragment Offset field. The M bit value of this header is copied to the IPv4 header MF bit. The DF of the IPv4 header is set to zero. As these three fields (id, offset and flags are already part of IPv4 main header, so no need to adjust any length field.

5. *Authentication Header* (AH): This header provides integrity of the data sent from a source host to a destination host. Router cannot intercept a packet having this option, so this packet is discarded.

6. *Encapsulating Security Payload Header* (ESPH): As this header provides security between source and destination hosts data transmission, router can not intercept a packet having this option, so this packet is discarded.

If the IPv6 packet does not contain a FH then the Identification, Fragment Offset and fields of the IPv4 header is set to zero, else leave them as it is.

The IPv4 header's Protocol field value is set depending upon the IPv6 packet main header and Extension Headers. The value of the Next Header field of the IPv6 main header is copied to the IPv4 header's Protocol field. If a FH is present in IPv6 packet then fragment header's Next Header field [17] value is set to the IPv4 header's Protocol field, else if a RH is present in IPv6 packet then Routing header's Next Header field value is set to the IPv4 header's Protocol field, else if a HbH is present in IPv6 packet then Hop-by-Hop Options header's Next Header field value is set to the IPv4 header's Protocol field.

4.3 Upper Layer protocol checksum calculation

Transport Layer of TCP/IP Protocol suite will include a pseudo header in the calculation of the packet checksum. This pseudo header contains some of the fields of the network layer header. So, the checksum of the upper layer protocol header has to be recalculated. When converting from IPv4 to IPv6 header, the IPv6 header Source and Destination addresses is used in constructing the pseudo header. Formats of IPv4 and IPv6 header pseudo headers are shown in Figures 4.3 and 4.4 respectively.

<table>
<tr><td colspan="3">Source Address (16 octets)</td></tr>
<tr><td colspan="3">Destination Address (16 octets)</td></tr>
<tr><td>Next Header</td><td>Protocol</td><td>Length(2 octets)</td></tr>
</table>

Figure 4.3: IPv6 Pseudo header format

When converting from IPv4 to IPv6, fill the fields of the IPv6 pseudo header and use this header in recalculation of the Checksum field of either TCP or UDP header. Do the same thing when converting from IPv6 to IPv4.The value of the Protocol field in these headers is set to 6 if the upper layer protocol is TCP, and set to 17 if the upper layer protocol is UDP. The value of the Length field in these headers is set to the packet length up to the upper layer header.

Source Address (4 octets)		
Destination Address (4 octets)		
Next Header	Protocol	Length (2 octets)

Figure 4.4: IPv4 Pseudo header format

4.4 Translating ICMP

The purpose of the Internet Control Message Protocol (ICMP) messages is to provide feedback about problems in the communication (data transmission) environment. ICMP is considered to be an integral part of IP (Internet protocol), although architecturally it is layered upon IP. ICMP **[15]** provides error reporting, control and first-hop router (which connects two or more networks) redirection. Some of the situations in which ICMP messages are used are when a packet cannot reach its destination, when the router does not have the buffering capacity to forward a packet, and when the router can direct the host to send traffic on a shorter route. In this section, first the basic need for this protocol is explained. Then the header format of this protocol is explained. Since ICMP is an integral part of IP, there are two versions of ICMP, ICMPv4 and ICMPv6. Both ICMPv4 and ICMPv6 message formats have been explained. Finally it is explained how an ICMPv4 packet is converted into an ICMPv6 packet and vice-versa during transmission between an IPv4 and IPv6 host.

There are two types of ICMP messages, one for reporting errors and the other for querying for information. A query type is used to query a host to get information about the host and error types are used to send transmission errors to the source host of the original packet. When sending an error type message, the intermediate node (can be router or final destination) includes the original packet's complete IP header and 8 bytes of the upper layer protocol in case of an IPv4 packet. For an IPv6 packet the intermediate host fills the ICMP [14] message with the original packet until the length of this packet reaches PMTU. A query type message will not include anything but the type of the query indicating that it is a query ICMP type message, and also includes code for the subtype in this query type message. By seeing the type of the query, the destination host will send the appropriate information required to the source host.

4.4.1 ICMP version 4 (ICMPv4)

In this Section the format of an ICMPv4 packet is explained. The general format of an ICMPv4 packet is shown in Figure below

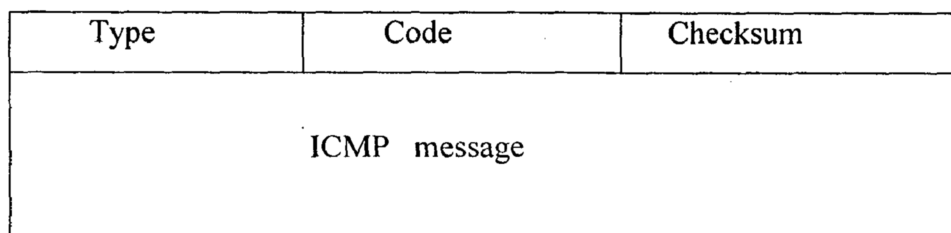


Figure 4.5: ICMPv4 general header format

- *Type* (8 bits): This field specifies the format of the ICMP message, i.e., whether the message is a query or error type.
- Code (8 bits): This field specifies the sub type of the Type.
- Checksum (16 bits): This field contains the checksum of the packet till this header excluding the other lower level headers. This checksum includes pseudo header, if any.

- ICMP message (Variable length): This is a variable length field and the contents of this field depend upon the type of the message (whether query or error).

The table below shows the ICMPv4 message types

Type	Description.
0	Echo reply
3	Destination unreachable
4	Source quench.
5	Redirect
6	Alternate Host Address
8	Echo request
9	Router advertisement
10	Router solicitation
11	Time exceeded
12	Parameter problem
13	Timestamp request
14	Timestamp reply.
15	Information request
16	Information reply
17	Address mask request
18	Address mask reply
19	Reserved (for security).
20-29	Reserved (for robustness experiment)
30	Traceroute.
31	Conversion error
32	Mobile Host Redirect
33	IPv6 Where-Are-You
34	IPv6 I-Am-Here
35	Mobile Registration Request
36	Mobile Registration Reply.
37	Domain Name request
38	Domain Name reply
39	SKIP Algorithm Discovery Protocol
40	Photuris, Security failures.
1,2,7,41-255	Reserved

Table 4.1: ICMPv4 message types

4.4.2 ICMP version 6 (ICMPv6)

ICMPv4 messages are used in the correspondence of IPv4 packets. In the same way ICMPv6 messages are used to report any errors regarding the processing of IPv6 packets

and to query for information about a host supporting IPv6. ICMPv6 is an integral part of IPv6 and must be fully implemented by every IPv6 protocol supporting host.

Every ICMPv6 [15] message is preceded by an IPv6 Header and zero or more IPv6 Extension Headers. The ICMPv6 header is identified by a Next Header field value of 58 in the immediately preceding header. The general format of an ICMPv6 message is shown in Figure 4.6.

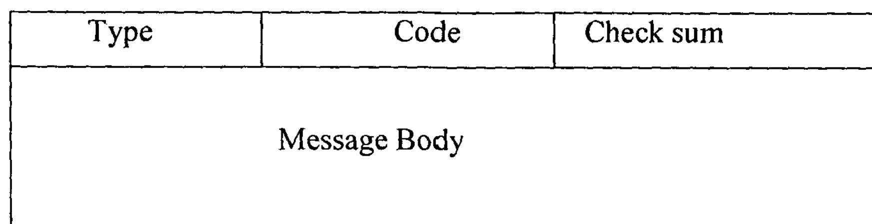


Figure 4.6: ICMPv6 Message general format

- *Type* (8 bits): This field indicates the type of the message. Its value determines the format of the remaining data.
- *Code* (8 bits): This field value depends on the message type. It is used to create an additional level of message granularity.
- *Checksum* (16 bits): This field is used to detect data corruption in the ICMPv6 message and parts of the IPv6 header.
- *Message Body* (Variable length): This field contains the data specific to the Type field value of the message.

There are two types of ICMPv6 messages, error type and query type. Query type messages can also be called as informational messages. Error messages are identified as such by having a zero in the high-order bit of their message Type field values. Thus, error messages have message Types from 0 to 127 and informational messages have message Types from 128 to 255

- *Unused* (32 bits) : This field is used by some Types of message to specify that message type specific data. If any message is not using this field should set this field value to zero.

