Implementation of IPv6 for Linux

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Jaya Ram M

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

June 1996

CERTIFICATE

This is to certify that the work contained in the thesis entitled Implementation of IPv6 for Linux by Jaya Ram M has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Dheeraj Sanghi,
Assistant Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology, Kanpur.

Abstract

The recent strides in networking technology, along with the wide availability of low-cost computing power, resulted in the explosive growth of the Internet. This growth has also opened up possibilities for a wide variety of new applications and services. The current Internet Protocol is not well equipped to handle neither the growth nor the widely varying service requirements of the new applications. So the IETF has standardized the new Internet Protocol IPv6 as the successor of IPv4. We have built a basic IPv6 module in the Linux kernel. This implementation can be used as a testbed for conducting research into various issues related to deployment and performance of IPv6. We have also created an IPv6 backbone for the IIT Kanpur campus network. This backbone allows the deployment of IPv6 in the campus network in an incremental manner.

Acknowledgements

I am grateful to **Dr. Dheeraj Sanghi** for his constant guidance throughout this work. I am also thankful to **Prof. K. R. Srivathsan** for allowing unlimited access to the facilities in the ERNET lab. I also thank him for supporting me through research assistantship for the first 18 months of my stay in IIT. I would like to thank **Dr. D. Manjunath** for his constant encouragement and support. The **Class of '94** has made my stay here, a beautiful and memorable experience.

Contents

1 Introduction **1**

1.1 IPv6 the *Next Generation* IP 1

1.2 Organization of thesis 2

2 Overview of IPv6 **4**

2.1 Growth of the Internet 4

2.1.1 Address space crunch 5

2.1.2 Routing table explosion 7

2.2 Emerging applications 8

2.3 Transition 11

2.4 Overview of IPv6 12

2.4.1 Simple header formats 12

2.4.2 Addressing architecture 13

2.4.3 Routing 17

2.4.4 QoS Handling 19

2.4.5 Miscellaneous features 21

3 Networking subsystem of Linux **23**

3.1 General design principles 23

3.2 Boot time initialization 24

3.3 Network buffer management 25

3.4 Network Device Layer 27

3.5 Adding a new protocol family 29

3.6 Member protocols in a family 30

4 Building the skeleton **32**

4.1 BSD API 32

4.1.1 New structures and constants 33

4.1.2 Address conversion functions 35

4.1.3 Socket options 36

4.2 Initializer for INET6 38

4.3 Initializer for raw IPv6 39

5 System calls **40**

5.1 Socket 40

5.2 Close 43

5.3 Sendto 43

5.3.1 Sanity checks 44

5.3.2 Next hop determination 45

5.3.3 Buffer allocation 47

5.3.4 Building the headers 49

5.3.5 Forwarding to the device layer 51

5.4 Recvfrom 52

5.4.1 Bottom half 52

5.4.2 Upper half 55

5.5 Setsockopt and getsockopt 56

6 Tunneling and 6Bone **57**

6.1 Tunneling 57

6.1.1 Tunnel driver 58

6.2 6Bone 60

6.2.1 Local setup 60

7 Tools and utilities **64**

7.1 The `/proc` file system interface 64

7.2 Packet snooper 66

7.3 Miscellaneous utilities 66

8 Conclusions **67**

8.1 Future Work 68

References **69**

Chapter 1

Introduction

1.1 IPv6 the *Next Generation* IP

The explosive growth of the Internet in the recent years has led to a number of difficulties with the existing Internet Protocol - IPv4 [J81]. The high speed networking technologies like ATM, etc., coupled with the wide availability of low cost computing devices, have resulted in a large number of new applications and services. IPv4 is not able to cope with the explosive growth of the internet. It also does not have adequate facilities to support the widely varying service requirements of these new applications. So the need for a new Internet Protocol, that would address these issues of scalability and quality of service, was felt widely. In 1991 the IETF embarked on the process of defining a new Internet Protocol - *IPng*. Later, this protocol was given a version number of 6 and now, officially, it is known as *IPv6*.

IPv6 has been designed as an evolutionary step from IPv4. This makes the transition from IPv4 to IPv6 easier, as IPv6 can be installed into the current IPv4 nodes as a normal software upgradation. It is expected to be efficient throughout the networking technology spectrum ranging from high-speed ATM networks to low-speed wireless networks. The protocol is also designed to be future-safe in the sense that, it provides a framework for easy addition of new features. This new protocol has a huge address space with 128-bit addresses and also allows route aggregation at multiple hierarchies.

In this project we built a basic IPv6 implementation that may serve as a good framework for further research and development. This can be used to experiment with various approaches and tradeoffs involved in the issues that are still unresolved. Attempts have been made to make this implementation as extendible as possible, so that extra functionality can be added as the standards evolve.

We implemented this in Linux-1.3.24 kernel. The choice of Linux was driven by its growing popularity and the wide availability of its source code. Moreover, Linux is being ported to a wide variety of architectures and expected be much more widespread.

We could also create an IPv6 backbone for the IIT Kanpur campus network, by running the router code on two Linux machines. The utility of this backbone has been demonstrated by running two more Linux machines as IPv6 hosts connected to this backbone. This facility allows any host in the campus network to be upgraded to IPv6 with minimal effort.

1.2 Organization of thesis

The rest of the thesis is organized as follows. In **Chapter 2** we present an over view of IPv6. We discuss the problems faced by IPv4 and explain how IPv6 is expected to address these problems.

In **Chapter 3** we explain the design and implementation of the networking subsystem in Linux.

In **Chapter 4** we describe how the basic framework of the IPv6 module is built.

In **Chapter 5** we explain in detail the implementation of the relevant system calls and processing done at the interrupts.

In **Chapter 6** we talk about tunneling support provided, and the IPv6 backbone that has been set up in IIT Kanpur.

In **Chapter 7** we describe the utilities that aid the setup and maintenance of the IPv6 network.

Finally, in **Chapter 8** we conclude by summarizing the results of our effort and discuss what can be done as future work.

Chapter 2

Overview of IPv6

In the following sections we will see how various problems and inadequacies of IPv4, the generally accepted vision of the future networking, and the plan for transition have influenced and driven the development of IPv6. Bradner [SA95] gives a detailed account of the development process.

2.1 Growth of the Internet

IPv4 was designed in 70s and the developers could not foresee the kind of growth the Internet is experiencing today. The TCP/IP stack was designed with a few thousand networks in mind. However the latest surveys indicate that at the beginning of 1996 there are over 9 million hosts in 240,000 domains spread over nearly 100,000 networks [Wiz96]. Even as late as 1987, the projections [R87] indicated that the number of networks in the Internet would be approaching the 100,000 mark at some *"vague"* point in distant future, but it has happened in 1996 itself. The early 90s witnessed a phenomenal growth with the Internet doubling its size every 12 months.

2.1.1 Address space crunch

Though the IPv4 addressing mechanism with its 32-bit addresses can address over 4 billion hosts spreading over some 16.7 million networks, the theoretically achievable efficiency of address space usage is very poor [C94]. The reason is the “granularity” of address assignment. The boundary between the network number and host number in an IPv4 address is always located on a byte boundary. A class ‘C’ network can have only 254 hosts which is too small for most organizations. So class ‘B’ network addresses were assigned which bundled 65535 addresses. which is again too large for most organizations and a lot of addresses went unused.

For the reasons mentioned above,, the class ‘B’ addresses were the most popular, and IETF felt in 1992, that they would get exhausted by March 1994, if they were allocated at the same rate. The Address Life Expectancy (ALE) working group of the IETF estimated that the last IPv4 address would be assigned some time between 2005 and 2011. But the ALEWG assumed the growth rate of the year 1992 which is far below that of 1996. The revised estimates put the address life expectancy between 4 and 5 years from 1996.

A number of solutions were suggested for extending the address life expectancy of IPv4.

- ALEWG suggested careful renumbering of major portions of internet to recover and minimize the unused addresses. The IETF felt that the effort involved was enormous and rejected the idea [Gro94].

- Use temporary IP addresses for intra-net conversations [Wan92].
  The idea here is to remove the restriction that the IPv4 addresses be unique across the world. All the hosts that are behind a firewall need not have globally unique addresses. If all of the hosts on one network have unique addresses among themselves, there would be no problem for the intra-net conversations. However when a host wants to establish a connection with another host outside

the net, it requests an address-server to assign a globally unique IPv4 address for the duration of the connection. thus a handful of IPv4 addresses will be shared dynamically among the all hosts of the net. But this will restrict the number of simultaneous outgoing connections and will also require some special setup.

- Since the Class 'C' addresses are relatively abundant, an organization should be assigned multiple class 'C' addresses instead of a scarce class 'B' address [VTJK93]. This technique conserves the addresses, as the smaller granularity of class 'C' addresses allows a more precise number of addresses to be assigned, which results in very small number of unused addresses. This scheme makes the total number of networks in the Internet very large, which means that the routing table sizes explode. But fortunately this can be tackled by deploying CIDR. (More detailed discussion will follow later.)
- Utilize the remaining class 'A' addresses in a more conservative way by assigning one class 'A' address to a number of smaller networks [IAN95]. Here the basic point is to use some sort of "subnetting" in class 'A' addresses.
- Reserved addresses can be considered for direct assignment.
- Reserved addresses can be used to extend the length of IP address from 32 bits [Siy92]. If this special address is seen in the address field, the node should look into the IPv4 options and get the remaining part of the address.

Actually, none of these techniques offer a permanent solution to the address space crunch. At the most they postpone the inevitable exhaustion of the addresses by a few more years. Furthermore, the long standing demands for better support for QoS and mobility, etc., were not addressed by any of them. Because of these factors, IETF started to evolve a new Internet Protocol with an easy transition plan.

2.1.2 Routing table explosion

The IPv4 addressing mechanism supports a very primitive level of hierarchy with only two levels: the host and the network. Aggregation of routes for different hosts in a particular network can be done by advertising only a single route for the entire network. The routers outside this network will have a single routing table entry for the entire net. But there is no easy way of sharing a routing table entry for collection of networks, that can be reached through the same next-hop node. Which means that the routing information cannot be aggregated beyond the level of a network. This is because, we cannot advertise single routes for a collection of networks, since the network is the biggest aggregation unit and there is no way of “identifying” or “addressing” a collection of networks.

But the general topology of the Internet is a hierarchical one. It is mandatory that the backbone routers keep a separate routing table entry for each of the networks it can reach. So these routers will have very large routing tables and this has a direct bearing on their performance. Both searching and updating these tables become expensive operations because of their size.

The concept of Classless Inter Domain Routing (CIDR) shows a way out of this gloomy situation [VTJK93]. It is a strategy that improves the address space utilization as well as solve the problem of the routing table explosion. The central theme of this technique is to abolish the concept of classes in the IPv4 addresses. The addresses are treated as continuous 32 bit identifiers, and a mask identifies a portion of the address that forms the network number. The rest of the address denotes the host number within that network.

In this new scheme of things, both the advertisement as well as usage of routing information needs to change. While routing, the router picks up the routing table entry, with the longest key that fully matches the destination address. Now it is possible to advertise single routes to a higher level router for a collection of networks, if the leading bits in their network numbers common. Such routing table entries will have only the common part address as the key, which makes sure that any packet destined to any of the networks in that collection, will match this entry and will be forwarded accordingly. The longest match ensures that the aggregation does not shield off better routes to the smaller groups i.e., if available, a route to a host will be preferred over a route to a network which inturn will be preferred over a route to a collection of networks.

This kind of route aggregation over topological boundaries will work, only if the address assignment is done carefully, so that the networks with identical leading bits in their numbers, are also closely located topologically. The discussion about how the addresses should be assigned can be found in [Rek93].

CIDR was implemented in backbone routers three years ago and it could contain the routing table explosion till recently. Now the routing tables have started to grow again. But the experience was very valuable and influenced the address architecture design of IPv6 to a great extent.

2.2 Emerging applications

Till now the IPv4 has been serving networks that are primarily data-oriented. Most of the existing applications belong to two categories: bulk file transfer and interactive. The QoS requirements of both these types are very minimal; they only required data integrity which was easily provided by checksums and retransmissions.

The advent of high speed networking technologies, like ATM, and the cheap computing power made available by the new generation micro processors, opened up a host of new generation applications ranging from distributed simulation to satellite navigation systems in cars. These kind of applications require fairly sophisticated QoS guarantees, in terms of bounds on delays, availability of bandwidth, and bounds on packet losses, etc., from the underlying protocols. As the world is moving towards

BISDN, IPv6 is expected to carry a variety of traffic with different QoS requirements. Similarly support for mobility, security and authentication, will all become mandatory. Further, IPv6 is supposed to be efficient on a wide variety of transmission media. Since most of these new applications are targeted at common users, there should not be any need for extensive configuration of devices unlike the IPv4 nodes. The IPv6 should allow and give some facilities for the devices to configure themselves as much as possible. These features are popularly known as “Plug and Play”.

In the rest of this section we discuss a few emerging application areas and examine the kind of requirements they place on IPv6.