Remaining part of the message contain the original IPv6 packet (as much as will fit in to this ICMPv6 message without exceeding the minimum IPv6 PMTU) for which this ICMPv6 packet is being sent. The figure below shows ICMPv6 message types

Type	Description
1	Destination unreachable.
2	Packet too big
3	Time exceeded
4	Parameter problem
128	Echo request
129	Echo reply
130	Group Membership Query.
131	Group Membership Report
132	Group Membership Reduction
133	Router Solicitation
134	Router Advertisement
135	Neighbor Solicitation
136	Neighbor Advertisement
137	Redirect
138	Router Renumbering
139	ICMP Node Information Query
140	ICMP Node Information Response
141	Inverse Neighbor Discovery Solicitation Message
142	Inverse Neighbor Discovery Advertisement Message
5-127,143-255	Reserved.

Table 4.2 ICMPv6 message types

4.4.3 ICMPv4 to ICMPv6 translation

Translating query type message is straightforward, it involves converting an ICMPv4 header into ICMPv6 header. But error type ICMP [31] message will also contain an IP header, which has to be converted also. Figure 4.7 depicts the idea of translating error type ICMPv4 messages to corresponding ICMPv6 messages.

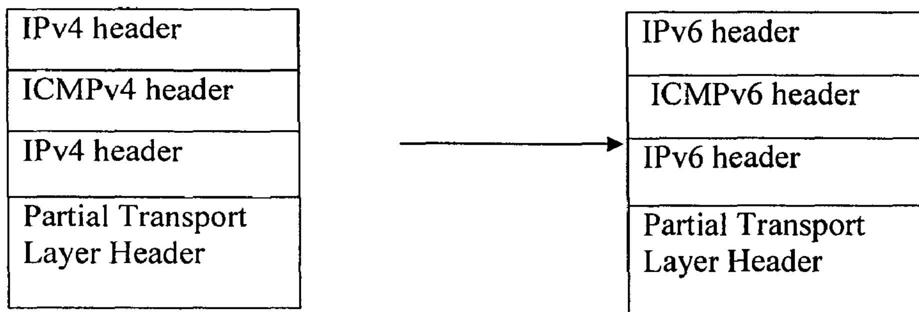


Figure 4.7: IPv4-to-IPv6 ICMP error message translation

The translation of a query type ICMPv4 header to the corresponding ICMPv6 query type header. is discussed first and later the error type is discussed.

4.4.3.1 Query Type translation

Table 4.3 describes how we can translate one header to another type of header. For example if we have Echo Request ICMPv4 type message, then pull off this header from the original packet then add the new ICMPv6 header (or as the two headers occupy the same number of bytes, change the field values) and change the fields accordingly, such as set the value of the Type field to 128 and do not change the value of the code field, which is zero. Other types (9, 10, 13, 14, 15, 16, 17, 18, and unknown) are ignored silently.

4.4.3.2 Error type translation

Error type ICMPv4 messages contain not only the ICMPv4 header but also the original packet IPv4 header and 64 bits of data in IPv4 message body. So not only the ICMPv4 header has to be translated but also the inner IPv4 header also has to be taken

into consideration. To translate the inner IPv4 header to IPv6 header follow the same procedure as described strip off the IPv4 header by copying it to a buffer and then add an IPv6. header to this header and the conversion is done as explained. Table 4.4 gives the conversion from ICMPv4 error type messages to ICMPv6 error type messages.

IPv4		IPv6		Description
Type	Code	Type	Code	
8	0	128	0	Echo request
0	0	129	0	Echo Reply

Table 4.3: ICMPv4 to ICMPv6 query type translation

IPv4		IPv6		Remarks
Type	Code	Type	Code	
3	0	1	0	Destination Unreachable
3	1	1	0	
3	2	4	1	Update pointer as 6,means pointing to next header filed in the error packet header
3	3	1	4	
3	4	2	0	Adjust MTU as that of the device which has received this packet
3	5	1	0	
3	6	1	0	
3	7	1	0	
3	8	1	0	
3	9	1	1	
3	10	1	1	
3	11	1	0	
3	12	1	0	
11	0	3	0	Time exceeded
12	0	4	0	Update the pointer as shown below from given pointer to the specified pointer 0-to-4, 8-to-7, 9-to-6, 12-to-8,16-to-24 for all others set -1

Table 4.4: ICMPv4 to ICMPv6 Error Reporting type translation

First strip of the ICMPv4 header then add ICMPv6 header, then set the values of the fields accordingly. Codes other than in Table 4.4 will be ignored.

4.4.4 ICMPv6 to ICMPv4 translation

ICMPv6 messages are also of two types, query type and information type. Converting information type ICMPv6 messages to ICMPv4 query type messages is straightforward by changing the corresponding fields in the ICMPv4 header.

Converting ICMPv6 error type messages to ICMPv4 [17] error type messages requires to conversion of the IPv6 header, which is in the body of the ICMPv6 message. In case of query type message, we do not have the inner IPv6 header, so we convert the outer IPv6 and ICMPv6 headers into corresponding headers in IPv4 packet. But, when converting from error type ICMPv6 message, convert the inner IPv6 header first then the ICMPv6 header, and then the outer IPv6 header.

4.4.4.1 Conversion of ICMPv6 Query Type Message

Conversion from ICMPv6 message to ICMPv4 message requires just changing the corresponding fields in the ICMPv4 header of ICMPv4 message. Table 4.5 gives the conversion from one type to another. As an example, if the ICMPv6 message header contains the Type field

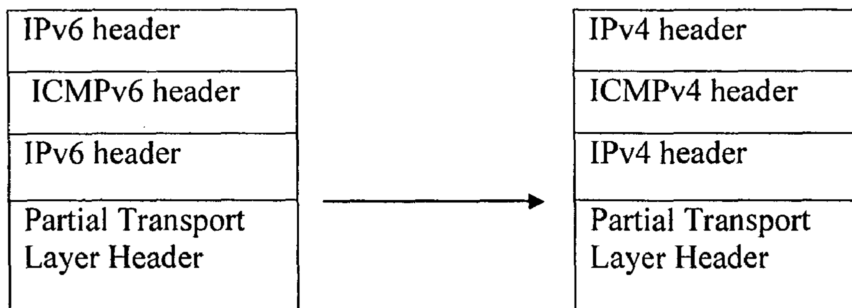


Figure 4.8 :IPv6-to-IPv4 ICMP error message type translation

128, which is Echo Request ICMPv6 message, it will be converted to 8, which is Echo Request ICMPv4 message. Other types (130, 131, 132, 133 to 137 and unknown query types in ICMPv6 message header) will be ignored.

4.4.4.2 Conversion of ICMPv6 Error Type Message

To convert ICMPv6 error type message to ICMPv4 error type message first of all we need to convert the inner IPv6 header of ICMPv6 message to the inner IPv4 header of the ICMPv4 message, and then convert the remaining part. We use the same procedure as we used to convert the IPv6 main header. For converting the ICMPv6 message to ICMPv4 message we use the mapping shown in Table 4.5 Other types (other than in Table 4.5) will be ignored.

IPv6		IPv4		Description
Type	Code	Type	Code	
128	0	8	0	Echo request,just change type and compute check sum
129	0	0	0	Echo Replyrequest,just change type and compute check sum

Table 4.5: ICMPv6 to ICMPv4 Query type translation

4.4.5 ICMP checksum calculation

ICMP header (either version 4 or 6) has a Checksum field and its checksum should be calculated by adding a pseudo-header The same IPv6 IPv4 rule is applied here to compute the Checksum field value. ICMPv6 message uses the pseudo-header having IPv6 addresses and ICMPv4 uses the pseudo-header containing IPv4 addresses. Including the pseudo-header in checksum calculation is not mandatory in case of ICMPv4 messages. But in case of ICMPv6 messages, it is mandatory and every host has to include the pseudo-header when it is calculating the checksum. In the pseudo-header, there is a

```
struct sk_buff {

struct sk_buff * next;    /* Next buffer in list */
struct sk_buff * prev     /* Previous buffer in list */
struct sk_buff_head * list; /* List we are on */
struct sock *sk; /* Socket we are owned by */
struct timeval stamp; /* Time we arrived */
struct net_device *dev; /* Device we arrived on/are leaving
                           by */

            }
```

Before storing this packet, the NIC routine will strip the physical level header (Ethernet header or Token Rink header or FDDI header etc.). It then calls the generic packet handling function *netif_ rx(),* which is in *net/core/dev.c file* . This function queues this packet in a packet list structure of the current processor. Each processor will have a separate list for packets; the processor, which handled the NIC interrupt request, will get this packet in its packet list. The *netif_rx( )* function returns one of the following values which are defined in *include/linux/netdevice.h.*

```
#define NET_RX_SUCCESS 0 /* keep them coming*/
#define NET_RX_DROP 1 /* packet dropped */
#define NET_RX_CN_LOW 2 /* storm alert, just in case */
#define NET_RX_CN_MOD 3 /* Storm on its way! */
#define NET_RX_CN_HIGH 4 /* The storm is here */
#define NET_RX_BAD 5 /* packet dropped due to kernel error
                      */
```

After queuing the packet this function raises a software interrupt request NET_IRQ _SOFTIRQ. After receiving software interrupt request the processor will call the appropriate function to handle this interrupt. Each software interrupt is attached with a function at the time of loading the packet handler module. The function, net_rx_action( ), is attached to the software interrupt request NET_ IRQ_ SOFTIRQ **[20]** at the time of driver loading.