Nomadic computing

It is fairly certain that personal computing devices like PDAs and the mobile phones are going to be ubiquitous in the near future. As they become more and more a necessity, they tend to merge together into a single gadget with capabilities of personal communication as well as computation and information retrieval. The main requirement for these devices is that they be networked.

This market is going to be enormous and it will place a great demand on the address space and the routing infrastructure. Since these devices are going to be mobile and are also going to be used for personal communication, mobility, security and authentication should be supported as basic elements of IPv6. IPv6 should also offer least possible overhead as these devices are going to be operated on the low bandwidth and lossy RF wireless networks. The protocol should have as little processing over head as possible. Accounting will be a useful feature to support because the market for these applications is going be consumer-oriented and they would like be charged exactly for what they have consumed. Support for policy-based routing will be helpful for multiple service providers to inter-operate. More detailed discussion can be found in [Tay94].

Networked entertainment

The strides made in hardware technology in recent times allowed complex multimedia encoding algorithms to run within the real-time constraints and this is going to increase the video and audio traffic in the future Internet. Servers for movies and television programs may become widespread in the near future. As the high definition TV emerges and becomes popular, the distinction between a computer and a television may diminish. Every television set may become an IPv6 net endpoint.

The major requirements of this segment of applications are the real-time guarantees on delay bounds and available bandwidths. The communication medium is likely to be a high bandwidth one, like fiber optic cables and IPv6 should be as efficient as possible in utilizing the bandwidth. Again, this being a consumer market, support for accounting and policy-based routing will be welcome. Since we are talking of almost every home on the globe having an Internet host, the strain on address space and routing is going to be enormous and the IPv6 should be able to cope. See [Vec94] for more details.

Smart devices

The concept of intelligent homes with all the everyday devices, like lighting, heating and cooling equipment, being controlled by a central computer is catching up fast. But today the devices are being monitored by control hardware that is analog and is proprietary and expensive. quite proprietary and expensive. The idea is to make all these instruments IPv6 hosts and work out a generalized solution.

At the moment, the requirements put forward by this kind of applications are not very clear. The packets may have to use the same power cable that feeds power to the instrument. Some new encoding techniques which allow this are being researched. IPv6 should be extendible to incorporate emerging solutions in this area.

2.3 Transition

The Internet as mentioned earlier, has grown too big making the transition from IPv4 to IPv6 on a “flag-day” virtually impossible. The major reason why many protocols, that attempted to replace TCP/IP stack, could not succeed was the lack of proper transition plan. So a white paper on transition requirements was prepared by Carpenter [Car94] and is summarized below.

IPv6 should be designed with this necessity for phased deployment in mind. Any node should be able to get upgraded to IPv6 in isolation i.e.,i.e., there should be no dependencies for the deployment of IPv6 in either a router or a host. All the old applications written for IPv4 should run on IPv6 hosts too. Communication between any pair of machines, at any protocol level should not become impossible i.e., TCP on an IPv4 host should be able to talk to TCP on an IPv6 host.

The routing infrastructure also is not going to change over night. So the IPv6 should be able to use the existing IPv4 routers during the initial period of the transition. Similarly the IPv4 only hosts should be able to use the predominantly IPv6 routing, during the later stages of the transition. Mechanisms for tunneling of IPv4 packets over IPv6 networks and vice versa should be worked out. An easy way for translating IPv4 addresses to and from IPv6 addresses will come in very handy in automating this kind of tunneling.

Header translation should be totally avoided because, besides being a performance overhead, it cannot be automated completely. This makes it mandatory for all the IPv6 hosts to support IPv4 also. This requirement can be withdrawn at the end of transition.

All the utility programs and services, like DNS, should work with both IPv6 and IPv4. The socket API should change as little as possible.

2.4 Overview of IPv6

It can be claimed that IPv6 will meet almost all the requirements that have been set for it. In this section, we look at the major improvements and new features.

2.4.1 Simple header formats

The IPv6 header is much simpler that that of IPv4. Considering the address size (128 bits), the IPv6 header is very small (40 bytes). All the IPv4 header fields, that are not effectively used have been deleted. As far as possible bit fields have been avoided. Fields are aligned to facilitate extraction of maximum performance out of 64-bit architectures.

Separate extension headers are designed for logically independent functionalities like routing, authentication, security, etc. The encoding and processing of options has been improved. IPv6 distinguishes between end-to-end and hop-by-hop options and they are placed in different headers. This allows routers to skip options that are not meant for it. It improves the router performance significantly. These extension headers are kept between the IPv6 header and the transport header. Not all packets need to have all the extension headers, but each header indicates what the next header is, forming a singly linked list.

The default actions needed for a particular option can be encoded in the option itself which makes it very easy to add new options. Similarly, the concept of extension headers allows addition of new functionalities in a relatively easy way. Unlike IPv4 the total length of the options is not limited to 40 bytes.

2.4.2 Addressing architecture

The addressing architecture of IPv6 is defined in [Hin95]. The main change has been the in the length of the address. The choices were 64 and 128 bits. Huitema in [C94] argues that 128 bits of address length is a safer choice and will provide with more than enough addresses. So the IPv6 addresses are standardized at 128 bits. As noted earlier, the actual efficiency of address allocation suffers, if we were to allocate addresses in such a manner to allow aggregation of routing information. But Huitema's calculations show that, even in the most pessimistic case we will have a total of $8 \text{x} 10^{17}$ addresses. The most optimistic estimate puts the total addresses at $2 \text{x} 10^{33}$. The preferred textual representation of an IPv6 address is of the form x:x:x:x:x:x:x:x. Each x designates the hexadecimal representation of a 16-bit quantity. Continuous run of zeros can be abbreviated by a "::". But it can appear only once. Embedded IPv4 addresses can be written in the well known dotted decimal notation, like x:x::a.b.c.d (IPv4 addresses always occupy the last 4 bytes.)

The concept of an "address prefix" or simply "prefix" is very widely used in IPv6. Any non zero number of contiguous bits from the left-most bit form a valid prefix. There can be many valid prefixes for a given address. The particular one can be identified by providing additional information, such as the length of the bit sequence. A popular notation is `xx::yy/len`, where len gives the length of prefix in bits.

There are various types of IPv6 addresses like, unicast addresses, local-use addresses, multicast addresses, etc. They are identified by a field, formed by a variable number of leading bits in the address, called the Format Prefix. various format prefixes that have been assigned can be found in [Hin95]. Use of 15% addresses are decided now, and the rest 85% are reserved for future use.

▮ *Unicast addresses*

There are a variety of unicast address forms in IPv6, like IPv4-compatible addresses, provider-based unicast addresses, addresses for local use, IPX hierarchical addresses, NSAP addresses, etc. Different format prefixes have been assigned for all these types. A few main categories are explained below.