This *net_rx_action( )* function is defined in *net/core/dev.c file*. This function checks whether the packet belongs to any one of the network layer protocols registered in this host (*sk_buff* structure will have a field called protocol, this field contains the type of this packet). If the packet belongs to any one of the protocols registered at the network

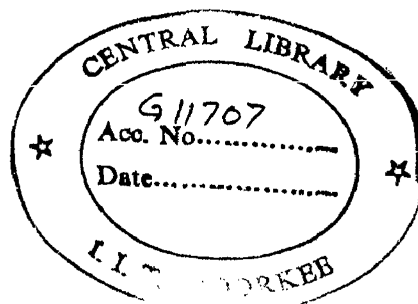
layer, then that particular protocol handling function will be called. For example if the packet belongs to IPv4 the function *ip_rcv()* will be called, else if it belongs to IPv6 then the function *ipv6_rcv( )* will be called. The function *ip_rcv( )* is defined in *net/ipv4/ip input.c* file. From this function *ip_input_finish( )* function will be called; in this function it will be decided whether this packet is actually for this host or for another host (because if this host is a router then it needs to process and forward the packet). By seeing the Destination Address field of the IP header, this function will decide whether this is for this host or not. If it is for the this host then that packet will be delivered first to the function *ip_local_deliver( )* and there to *ip_local_deliver_finish( )*. All these functions are defined in the same file, *net/ipv4/ip_ input.c*. In *ip_local_deliver_finish( )* function, it will be decided to which upper layer protocol (TCP/ UDP/ etc.) this packet has to be delivered. If this packet upper layer protocol is one of the registered upper layer protocols in this host then this function strips the IPv4 header and hands over the packet to the upper layer protocols. From the upper layer protocols this packet data goes to the application. When the upper layer protocol of packet is TCP then the function *tcp_v4_rcv( )* is called, else if UDP then the function *udp_rcv( )* will be called etc. If the packet upper layer protocol is not one of the upper layer protocols registered in this host, then the function *ip_ local_ deliver_finish( )* will send a destination unreachable (type 3) an ICMPv4 error packet having the Code field value 2 (protocol unreachable), to the source host of the original packet.

If the packet is not for this host and the host does not have the capability of forwarding a packet, then the packet will be discarded. If the host has the capability of forwarding a packet then the function *ip_rcv_finish( )* will call the function *ip_forward( )* which is in the file *net/ipv4/ip_forward.c*. This function calls the function *ip_forward_finish( ),* which is in the same file, from there it goes to *ip_send( )* function, where the packet length is checked. If the packet length is more than PMTU then the packet is fragmented. After fragmenting the packet is sent to the function *ip_ output _finish( )* function, which is defined in *net/ipv4/ip_output.c* file. If the packet length is not more than PMTU then the packet is handed over to the same function (*ip_output_ finish( )*) for further processing. This function calls the function *ip_finish_output2( ),*

which is defined in the same file. Finally this function calls the appropriate physical layer function to add that layer's header and to transmit the packet in the network.

If the packet is of type IPv6 then the function ipv6_rcv( ) will be called which is defined in *net/ipv6/ip6_input.c* file. From this the packet goes to the function ip6_rcv_finish( ), this function will decide whether the packet is for this host or for other host. If this host has forwarding capability then it forwards this packet otherwise discards it. If the packet is for this host then the function *ip6_input( )* will be called and from this function the packet goes to the function *ip6_input_finish( )* both these functions are defined in the file *net/ipv6/ip6_input.c*. In these functions, all the Extension Headers will be processed and the function *ip6_input_finish( )* will know what is the upper layer protocol. This function will check the packet's upper layer protocol with the registered upper layer protocols within this host. If it suits to any one of the upper layer protocols then this function strips the IPv6 header and calls the function depending upon the protocol. If the upper layer protocol is TCP it calls the function *tcp_v6_rcv( )* else if UDP then the function *udpv6_rcv( )* will be called etc. If the packet's upper layer protocol is not one of the upper layer protocols then this function sends an ICMPv6 error message **[15]**, by setting the Type field value to 4 (Parameter Problem type) and the Code field value to 1 (Unknown next-header code), to the packet source host. From the upper layer protocol function the packet goes to the appropriate socket and finally to the application.

If the host is a router and the packet is not for this host then the packet has to be forwarded, for this purpose the function *ip6_forward( )* will be called by the function *ip6_rcv_finish( )*, which is in *net/ipv6/ip6.output.c*. From this function the packet goes to the function *ip6_output( )* then goes to the function *ip6_output _finish( )* function. All these functions are defined in the same file *net/ipv6/ip6.output.c*. In these functions some of the Extension Headers, such as HbH, DH and RH will be processed if they exist otherwise handed over this packet to the physical layer function. This physical layer function will append the physical layer's header and transmits packet in to network. Figure 5.1 depicts the processing flow of a packet in a host.


CENTRAL LIBRARY
G 11707
Acc. No.
Date

128, which is Echo Request ICMPv6 message, it will be converted to 8, which is Echo Request ICMPv4 message. Other types (130, 131, 132, 133 to 137 and unknown query types in ICMPv6 message header) will be ignored.

4.4.4.2 Conversion of ICMPv6 Error Type Message

To convert ICMPv6 error type message to ICMPv4 error type message first of all we need to convert the inner IPv6 header of ICMPv6 message to the inner IPv4 header of the ICMPv4 message, and then convert the remaining part. We use the same procedure as we used to convert the IPv6 main header. For converting the ICMPv6 message to ICMPv4 message we use the mapping shown in Table 4.5 Other types (other than in Table 4.5) will be ignored.

IPv6		IPv4		Description
Type	Code	Type	Code	
128	0	8	0	Echo request,just change type and compute check sum
129	0	0	0	Echo Replyrequest,just change type and compute check sum

Table 4.5: ICMPv6 to ICMPv4 Query type translation

4.4.5 ICMP checksum calculation

ICMP header (either version 4 or 6) has a Checksum field and its checksum should be calculated by adding a pseudo-header The same IPv6 IPv4 rule is applied here to compute the Checksum field value. ICMPv6 message uses the pseudo-header having IPv6 addresses and ICMPv4 uses the pseudo-header containing IPv4 addresses. Including the pseudo-header in checksum calculation is not mandatory in case of ICMPv4 messages. But in case of ICMPv6 messages, it is mandatory and every host has to include the pseudo-header when it is calculating the checksum. In the pseudo-header, there is a

Length field, which specifies the length of the packet till the ICMPv6 message excluding the lower layer protocols. The checksum is calculated by taking 16-bit one's complement of the one's complement sum of the entire ICMP message starting with the ICMP message type field, prepended with a pseudo-header of IP header fields.

In this Chapter the translation procedure has been described and the basic formats of ICMPv4 and ICMPv6 messages have also been described. The conversion mechanism of an ICMPv4 packet to an ICMPv6 packet and vice-versa. Finally the concept of checksum and its calculation for the headers of ICMPv4 and ICMPv6 have also been described.

Chapter 5

IMPLEMENTATION

In this Chapter the implementation details have been explained. Linux Mandrake 8.1 Operating System (OS) having the kernel version 2.4.8.The machine configured as router uses two Intel® PRO/100 with eepro100 having chip no : 82557/58 supporting a data rate of 10/100 mbps. The systems configured as IPv4 LAN and IPv6 LAN use Intel series cards. The Networking Utilities used are tcpdump (3.6.2-1 mdk), which needs Libpcap0-0.6.2-1mdk. The working of the IPv4 and IPv6 packet processing in a receiving host, which is having Linux kernel 2.4.8 is described below.

5.1 Processing of a packet at a receiving host

A host, which is connected to a network through a Network Interface Card (NIC), will listen to all of the packets going through that network, even though all packets are not intended for this host. But NIC will listen to all of the packets and filters those packets that are intended for this host. These packets are stored and processed in the host. Each NIC has an address, what is usually 6 octets in length, unique in the network. The NIC will find the packets intended for this host by masking this NIC address with the packet's NIC address; if they are equal or the packet NIC address contains all 1s (broadcast address) then the NIC stores this packet in its local buffer. The NIC then raises an interrupt to signal that it received a packet, which is intended for this host. The processor will process this request by calling the appropriate driver interrupt handling routine. This routine will start the processing of the packet.