- Provider-based unicast addresses
  These are the normal globally unique addresses, used in unicast IPv6 datagrams. The functionality of this is similar to the IPv4 addresses under CIDR. Since IPv6 is also going to use the underlying concept of CIDR for route aggregation, the allocation should be done in an orderly fashion. Such a plan is outlined in [Rek95]. The address format is shown in Figure 1

- Local use addresses
  These addresses are not unique across the globe. There are two kinds of local use addresses: link-local and site-local. The formats are shown in Figure 1. As their names indicate, link-local and site-local addresses are guaranteed to be unique only on the link and the site (organization) respectively. Link-local designed to be used in auto address configuration, neighbor discovery, etc. Site-local addresses are meant for sites which are not yet connected to the Internet. When they connect, site-local addresses can be changed to globally unique unicast addresses by replacing the prefix with the one supplied by the provider.

- Addresses with embedded IPv4 addresses
  These addresses are meant for providing an easy way of translating IPv4 addresses to and from IPv6 way addresses This is useful in implementing the automatic tunneling of IPv6 packets over IPv4 and vice versa more easy. There are two kinds of addresses with embedded IPv4 addresses: IPv4-compatible IPv6 addresses and IPv4-mapped IPv6 addresses. The formats are shown in Figure 1. The IPv4-compatible addresses identify nodes which support both

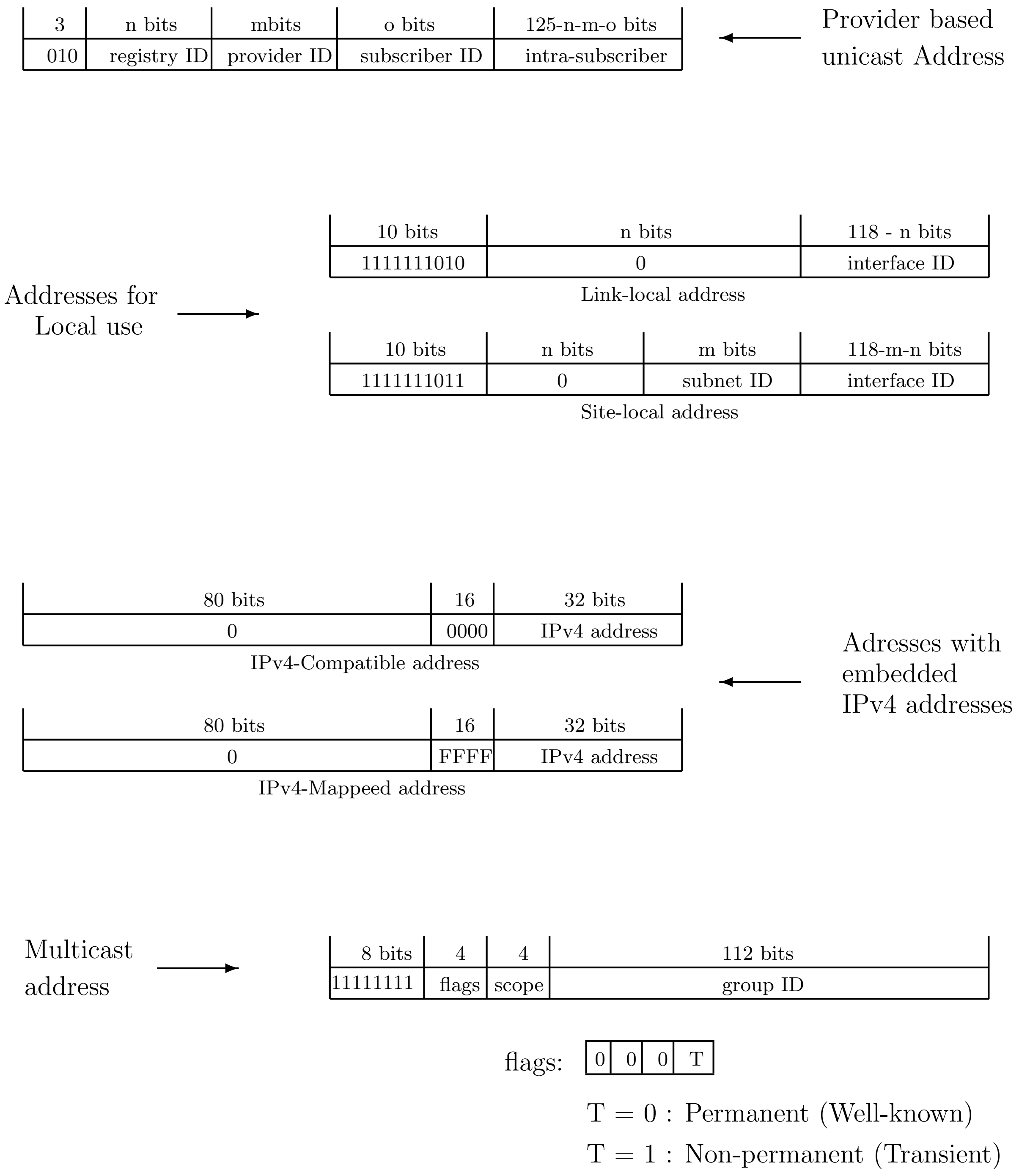

Figure 1: IPv6 Addressing Formats

IPv4 and IPv6 and employ tunneling. The IPv4-mapped addresses are used nodes that are "IPv4 only" and does not know IPv6. IPv4-compatible addresses also help in minimizing the effort in changing over to IPv6. When an IPv4 host is upgraded to IPv6, no new IPv6 address needs to be assigned. The node can start using the corresponding IPv4-compatible IPv6 address.

- Anycast addresses
  These are essentially unicast addresses that are assigned to more than one interface, possibly on different hosts. The property that distinguishes an anycast address from a unicast address is that, a packet sent to an anycast address will be delivered only to the nearest interface having that address, (by th measure of routing metric). Anycast addresses are syntactically indistinguishable from unicast addresses. When a unicast address is assigned to multiple interfaces making it an anycast address, the nodes should be explicitly configured to know that the address is assigned elsewhere too. Anycast addresses can be used to implement policy-based routing, to locate a popular service, to locate a default router, etc.

▮ *Multicast addresses*

Multicast addresses are distinguished from unicast addresses by a format prefix of "FF" in the highest order octet. (Figure 1) There is no special broadcast address in IPv6. Broadcast is treated as a special case of multicast. The four higher order bits of the octet that follows the format prefix, have been designated as flags, of which three are reserved for future use and the remaining one is used to denote the type of the multicast group: permanent or transient. The permanent multicast groups are going to be well known and will be used to address popular services. For example, all the NTP servers can be given a well known multicast address, which can be used to multicast time updates among themselves. These well known multicast groups are expected to make the application development simpler. Transient groups will be created dynamically.