The NIC driver routine will take the packet from its buffer and then allocate kernel memory (because kernel has to process the packet until it reaches the application) for a *sk_buff* [26] structure and stores this packet in this structure. The *sk_buff* structure is defined in *include/linux/skbuff.h* When a packet arrives to the kernel, either from the user space or from the network card one of these structures is created. Changing packet fields is achieved by changing its fields.

```
struct sk_buff {

struct sk_buff * next;    /* Next buffer in list */
struct sk_buff * prev     /* Previous buffer in list */
struct sk_buff_head * list; /* List we are on */
struct sock *sk; /* Socket we are owned by */
struct timeval stamp; /* Time we arrived */
struct net_device *dev; /* Device we arrived on/are leaving
                           by */

        }
```

Before storing this packet, the NIC routine will strip the physical level header (Ethernet header or Token Rink header or FDDI header etc.). It then calls the generic packet handling function *netif_ rx(),* which is in *net/core/dev.c file* . This function queues this packet in a packet list structure of the current processor. Each processor will have a separate list for packets; the processor, which handled the NIC interrupt request, will get this packet in its packet list. The *netif_rx( )* function returns one of the following values which are defined in *include/linux/netdevice.h.*

```
#define NET_RX_SUCCESS 0 /* keep them coming*/
#define NET_RX_DROP 1 /* packet dropped */
#define NET_RX_CN_LOW 2 /* storm alert, just in case */
#define NET_RX_CN_MOD 3 /* Storm on its way! */
#define NET_RX_CN_HIGH 4 /* The storm is here */
#define NET_RX_BAD 5 /* packet dropped due to kernel error
                        */
```

After queuing the packet this function raises a software interrupt request NET_IRQ _SOFTIRQ. After receiving software interrupt request the processor will call the appropriate function to handle this interrupt. Each software interrupt is attached with a function at the time of loading the packet handler module. The function, net_rx_action( ), is attached to the software interrupt request NET_ IRQ_ SOFTIRQ **[20]** at the time of driver loading.

This *net_rx_action( )* function is defined in *net/core/dev.c file.* This function checks whether the packet belongs to any one of the network layer protocols registered in this host (*sk_buff* structure will have a field called protocol, this field contains the type of this packet). If the packet belongs to any one of the protocols registered at the network

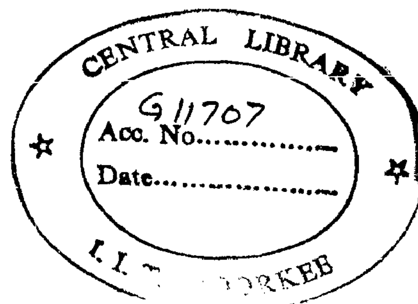
layer, then that particular protocol handling function will be called. For example if the packet belongs to IPv4 the function *ip_rcv()* will be called, else if it belongs to IPv6 then the function *ipv6_rcv( )* will be called. The function *ip_rcv( )* is defined in *net/ipv4/ip input.c* file. From this function *ip_input_finish( )* function will be called; in this function it will be decided whether this packet is actually for this host or for another host (because if this host is a router then it needs to process and forward the packet). By seeing the Destination Address field of the IP header, this function will decide whether this is for this host or not. If it is for the this host then that packet will be delivered first to the function *ip_local_deliver( )* and there to *ip_local_deliver_finish( ).* All these functions are defined in the same file, *net/ipv4/ip_ input.c.* In *ip_local_deliver_finish( )* function, it will be decided to which upper layer protocol (TCP/ UDP/ etc.) this packet has to be delivered. If this packet upper layer protocol is one of the registered upper layer protocols in this host then this function strips the IPv4 header and hands over the packet to the upper layer protocols. From the upper layer protocols this packet data goes to the application. When the upper layer protocol of packet is TCP then the function *tcp_v4_rcv( )* is called, else if UDP then the function *udp_rcv( )* will be called etc. If the packet upper layer protocol is not one of the upper layer protocols registered in this host, then the function *ip_ local_ deliver_finish( )* will send a destination unreachable (type 3) an ICMPv4 error packet having the Code field value 2 (protocol unreachable), to the source host of the original packet.

If the packet is not for this host and the host does not have the capability of forwarding a packet, then the packet will be discarded. If the host has the capability of forwarding a packet then the function *ip_rcv_finish( )* will call the function *ip_forward( )* which is in the file *net/ipv4/ip_forward.c.* This function calls the function *ip_forward_finish( ),* which is in the same file, from there it goes to *ip_send( )* function, where the packet length is checked. If the packet length is more than PMTU then the packet is fragmented. After fragmenting the packet is sent to the function *ip_ output _finish( )* function, which is defined in *net/ipv4/ip_output.c* file. If the packet length is not more than PMTU then the packet is handed over to the same function (*ip_output_ finish( )*) for further processing. This function calls the function *ip_finish_output2( ),*

which is defined in the same file. Finally this function calls the appropriate physical layer function to add that layer's header and to transmit the packet in the network.

If the packet is of type IPv6 then the function ipv6_rcv( ) will be called which is defined in *net/ipv6/ip6_input.c* file. From this the packet goes to the function ip6_rcv_finish( ), this function will decide whether the packet is for this host or for other host. If this host has forwarding capability then it forwards this packet otherwise discards it. If the packet is for this host then the function *ip6_input( )* will be called and from this function the packet goes to the function *ip6_input_finish( )* both these functions are defined in the file *net/ipv6/ip6_input.c*. In these functions, all the Extension Headers will be processed and the function *ip6_input_finish( )* will know what is the upper layer protocol. This function will check the packet's upper layer protocol with the registered upper layer protocols within this host. If it suits to any one of the upper layer protocols then this function strips the IPv6 header and calls the function depending upon the protocol. If the upper layer protocol is TCP it calls the function *tcp_v6_rcv( )* else if UDP then the function *udpv6_rcv( )* will be called etc. If the packet's upper layer protocol is not one of the upper layer protocols then this function sends an ICMPv6 error message [15], by setting the Type field value to 4 (Parameter Problem type) and the Code field value to 1 (Unknown next-header code), to the packet source host. From the upper layer protocol function the packet goes to the appropriate socket and finally to the application.

If the host is a router and the packet is not for this host then the packet has to be forwarded, for this purpose the function *ip6_forward( )* will be called by the function *ip6_rcv_finish( )*, which is in *net/ipv6/ip6.output.c*. From this function the packet goes to the function *ip6_output( )* then goes to the function *ip6_output _finish( )* function. All these functions are defined in the same file *net/ipv6/ip6.output.c*. In these functions some of the Extension Headers, such as HbH, DH and RH will be processed if they exist otherwise handed over this packet to the physical layer function. This physical layer function will append the physical layer's header and transmits packet in to network. Figure 5.1 depicts the processing flow of a packet in a host.


CENTRAL LIBRARY
Acc. No. G11707
Date.............

otherwise handed over this packet to the physical layer function. This physical layer function will append the physical layer's header and transmits packet in to network. Figure 5.1 depicts the processing flow of a packet in a host.

The below subsection discusses how the translation from IPv4 to IPv6 and vice-versa is implemented. We will call the IPv4 header protocol processing and upper layer protocol processing modules as IPv4 stack and we will call IPv6 header processing module as IPv6 stack. Whenever a router receives a packet it checks whether it is for the host or has to route it. If it has to route it then it calls appropriate functions to process this packet.

If the packet contains IPv4 header then it hands over the packet to the function *ip_forward( )* to route the packet. The code in this function is changed whenever the packet has to be routed from an IPv4 domain to an IPv6 domain, then call a function named as *ipv4_ to_ ipv6( )* which is defined in the file *net/ipv4/ipv4to6.c*. This function checks the packet to see if it is having any ICMP [9] header or not. If not, it just translates the IPv4 header to IPv6 header and computes the checksum of the upper layer protocols, such as TCP/ UDP (by using pseudo-header). If the packet contains an ICMP message then it checks the type of this message (whether query or error type). If the message is of type query then it translates the fields of the ICMPv4 message to the ICMPv6 message fields. If it contains an ICMPv4 error type message then this function first translates the inner IPv4 header into the IPv6 header, then it converts the ICMPv4 message into the ICMPv6 message.