The other interesting feature about IPv6 multicast addresses is its scope field. This field, in case of transient groups, encodes the geographical spread of the group but is not interpreted for permanent groups. This is very helpful in making the multicast routing scalable, as we can use this information to eliminate all unnecessary membership advertisements. For example: if the scope field of a multicast address identifies it as a site-local group, the routers at the site-boundary will refrain from advertising the membership for this group, to outside world. The transient multicast groups are meaningful only within their scope. Two groups with same group-id but different scopes, are treated as two different groups, even if one scope encompasses the other i.e., a link-local group is different from a site-local group with the same group-id. Similarly a transient group irrespective of its scope field bears no relationship with a permanent group with the same group-id.

2.4.3 Routing

The basic routing remains same as IPv4 routing under the CIDR. All the major routing protocols that support classless routing will work with a few, simple changes, like changing the address length to 128 bits from 32 bits.

IPv6 introduced a simple, but powerful routing feature in the form of a new routing header to be added after the IPv6 header in a packet. This header is identified by a value of 43 in the immediately preceding header. It allows the sender to choose a variant of router behaviour by setting the routing data appropriately. This type is carried in the RoutingType field. The intermediate nodes will check this type, and interpret the rest of the fields accordingly. On encountering a non-empty routing header of unknown type, the routers are required to send an ICMPv6 parameter problem message back to the source. Like all other extension headers, this header too contains the standard fields, NextHeader and the HdrLength.

Type-0 routing header is defined in [Dee95]. Type-1 [SA95] is designed to support the Source Demand Routing Protocol (SDRP) which allows source initiated routes to augment the routes selected by the routing protocols.

The functionality of Type-0 routing is similar to the IPv4 Source Routing option. It contains a series of intermediate hosts the packet should visit, en route to its destination. But unlike IPv4 source option, choice of loose/strict routing is not same for all the hops. It can be set for each hop independently. This choice is encoded into a bitmap of length 24. (This restricts the maximum number of intermediate nodes.)

Unlike the IPv4 source routing this will not become a security threat since any attempt of abuse will be detected by the authentication mechanism.

This routing functionality has made it possible to support several features. Few examples are explained below.

- Mobility can be supported with ease, as most of the solutions for the mobility problem at the network layer level assume a powerful and robust source routing. Further, IPv6 makes it mandatory for the end hosts, to reverse any source routes they receive in the routing headers, while replying.
- Provider selection can also be realized with the help of this option. We can make a host send all its packets through a particular provider's infrastructure, by keeping the anycast address of the corresponding routers, at an appropriate place, in the routing header.
- QoS based routing will be easier to implement. Resource reservation protocols, like RSVP, can choose paths that support the required QoS better, than the paths suggested by the normal routing protocol. Source routing can make all the packets belonging to a particular "flow" follow a given path. For example, we can make all the packets of a video conference take a path that is entirely optical fiber, and direct all the packets of an FTP session over normal phone lines.

2.4.4 QoS Handling

IPv4 supports only two simple QoS classes: low delay and high throughput. even these two classes are not supported by typical routers. They processed the packets essentially in FIFO order. To implement support for more sophisticated QoS classes, the routers should use some sort of a priority queuing, instead of simple FIFO.

▮ *Priority*

The router should be able to find out what kind of data a packet is carrying, before it can decide upon the special treatment or the priority, for that packet. This information is being encoded in the ToS field of the IPv4 header, but as mentioned above, this allowed only two values to choose from, providing a crude classification. The IPv6 allocated 4 bits for denoting the priority class, allowing a much finer classification with 16 priority classes. They are split evenly between traffic that backs-down as a response to congestion and the traffic that does not respond to congestion. TCP traffic is an example for the first kind, and UDP traffic is is an example for the second kind. There is no relative priority defined between these two main subclasses. Different priority classes defined can be found in [Dee95].

▮ *Flow ID*

In some cases, knowing just the priority class may not suffice. For example, some classes of traffic, like constant bit rate audio can be served better if, we can make some pre-assignment or reservation of resources. Reservation protocols like RSVP can reserve resources all along the path, but there should be a way to identify all the datagrams that should be allowed to use these reservations. So all the packets that are going to utilize these reservations, should present some identification to the routers. In other words we should have a way to associate a datagram to a logical "flow". A flow can be defined as a sequence of related packets being sent from a particular source to a particular destination (can be multicast) and need the same type of service from the routers. The service they desire can be conveyed to the routers, by means of a set-up protocol like RSVP.

IPv4 being a pure datagram protocol, does not give any facility to mark the related packets. IPv6 provides for this by means of a 24 bit flow-id in the IPv6 header. This flow-id is generated pseudo randomly in a source so that the intermediate routers can use hashing effectively. The flow-id is going to be unique only within the source, so the routers should form a unique key by either prefixing, or suffixing the source IPv6 address to the flow-id. The issues in flow-label usage are discussed in [Par95].

IPv6 also tries to use the flow-label concept in improving the router performance. It makes it mandatory for the source to have all the packets belonging to a particular flow carry identical IPv6 header, and any other necessary extension headers like, routing header and hop-by-hop header, etc. This makes it possible for the routers to "save" the computation done for one packet in the flow, and reuse it for the subsequent packets. The routers are free to check for any changes in any of the headers and send an ICMPv6 parameter problem message back to the source.

The routers should detect a "dead-flow" too. Flows can become inactive, because of changes in routing or because of the session getting concluded. Both of these cases can be taken care, by associating a timeout with the state entries the router caches. The hosts should not reuse the flow-ids until they are sure that the caches in all the routers have been flushed.

A value of zero for flow-id specifies the non-existence of a flow-id. A router on receiving a flow-id for which it cannot find a cache entry, should process the packet as if the flow-id were zero and create a new entry for it in the caches. Any standard cache management policies like LRU should work here too.

2.4.5 Miscellaneous features

▮ *Security and authentication*

The IPv6 provides framework for the implementation of authentication and security separately.

The Authentication Header (AH) provides for conveying the information relating to authentication. Though this header is designed to be algorithm independent, the use of keyed MD-5 algorithm is advocated. Authentication allows the destination to verify and be sure about the sender as well as the integrity of the datagram.

The other mechanism is called the Encapsulating the Security Payload (ESP). This provides the confidentiality. The standard DES encryption algorithm will be used to encrypt the message.