If the packet contains an IPv6 header then it hands over the packet to the function *ip6_forward( )*. In this function check whether this packet is traveling from an IPv6 to an IPv4 domain, if so then this function calls a conversion function *ipv6_to_ipv4( )* which is defined in the file *net/ipv4/ipv4to6.c* else the function processes the packet as usual manner explained in the above paragraph. This function checks whether this packet has an ICMPv6 message or not. If it contains an ICMPv6 message then the conversion function checks the type of the ICMPv6 message (whether query or error). If the ICMPv6 message is query type then it converts the ICMPv4 message into the ICMPv6 message. If

it contains ICMPv6 error type message then it first translates the inner IPv6 header into the inner IPv4 header then it translates the ICMPv6 message into the ICMPv4 message. If the packet does not contain the ICMPv6 header then it just translate the IPv6 header and computes the checksum of the upper layer protocols (TCP/ UDP) by using the pseudo-

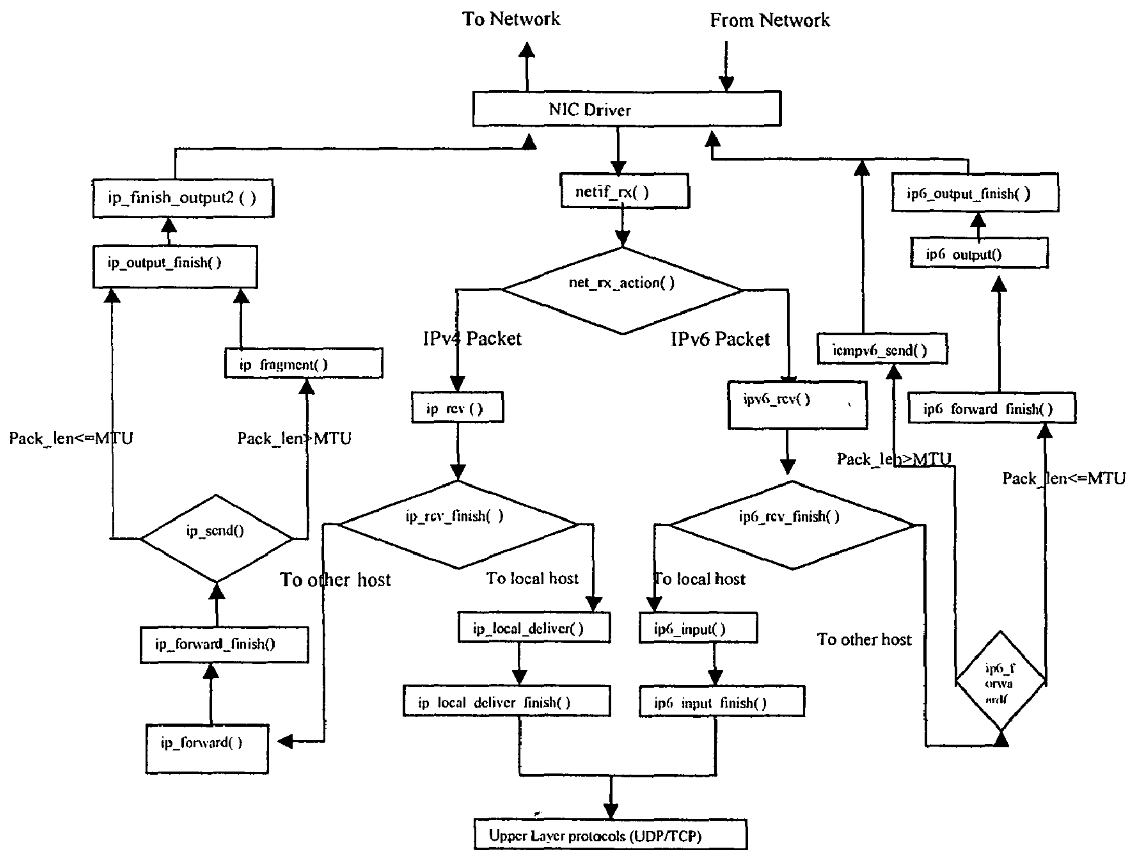


Figure 5.1: Processing of a packet in a host

header. Figure 5.1 gives chart of the packet processing in the router when it has to translate the packet from an IPv4 domain to an IPv6 domain or vice-versa.

5.2 Files changed

Some of the files in the IPv4 and IPv6 stack have been changed to implement this header conversion protocol. Here all of the files that have been changed and the files that have been added to the stacks are listed.

- # INCLUDE NET/ipv4to6.h, mangle_packet.h, ipv4to6stat.h, ipv6to4stat.h:

- # IPv4NET/ipv4to6.c, ipv4to6stat.c: These are two files added to IPv4 stack and contain the definitions of the header translations and statistics collecting functions.
- # IPv4NET/af_inet.c, ip_forward.c: These are two files, which have been changed in the IPv4 stack. In the first file we added proc files, which will help in collecting statistics. In the second file we added a condition which will check the packet path, (IPv4 to IPv6 or not) and calls the appropriate function.
- # IPv6NET/ipv6to4.c, ipv6to4stat.c, mangle_packet.c: These are three files added to the IPv6 stack and these contain the definitions of the functions which will translate an IPv6 header to an IPv4 header, collect statistics, and mangle the packet for the sake of generating bad formatted packets, which will be useful in testing ICMP message conversion procedure.
- # IPv6NET/af_inet6.c, ip6_output.c: These are two files, which have been changed in the IPv6 stack to suit the work that is being done. The first file some statements have been added to create some of the proc files, from which we can collect different statistics of the translation procedure. In the second file a condition is added, which will check the packet path, (IPv6 to IPv4 or not) and calls appropriate function.
- # IPv4NET/Makefile: As new files added to IPv4 stack, they need to be included in that stack. This file has been changed to compile these new files.
- # IPv6NET/Makefile: As new files added to IPv6 stack, they need to be included in that stack. This file has been changed to compile these new files.

Below shown are the data structures used in this conversion

```
Struct ipv6_opt_offset
        {
        unsigned short hop;
        unsigned short ra;
        /* --u16 zp; Jumbo payload is not considered */
        unsigned short rt0;
        unsigned short frag;
        unsigned char hop_flag:1,
        rt0_flag:1,
```

```
frag_flag:1,
reserved :5;
};
```

Above structure is useful in finding the offset of various extension headers in IPv6 packet. Flags says whether that particular header is existing or not and '*unsigned short* ' variable says the offset of particular extension header.

```
struct exthdr_info
{
u8 protocol;     // Which is upper layer protocol extracted from extension header next header field
u32 size;        // Size of extension headers in IPv6 header
u32 nxthdr_offset; //offset of above protocol field from IPv6 main header
};
```

This structure is useful to remember the IPv6 upper layer protocol value, size of the extension headers and offset is 'next header' field value for this protocol

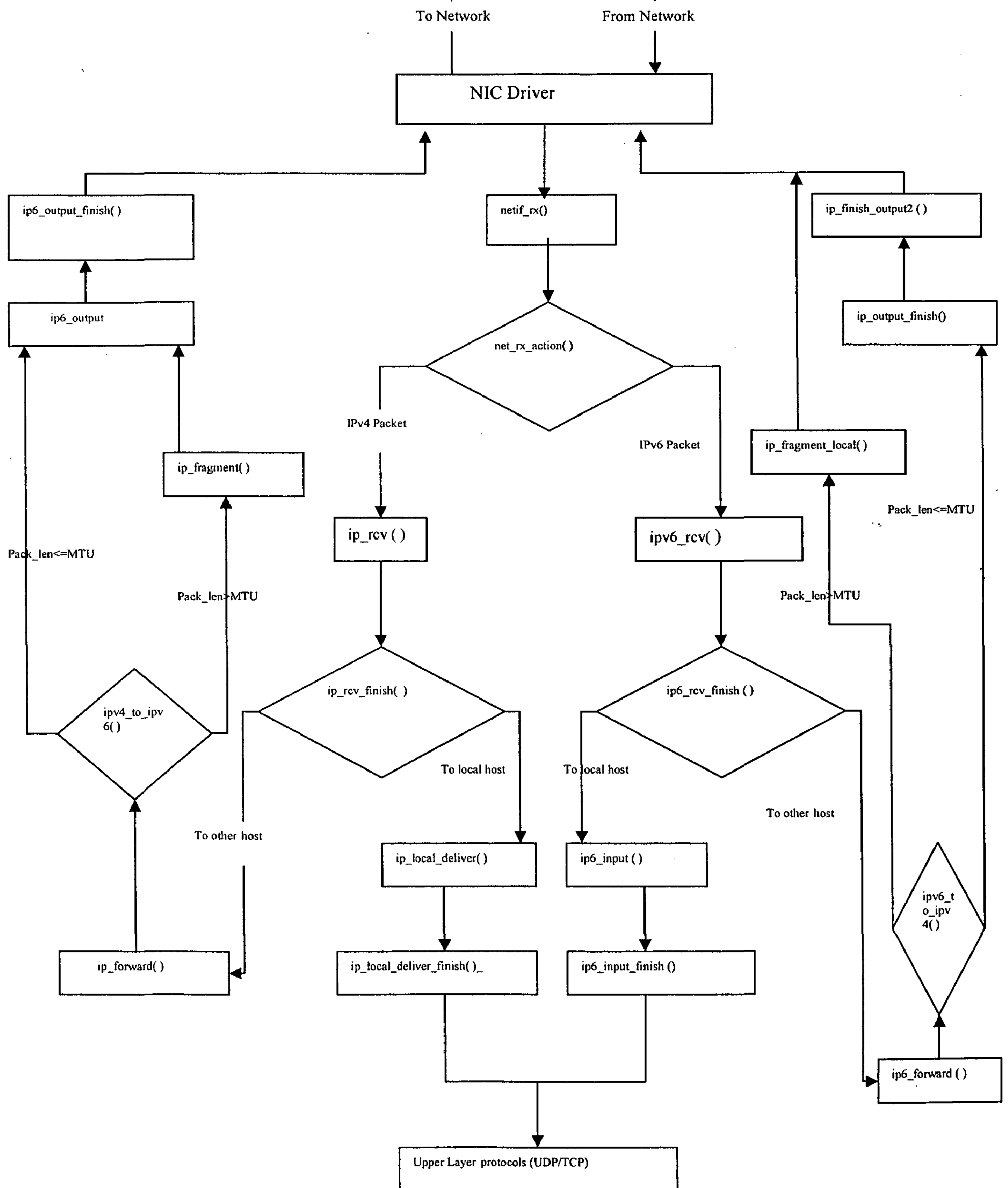


Figure 5.2 : Processing flow of a packet in a host

5.3 Testing

The translation procedure is tested on 3 hosts, in which, one is acting as a router, one is acting as a host in IPv6 domain, and last one is acting as a host in IPv4 domain. All of these 3 hosts have the same Linux kernel, which is 2.4.8. By using the socket programming technique, we have checked the header conversions, like by opening TCP/ UDP socket checked whether TCP/ UDP packets are traveling from one side to other. To check the robustness of the system, a packet generator is used, which will generate number of packets that have been specified, like if it is specified to generate 200 TCP type IPv4 packets then it will generate 200 IPv4 packets having TCP as the upper layer protocol, in generating the packets.

Chapter 6

RESULTS

Configuring the Kernel

The kernel is configured to optimize for harware and usage patterns .one can use any of the available ways to edit and configure the kernel options before compiling it, for example make menuconfig [24] or make xconfig. The make *xconfig* has been used for configuring the kernel. The most important kernel options that must be enabled to support IPv6 is:

• Under the *"Networking Support" section: The IPv6 protocol (experimental) – yes.*

This enables the experimental support for IPv6.When one has finished configuring the kernel options and one can save configuration, one will have the kernel configuration file .config created in /usr/src/linux.

Code maturity level options	SCSI support	File systems
Loadable module support	Fusion MPT device support	Console drivers
Processor type and features	IEEE 1394 (FireWire) support (EXPERIMENTAL)	Sound
General setup	I2O device support	USB support
Memory Technology Devices (MTD)	Network device support	Bluetooth support
Parallel port support	Amateur Radio support	Kernel hacking
Plug and Play configuration	IrDA (infrared) support	
Block devices	ISDN subsystem	
Multi-device support (RAID and LVM)	Old CD-ROM drivers (not SCSI, not IDE)	Save and Exit
Networking options	Input core support	Quit Without Saving
Telephony Support	Character devices	Load Configuration from File
ATA/IDE/MFM/RLL support	Multimedia devices	Store Configuration to File

Figure 6.1:kernel options for configuration

Compiling the Kernel

Once you exit the kernel configuration tool, we need to make dep **[7]** and make clean. This ensures that all of the dependencies, such as the include files, are in place and removes all of the object files and some other things that an old version or compilation left behind.

make dep
make clean
Now one can compile the kernel. The following command puts the kernel with the changes in the source code in the appropriate location.

make bzImage

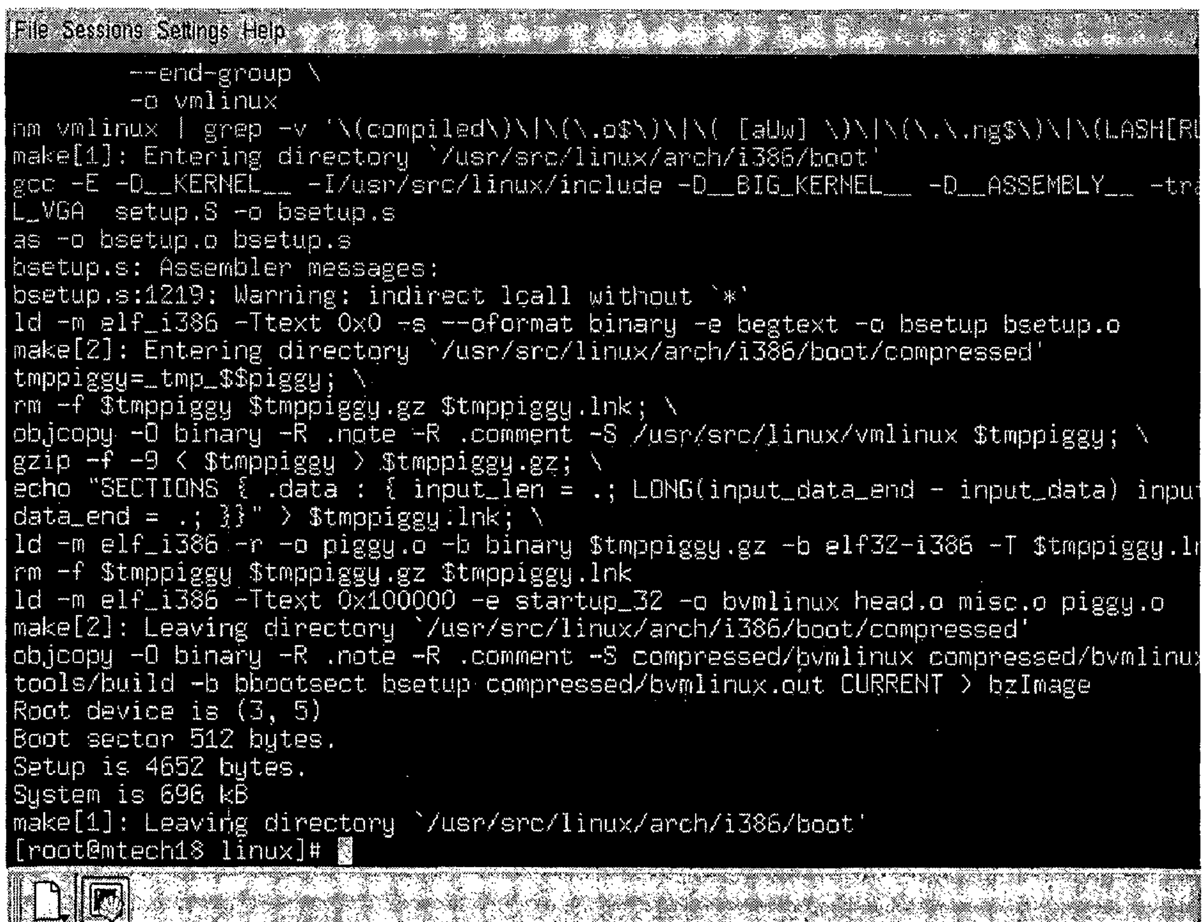


Figure 6.2: Compiling the kernel

The result of the compilation will be a new compressed kernel image created in /usr/src/linux/arch/i386/boot/bzImage. There is support for features such as loadable modules, so one can compile and install the modules by applying:

```
# make modules
# make modules_install
```

Once completed, copy the new kernel image bzImage and System.map to the boot directory:

```
# cp arch/i386/boot/bzImage /boot/bzImage-2.4.8-2
# cp /usr/src/linux/System.map /boot/System.map-2.4.8-2
```

Adding a New Kernel Entry in the Boot Loader

As the boot loader being used is LILO a new entry in the LILO configuration file (/etc/lilo.conf) has to be made for the newly baked kernel. At boot time, this kernel entry will be presented to the user in the list of kernels to choose from. On our test system, we added an entry that looks as follows:

```
Image=/boot/bzImage-2.4.8-2
label=Bridge
root=/dev/hda5
read-only
```

Following that, run */sbin/lilo* to install the boot loader with the newly configured options you configured in */etc/lilo.conf*. The next time after booting the machine the presence of a kernel entry presented in LILO at boot time with label. Bridge is found indicating the presence of a newly compiled kernel that supports the interoperability between IPv4 and IPv6.

The snapshot below shows the packet generator used for sending the packets between hosts and generating IPv4 and IPv6 packets .The fields are filled in appropriately as required. For testing TCP/IP packets have been sent .The below figure shows the contents of IP header and payload .The fields are changed randomly, and statistics have been taken.