No specific key distribution algorithm is made standard as yet, but the AH allows both host-to-host and user-to-user keying.

These two mechanisms open up a lot of new possibilities like, corporate virtual networks that use public data carriers, electronic commerce, etc.

▮ *Path MTU discovery*

Fragmentation has been considered harmful for the following reasons.

- It increases the processing that needs to be done at intermediate routers.
- Fragmenting the packet into a number of smaller units actually increases the probability for the packet to get lost, since loss of even a single fragment effectively make the whole packet use-less.

- Since the fragments can be reassembled only at the destination, once the packet gets fragmented, these smaller units travel up to the destination as independent packets and they will not be able to utilize the higher MTUs available en route.

The IPv6 prohibits fragmentation at the intermediate routers. So the source tries to find out the MTU of the path and fragments the packet, if necessary, at the origin itself. The exact method of doing this is described in [CE90].

Chapter 3

Networking subsystem of Linux

In this chapter we explain about the general principles around which the Linux networking subsystem is built. We also describe how the network buffers are managed and how the different logical entities like device layer, protocols and the protocol families interact and co-exist.

3.1 General design principles

Linux design has been influenced by the object-oriented technology to a great extent. Almost all logical modules are treated as objects with well defined interfaces. For example, each file system is an object, with methods to read and write ordinary files as well as directories. Similarly each network device is also viewed as an object with methods to send, receive packets, and to perform ioctls etc.

The general design and algorithms used are generally on the lines of those described in the text books [Bac86] and [LMKQ89].

Though the Linux kernel is fully coded in ANSI C, its object oriented design manifests itself in the implementation. As mentioned earlier, each file system is implemented as a self contained object and socket-layer is also treated as a special file system under the standard Virtual File System (VFS). The abstraction of a method is implemented with the aid of function pointers. Each of the file systems implements a standard set of functions to handle all the operations on a file. Pointers to all these functions are saved in a per-file system structure called `file_ops`, and as part of opening a file, a pointer to this structure is saved in the system wide file-table entry of the file being opened. VFS transfers all the system calls on a file descriptor, to an appropriate function found in the `file_ops` structure. If a pointer for the handler function is null, VFS returns an appropriate error to the application. Socket-layer as a file system, makes available handlers for only `read` and `write` system call.

On similar lines, each protocol family "exports" to the socket-layer, a well defined set of operations, in the form of function pointers, which serve as entry points to the protocol code. This set is defined as a structure named `proto_ops`. As a part of boot-up initialization, each protocol family fills in a `proto_ops` structure and registers it with the socket-layer. Socket-layer while creating a new socket chooses the protocol family and saves a pointer to its `proto_ops` struct, in the newly allocated socket structure. Subsequent operations on this socket will result in calls to appropriate function chosen from the `proto_ops` struct found in the socket.

3.2 Boot time initialization

The processing in the kernel can be divided into two main types: bottom half and upper half. The term "bottom half" is used to denote the part of the kernel that runs on clock interrupts. Similarly, the "upper half" refers to the processing that's done when a system call is made by the application.

During the boot time, after the kernel is loaded into the memory, initializing functions of all the modules are called. This is done just before the init process is forked. The function `start_kernel` is responsible for this initialization and as a part of starting the kernel, it calls `sock_init` to initialize the networking subsystem. The function `sock_init` in turn sets up all the networking devices, bottom half handler and also calls the initializer function for each of the protocol families configured. For this, it steps through the `protocols` list and calls the initializer for each of the families. This list is an array of structures of type `net_proto`, with two fields: an ascii string denoting the name of the family and a pointer to the initializer function. This is built statically by the compile-time initialization.

A typical protocol family initializer apart from taking care of the family specific initialization, also exports a set of pointers to its handler functions to the socket-layer. Socket-layer maintains a table, called `pops`, with each entry holding information about exactly one protocol family, in two fields : the protocol family identifier and and a pointer to its `proto_ops` structure. The family initializer, on being invoked from `sock_init` prepares a `proto_ops` structure and passes it on to the socket-layer, by calling the `sock_register` function. The `sock_register` function finds an empty slot in the `pops` table and fills it in with the supplied `proto_ops` structure. This list is going to be searched when a socket is being opened with `socket` system call.

3.3 Network buffer management

Linux network memory management is built around the structure called `sk_buff`. It is essentially a piece of memory for data, with a header for control information. Each `sk_buff` holds exactly one packet and unlike the BSD mbufs, can be of variable size. Because of the fixed size of mbufs, packets are often stored as chains of mbufs. This allows dynamic shrinking and expanding of the packet, but operations even on a single packet become quite complicated. Instead Linux uses a single linear buffer to store the entire packet. Since we cannot expand the allocated linear buffer, with the growing packet size, we need to take the worst case packet length into account, before allocating an `sk_buff`. Similarly parts of an `sk_buff` cannot be released back, as we strip off the headers. Though these two factors result in slight wastage of memory, the linear buffers are preferred over chains of `mbufs`, because it makes the packet manipulation very simple. The `sk_buffs` can also be linked into doubly liked lists to form chains of packets, but a packet never spans over two `sk_buffs`.

The other difference is that the "header" that has the control data is kept at the end of the allocated buffer instead of at the beginning. This is done to exploit the underlying memory allocation routine `kmalloc`, which returns always an address that is a power of two and this helps in minimizing the memory wastage while aligning the start of the packet.

A collection of general purpose utility routines are made available to operate on these buffers. The three main types are: functions that allocate and free `sk_buffs`, functions that help in manipulating the data part of an `sk_buff` and the ones that help in organizing lined lists of `sk_buffs`. Since most of the `sk_buff` operations are done at interrupts, these routines are made atomic.

There are four fields in the `sk_buff` header that mark various points in the data area. The variables `head` and `end` mark the beginning and end of the allocated buffer respectively. Similarly the `data` and `tail` mark the beginning and the end of usable data. The length of the usable data is kept in the `len` field.

`skb_alloc` allocates a new `sk_buff`. Space can be reserved at the head of the packet using `skb_reserve`. To add data at the end, we should first advance the `tail` pointer and also adjust the `len`. The function `skb_put` does exactly this. Data can be pushed into the space reserved at the head, using the `skb_push`. `skb_pull` and `skb_trim` remove data from the head and tail of the packet respectively. Finally `kfree_skb` frees an `sk_buff`.

Two simple code fragments are shown in Figure 2.

```
struct sk_buff *skb;

skb = alloc_skb(len,GFP_ATOMIC);
if(skb == NULL)
   /* Drop the packet */
else {
   skb_reserve(skb,hdrlen);
   memcpy(skb->data,data,datalen);
   skb_put(skb,datalen);
   skb_push(skb,hdrlen);
   memcpy(skb->data,hdr,hdrlen);
}

          Assembling a Frame
```

```
struct sk_buff *skb;

skb = skb_dequeue(list);
if(skb == NULL)
   /* No packets - return */

else {
   process_hdr(skb->data);
   skb_pull(skb, hdrlen);
   process_data(skb->data);
   kfree_skb(skb, FREE_READ);
}

          Disassembling a Frame
```

Figure 2: Example usage of sk_buff support routines.

3.4 Network Device Layer

The network layer protocols like IPv6 directly rest on the device layer in the standard protocol hierarchy. They need to use the services of the device layer to send and receive packets. The same general theme of object-oriented design can be found here too. Each device is treated as an object and the other kernel modules access it via a device description structure, called device. This structure holds pointers to various functions that implement the operations on the device. For example every device supplies function pointers to initiate the transmit operation, to build MAC header etc. Along with these function pointers, the device structure, carries several variables describing various parameters and the device state.

Any network module that is interested in receiving particular type of packets should first inform the device layer, using the function dev_add_pack. This function takes in a description of the packet in the structure packet_type, and saves it in a table that bottom half has access to. This structure contains protocol identifier constant defined in [JJ94] and a pointer to a handler function. The protocol modules usually supplies this information to the device layer as a part of its boot time initialization.

When the device receives a packet, the device driver allocates an `sk_buff` and copies the packet into it. Then it extracts the protocol ID from the MAC header and saves it in the control header of the `sk_buff`. This is done because the the function that actually feed the upper-layer protocols with these packets, is a generic one and will not know to decipher various kinds of MAC header. It then pulls the MAC header off the packet and queues it up in a system wide `back_log` queue for further processing by the bottom-half handler.

The bottom half handler which is run at regular intervals, on a clock interrupt, examines this `back_log_queue` and if there is some packet there to be despatched to upper-layers. If there are any, for each of the packets, it tries to match the protocol ID found in the control data of the `sk_buff` with one of the entries in a table that maps the protocol ID to a pointer to a receive function. This table is built incrementally, by calls to `dev_add_pack` by various protocol modules. On a successful match, it calls the receive function with the packet (`sk_buff`) as a parameter.

A single interface function `dev_queue_xmit` is provided, for all the devices, to send packets. First the routing mechanism selects an outgoing interface. Then the packet is prepared with all the headers. This packet is then passed on to the device layer for transmission, by calling the `dev_queue_xmit` function. This function takes: a pointer to the `sk_buff` and a pointer to the `device` structure of the outgoing device as parameters. The protocol module can make sure that the device is up and running by examining the `flags` in the `device` structure. (These flags are set by the system administrator through the `ifconfig` command.)

The `dev_queue_xmit` first checks whether the device can accept a new packet right away. The `busy` flag is set in the device structure, if the device is busy in transmitting a previously queued packet. If the device is not busy, it calls the function pointed to by the `hard_xmit` field in the `device` structure, which invokes the actual hardware handling routines in the device driver. On the other hand if the device is found to be busy, the packet is queued up for later trying. The bottom half when run on a clock interrupt, attempts to send all such pending packets.

3.5 Adding a new protocol family

The major steps in adding a new protocol family are

1. Define a new protocol family ID in the file `/usr/include/linux/socket.h`. Change the `PF_MAX` constant also, if necessary, to an appropriate value, so that it is larger than the largest value given to any protocol family. The code in networking subsystem assumes that it always holds such maximum.
2. Change the NPROTO constant in the file `/usr/include/linux/net.h`, if necessary, to reflect the maximum number of protocol families that can be supported. This is used to statically define lists like the above mentioned `pops` and `protocols`.
3. Define the new protocol specific address structures and other necessary constants.
4. Write handler functions for the operations that the new protocol family is going to support. The signatures of these functions can be taken from the definition of the `proto_ops` structure. Initially they all can simply return an error like ENOTSUPP after printing some debug messages.
5. Write a handler function for the incoming packets from the device layer.
6. Write an initializer function that takes no arguments and returns a void. It should do the following.
   - Prepare a `proto_ops` structure, with pointers to all the handler functions. Make sure that the pointers to undefined handlers are set to NULL. Pass this structure to the socket layer, which reads it and makes an entry in the `pops` list, for the new protocol family.

- Inform the device layer about what protocol ID the packets the the new protocol module is interested in, should carry, along with a pointer to the handler function. It should prepare the `packet_type` structure and add it to the system wide table the bottom half uses.
- Perform other miscellaneous things like initializing various tables and chains, assigning default or startup values to various configurable parameters etc.
- Should initialize the member protocols by calling their initialize functions in sequence.

7. Decide upon a new name to be given to the family and edit the portion of the code that statically initializes the `protocols` list to include an entry for the new protocol. This causes the `sock_init` to initialize our new protocol family too during the boot time.
8. Change appropriate makefiles to get the new files also compiled properly and make the new kernel.

In the next chapter, we will explain how we added the INET6 family following these steps.

3.6 Member protocols in a family

The individual protocols with in a family are again independent objects with methods for various operations like read and write. They are accessed by the methods of the family which inturn are accessed by the socket layer. The handlers at the family level do every thing that is common to all the member protocols and is specific to the family. In the same lines the handlers at the protocol level do every thing that's specific to the protocol. These form the lowest level in the hierarchy of functions that implement the system calls.

Initializing internal details of the protocol is done by the per-protocol initializer, which are called by the family initializer. For this the family initializer uses a list of protocols which is constructed by compile time initialization.

A typical protocol initializer does the following.

- Initialize various internal tables, assign default values or initial values for configuration parameters etc.
- If the protocol is interested in receiving packets directly from the device layer, it should inform the device layer about what protocol ID the packets it is interested in should carry, along with a pointer to a handler function. It should prepare a `packet_type` structure and add it to the system wide table for the use of the bottom-half
- Should set up the linkages with other protocols if necessary.

In the following chapter we explained how the raw IPv6 protocol is added to the INET6 protocol family.

Chapter 4

Building the skeleton

In this chapter we will explain the design and implementation of the basic framework for the new IPv6 protocol module. Linux provides the standard 4.3BSD style Application Programming Interface (API) for the networking modules. We explain the changes and additions to this API that are needed to support IPv6. We also, explain how the new module fits into the kernel, and gets initialized etc.

4.1 BSD API

The API provided by the 4.3BSD, more popularly known as the socket interface, became the de facto standard in recent times. Since the applications written for this API enjoy a high degree of portability, Linux also provides an API that closely resembles it. The main components that form the API are: core socket layer functions, address structures, various constants, various socket options, ioctl commands and address conversion functions. No standard has emerged yet for how these components should be adapted to IPv6. The current status can be found in [ESJ96].