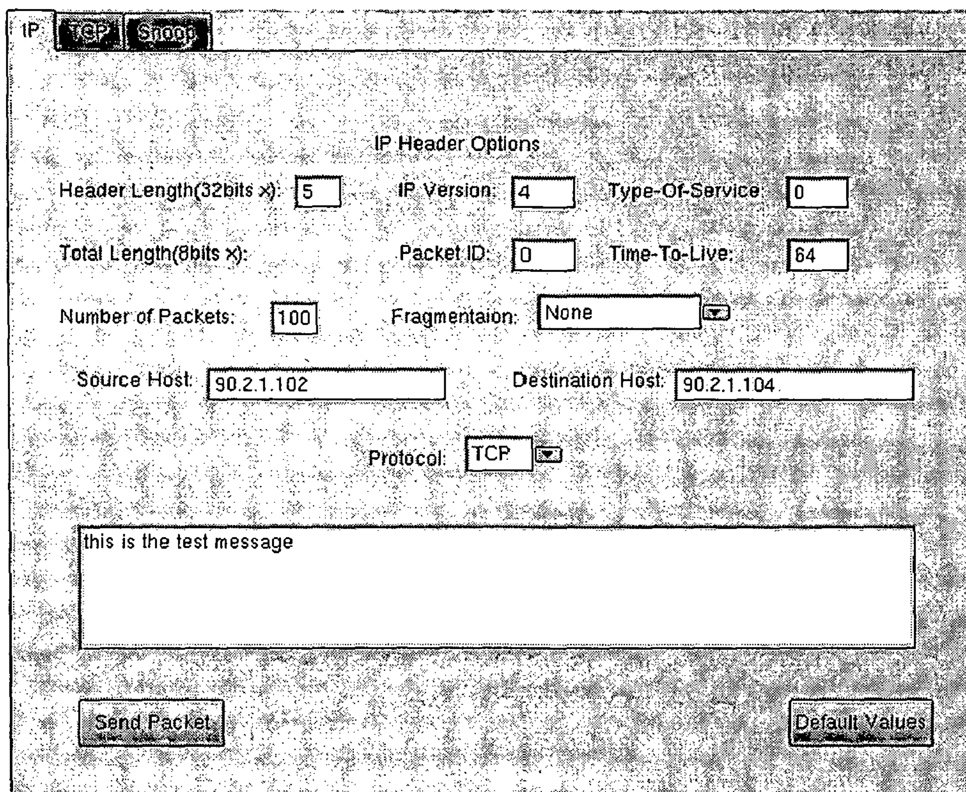


Figure 6.3: IP header and data fields of the packet generator

The above snapshot suggests that 100 TCP/IP packets are sent from the source having the IP address 90.2.1.102 to 90.2.1.104 with header length 5, version is 4 as the address being used is 32 bits. Time to live is set to 64. The Type-Of-Service field has been set to zero. The payload of the packet is the string " this is the test message"

The Figure 6.4 below shows the TCP header fields to be filled in for the packet. The source port and destination port can be anything above the commonly used port numbers, which are in the range 0-1023.The flags also can be set and the packets are sent. The source port no that is being used is very high and has the no 25000 and the destination port no is 36000. The data offset is 5. The sequence number is 532387333. The window size is 1504 bytes. The send button option sends the required no of packets.

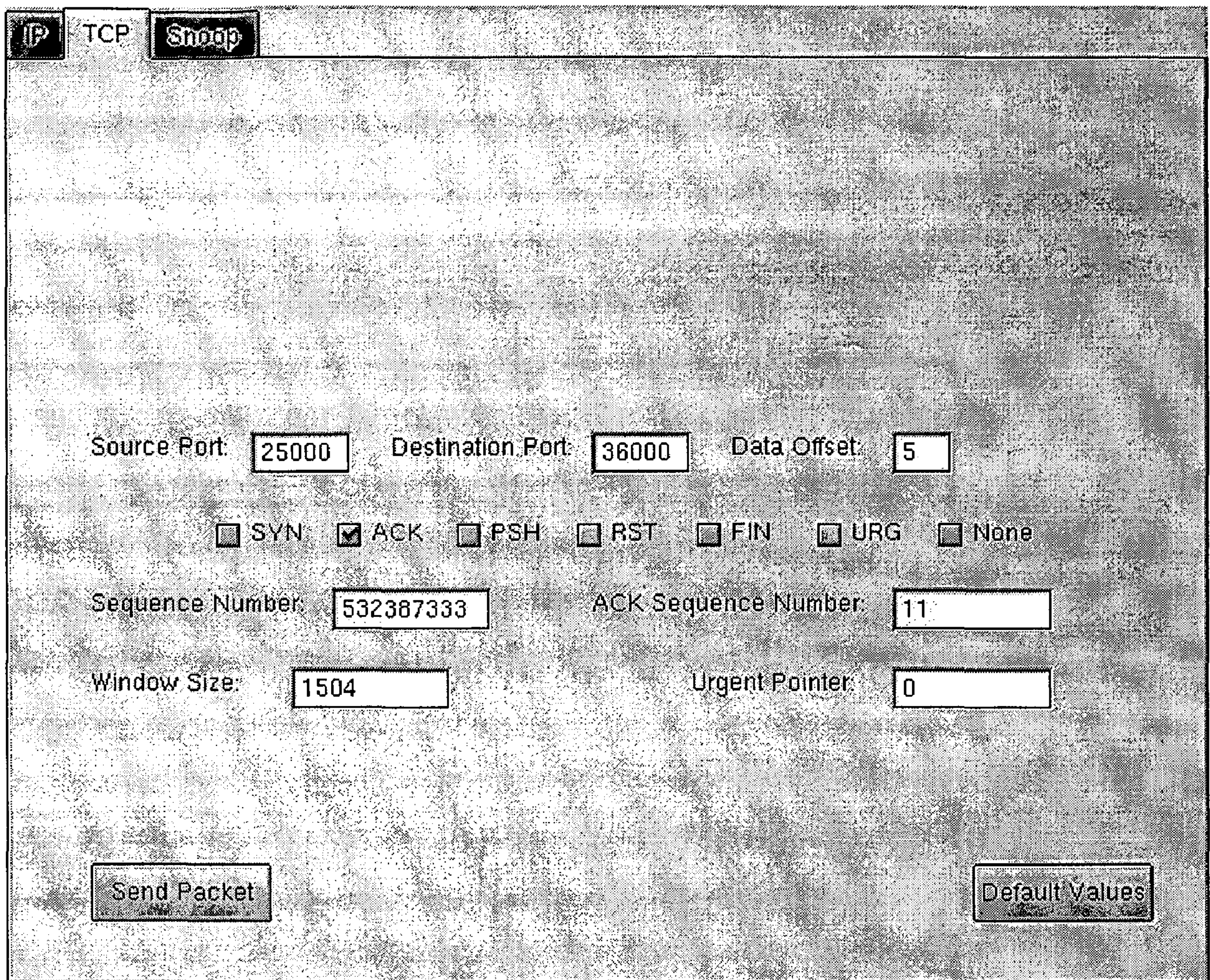


Figure 6. 4: TCP header and data fields of the packet generator

At the receiver side packets are captured using the tcpdump utility .into text file

#tcpdump tcp > pkttest // Dump TCP packets onto the file pkttes

The entries of "pkttest" are as follows:

```
01:07:11.902686 90.2.1.102.25000 > prem.local.domain.36000: . 270736930:270736937(7) win 6045
01:07:11.902800 prem.local.domain.36000 > 90.2.1.102.25000: R 0:0(0) ack 1 win 0 (DF)
01:07:12.062419 90.2.1.102.25000 > prem.local.domain.36000: . 270736930:270736937(7) win 6045
01:07:12.062520 prem.local.domain.36000 > 90.2.1.102.25000: R 0:0(0) ack 1 win 0 (DF)
01:07:12.902662 90.2.1.102.25000 > prem.local.domain.36000: . 270736930:270736937(7) win 6045
01:07:12.902769 prem.local.domain.36000 > 90.2.1.102.25000: R 0:0(0) ack 1 win 0 (DF)
01:07:13.062331 90.2.1.102.25000 > prem.local.domain.36000: . 270736930:270736937(7) win 6045
01:07:13.062396 prem.local.domain.36000 > 90.2.1.102.25000: R 0:0(0) ack 1 win 0 (DF)
01:07:13.902710 90.2.1.102.25000 > prem.local.domain.36000: . 270736930:270736937(7) win 6045
01:07:13.902823 prem.local.domain.36000 > 90.2.1.102.25000: R 0:0(0) ack 1 win 0 (DF)
01:07:14.902652 90.2.1.102.25000 > prem.local.domain.36000: . 270736930:270736937(7) win 6045
01:07:14.902762 prem.local.domain.36000 > 90.2.1.102.25000: R 0:0(0) ack 1 win 0 (DF)
01:07:15.902799 90.2.1.102.25000 > prem.local.domain.36000: . 270736930:270736937(7) win 6045
```

01:07:15.902908 prem.local.domain.36000 > 90.2.1.102.25000: R 0:0(0) ack 1 win 0 (DF)

A typical entry in the textfile "pkttest" is as below and interpreted as said below
01:14:27.904238 90.2.1.102.25000 > prem.local.domain.36000:
270736930:270736937(7)
win 6045

The above entry shows the timestamp of the packet i.e time at which the packet was sent from source, followed by the source IP 90.2.1.102 with the source port no 25000 and destination IP resolved to name as prem.local.domain with destination port no 36000 and the byte sequence range starting at 532387333 and transferring a total of 7 bytes as specified in the brackets and the window size of the buffer.

A walkthrough of the " pkttest" indicates that for first 10 packets sent from the system the start and end packet timestamps were 01:14:04.996125 and 01:14:27.904283 indicating the timestamps at around 1:15 p.m. The time taken per packet to travel from one host to another was 2.3007968 milliseconds. The first packet that is emerging out is taking more time as compared to the successor packets because the line link was empty at the first packet was sent and the following packets after the first one are in a pipeline. so the end-to-end delay is more for the first packet than the second, third and the following packets.

Following the same scheme a packet is sent from a system that is configured as IPv4 system having a 32 bit address that goes to the router first and then it is processed by the kernel, the fields of the packet are changed as specified and pushed on to the network wire by the system that acts as a router and the packet reaches the system that is configured for receiving IPv6 packets and it processes the packet appropriately. This process is to repeated for 100, 200. packets for getting the statistics about the translation time. In this way system performs the required task.

Chapter 7

CONCLUSION AND FUTURE WORKS

In this work, we have developed a mechanism by which hosts in IPv4 network can communicate with the hosts in IPv6 network and vice-versa .The work is done on Linux mandrake 8.1 having the kernel 2.4.8.The kernel files have been modified and new file entries have been made to suit the requirements. The entries in the make files corresponding to the directory having the modified and newly added files to the ipv4 and ipv6 stacks (directories) of the kernel source tree are set appropriately. The exhaustive process of kernel recompilation after many trails and errors took 50 minutes on our system, which uses a Pentium –III processor. We have implemented this mechanism and deployed on a Linux-based router and tested it rigorously. The test results have been analyzed for correctness and estimated the time required for translating the packets. The translation time has been found to be quite low in our implementation.

7.1 Future Works

A configuration file is used to convert from an IPv4 to an IPv6 address in this mechanism. Due to this limitation of one-to-one address mapping, this mechanism will be useful in a small network, where address management can be done easily. By extending the address mapping mechanism at the DNS (which will provide mapping addresses), this system can be deployed in a bigger network containing more number of hosts. The mechanism uses general purpose Linux OS, if the OS kernel is customized to suit this mechanism then the present system performance can be improved. The system customization not only improves the performance it also decrease the size of the OS kernel. We can use tunneling in sending packets from an IPv4 subnet to another IPv4 subnet via an IPv6 subnet. By implementing the DNS address mapping technique we can deploy the router some where in the middle of the network (not as we deployed at the end of the IPv6 subnet) we can do the tunneling. When ever a packet comes to the router the router can decide whether it has to tunnel it or to translate to another protocol. By implementing these mechanisms we can have a full router.

REFERENCES

[1] Postel J., "Internet Protocol", RFC 791, September 1981. http://www .ietf.org/rfc/rfc791.txt.

[2] Jorg Ottensmeyer., " IP version 6 –analysis of the long way from concept to large scale deployment" Proceedings of the 28th Euromicro Conference, IEEE 2002, pp.229-235 .

[3] Deering S and Hinden R., " Internet Protocol, Version 6, (IPv6) Specification ", RFC 2460, December 1998. http://www .ietf.org/rfc/rfc1998.txt.

[4] Brian Carpenter Internet Society (ISOC) – "IPv6 and the Future of the Internet", 23 July 2001. http://www.isoc.org/briefings/001.

[5] Marsan C.D., "The next best thing to IPv6?", NetworkWorldFusion, September , 1999. http://www.nwfusion.com/news/1999/0920ipv6.html

[6] Jun Tian and Zhongcheng Li., "The Next Generation Internet Protocol and its Test" , IEEE internet computing, July/August 2001, pp:210-215.

[7] J. W. Atwood, Kedar C. Das, and Ibrahim Haddad , "Providing IPv4 /IPv6 and IPv6/IPv4 Address translation" Ericsson –Canada Open Systems Lab- IPv6 page http://www.linux.ericsson.ca/ipv6

[8] Pete Loshin, "IPv6 Over Everything", Data Communications Technical Tutorials, October 21,1999.

[9] "IPv6 Introduction and Conversion Guide", Fujitsu Seimens Computers GmbH, 2001. http://manuals.fujitsu-siemens.com

[10] Jun Tian, Qi wu,Zhongcheng Li., "A Formal Method in IPv6 Testing", The Sixth Computer Sciences and Technology Conference, China, pp.288-295, July 2000.

[11] E .Nordmark., "Stateless IP/ICMP Translation Algorithm (SIIT)", RFC 2765, February 2000.

[12] Vano Guardini ,Paolo Fasano and Guglielmo girardi., "IPv6 Operational Experience Within the 6bone ", INET 2000, Japan, July 2000, pp.147-152.

[13] IPv6 Forum, http://www.ipv6forum.com/.

[14] P.E. Miller and M.A. Miller., “Implementing IPv6: Supporting the Next Generation Internet Protocols, 2/e", John Wiley & Sons, Inc, 2000.

[15] Conta A. and Deering S., “ICMP for the Internet Protocol Version 6 for the Internet Protocol Version 6 (IPv6)", RFC 2463, December 1998. http://www .ietf.org/rfc/rfc2463.txt.

[16] Tanenbaum, A.T, “Computer Networks”, 3/e", Prentice-Hall, Inc., 1996.

[17] Comer D.E., “Internetworking with TCP/IP, Volume I: Principles, Protocols and Architecture, 4/e", Prentice-Hall, Inc., 2000.

[18] Bovet P.D. and Cesati M., “Understanding the Linux Kernel", O'Reilly & Associates, 2000.

[19] Fuller V., Li, and Varadhan K., “Classless Inter- Domain Routing (CIDR): An Address Assignment and Aggregation Strategy", RFC 1519, September 1993.

[20] Aivazian, T, “Linux Kernel 2.4 Internals", http://en.tldp.org/LDP/lki/index.html

[21] Bieringer, P, “Linux IPv6 HOWTO", http://www.tldp.org/HOWTO/Linux+IPv6-HOWTO

[22] O. Kirch, and T. Dawson, “The Linux Network Administrator's Guide, 2/e". http://en.tldp.org/LDP/nag2/index.html.

[23] Hinden R and Deering S., “IP Version 6 Addressing Architecture", RFC 2373, July 1998. http://www .ietf.org/rfc/rfc2373.txt.

[24] Kwan lowe, “ Kernel Rebuild Guide” ,February 2004. http://www.digitalhermit.com/linux/Kernel-Build-HOWTO.html

[25] Atkinson R., “IP Encapsulating Security Payload (ESP)", RFC 2406, November 1998. http://www .ietf.org/rfc/rfc2406.txt.

[26] "IP Layer Input Packet Processing and Received Packet Processing"
http://www.ecsl.cs.sunysb.edu/ elibrary/linux/network/iprecv.pdf

[27] "IPv6 Address Allocation and Assignment Policy", February 2003.
http://www.ripe.net/ripe/docs/ipv6-policy.html.

[28] A general IPv6 information homepage.
http://www.ipv6.org/

[29] M. Blanchet, F. Parent., "IPv6 transition mechanisms", May 2002, Viagenie
http://www.viagenie.qc.ca/

[30] Tsirtsis G and Srisuresh P., "Network Address Translation - Protocol Translation (NATPT)", RFC 2766, February 2000, http://www .ietf.org/rfc/rfc2766.txt.

[31] Postel J., "Internet Control Message Protocol", RFC 792, September 1981.
http://www .ietf.org/rfc/rfc792.txt.

[32] Michael Mackay, Christopher Edwards, and Martin Dunmore , Tim Chown, and Graca Carvalho "A Scenario-Based Review of IPv6 Transition Tools" , IEEE internet computing, May/June 2003. http://computer.org/internet/

[33] Commission of the European communities: Communication from the Commission to the Council and the European Parliament; "Next Generation Internet –priorities for action in migrating to the new Internet protocol IPv6", Brussels , Feburary2002 .
http://www.europarl.eu.int/meetdocs/committees/agri/20020710/com(2002)080_en.pdf

[34] Kent S and Atkinson R., "IP Authentication Header", RFC 2402, November 1998. http://www .ietf.org/rfc/rfc2402.txt.

[35] T. Chown, "IPv6 Initiatives within the European National Research and Education Networks (NRENs)," Proc. IEEE 2003 Symp. Applications and the Internet (SAINT) Workshops, IEEE CS Press, 2003, pp.149-152.