Building the API corresponds to the first three steps out lined in Section 3.5.

The core socket layer functions are inherently very much adaptable for different protocols, as they require the applications to cast the protocol specific addresses into the generic `sockaddr` structure and also takes the length of the protocol address structure as one of the parameters. The only restriction placed on the address format is that it should carry the Address Family identification (the `AF_*` constants) in the first two bytes. So this component of the API should serve its purpose without any changes.

4.1.1 New structures and constants

The IPv4 address is only 4 bytes long and this leaves 8 bytes of data unused in its protocol specific socket address `sockaddr_in`. But this is not sufficient to hold the 16 byte long IPv6 address. So it is mandatory that we define a new address family with new address structures.

A new Address Family constant `AF_INET6` and a new Protocol Family constant `PF_INET6` are defined in <`linux/socket.h`> along with the other address and protocol families.

```
#define    AF_INET6    10
#define    PF_INET6    AF_INET6
```

Since none of the standard C types can hold the 128 bit long IPv6 address, a structure `in6_addr` is defined in <`linux/in.h`>, The `s6_addr` field in this structure, is an array of 16 unsigned characters in which the IPv6 address is stored in network byte order.

```
struct in6_addr {
    unsigned char s6_addr[16];      /* IPv6 address */
};
```

This `in6_addr` is used in defining the IPv6 specific socket address `sockaddr_in6`. IPv6 applications like the IPv4 applications should include <`linux/in.h`> to get this definition.

```
struct sockaddr_in6 {
    unsigned short      sin6_family;    /* AF_INET6 */
    unsigned short      sin6_port;      /* Transport layer port # */
    unsigned long       sin6_flowinfo;  /* IPv6 flow information */
    struct in6_addr     sin6_addr;      /* IPv6 address */
};
```

This address structure conforms to the standard convention by over-laying the `sin6_family` field on the `sa_family` field of the generic socket address `sockaddr`. `sin6_port` is exactly same as the `sin_port` in the IPv4 socket address `sockaddr_in`. The IP v6 address is carried in `sin6_addr`, which by virtue of its position in the structure is aligned on a 64 bit boundary. This kind of alignment is expected to result in improved performance on 64 bit architectures which are fast becoming the standard.

IPv6 applications can control the values the flow-id and the priority fields, in the IPv6 headers of the datagrams they generate. These are input to the IPv6 module via the `sin6_flowinfo` field. This 32 bit field carries both the 24 bit flow-id and the 4 bit priority field. This field contains the flow-id and the priority found in the incoming datagram, when the `sockaddr_in6` is received from a system call like `recvfrom`. As of now it is not very clear how the unique flow-ids are going to be generated and used. There are also arguments in favour of replacing this single integer with a structure or a union with separate fields for flow-id and priority. Right now, in our implementation we do not include any mechanism to supply a unique flow-id. We do not even have any sanity checks for the application supplied flow-ids, but will copy whatever is found in the destination `sockaddr_in6` into the IPv6 headers.

In IPv4 domain, the INADDR_ANY allows the applications to specify a system selected address. Similar facility in IPv6 cannot be provided in a straight forward manner since unlike IPv4, IPv6 address is defined as a structure. So we need to supply a macro IN6ADDR_ANY_INIT and a global variable in6addr_any to allow compile and run time initializations respectively. Application programs can access these by including the <linux/in.h>. Similarly, a macro IN6ADDR_LOOPBACK_INIT and a variable in6addr_loopback are also defined in the same file, to achieve functionality same as that provided by the constant INADDR_LOOPBACK in IPv4 domain.

```
/*  Use these macros only for COMPILE TIME initializations. */
#define IN6ADDR_ANY_INIT      {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
#define IN6ADDR_LOOPBACK_INIT {0,0,0,0,0,0,0,0,0,0,0,0,0x7f,0,0,0x01}

/*  Use these variables only for RUN TIME initializations. */
struct in6_addr  in6addr_any      = IN6ADDR_ANY_INIT;
struct in6_addr  in6addr_loopback = IN6ADDR_LOOPBACK_INIT;
```

4.1.2 Address conversion functions

The address conversion functions of IPv4 inet_addr and inet_ntoa assume the 4 byte long IPv4 addresses and their functionality can not be extended transparently to handle a different protocol family. More generic address conversion functions inet_pton and inet_ntop, which accept the protocol family also as a parameter are defined. Another improvement over the old functions is that these new functions are guaranteed to be thread-safe, as they do not employ any static storage. These functions are eventually going to be part of the standard C librarylibc, but right now they are kept in a separate library - the libaconv. Manual pages for these functions are available on-line and can be referred for exact syntax and semantics.

4.1.3 Socket options

The main socket option that is being proposed is to change the type of addresses the socket operates with, from IPv6 to IPv4 dynamically. This is necessary in cases where an IPv6 aware application forks and execs an application that knows only the IPv4. Since the child inherits all the open sockets, in the above mentioned case, the IPv4-only child process inherit the IPv6 sockets of the parent also. Then the child can get confused, by the "strange" IPv6 addresses returned by system calls by such inherited IPv6 sockets. So the parent should make all the IPv6 sockets return the IPv4 addresses before forking and execing the older applications.

However at the moment, the semantics for this socket option are not very clear and we do not support it. There are a host of multi cast related setsockopts being proposed, but as our implementation supports only unicast these too are not relevant for us.

The two important socket options we implemented are:`IPV6_HDRINCL` and `IPV6_SRCROUTE`. Both of them operate at the IP level (`SOL_IP`).

Applications can choose to take the responsibility of constructing and consuming the IPv6 headers, when a raw IPv6 socket is being used. To do this, they have to set the `hdrincl` flag in the `socket` structure using the socket option `IPV6_HDRINCL`. This option takes an integer as a parameter. A non zero value for this parameter sets the `hdrincl` flag and a value of zero resets it. If this flag is on, the raw IPv6 module does not attempt to construct any headers, but assumes that headers have been constructed by the application itself. It will simply, route, build the MAC headers and forward the packet to device layer. Similarly on receiving a packet, the IPv6 module queues it for a raw IPv6 socket with the `hdrincl` flag set, without stripping off the IPv6 headers. This facility is very much useful to generate test packets and aids in debugging.

The other socket option `IPV6_SRCROUTE` sets the source route associated with a socket. This is operation is allowed only once on any socket in the same lines as `connect`. This is done to avoid problems that can occur when the socket is being accessed by more than one process. Consider the scenario in which the first process sets up the source-route on a datagram socket and gets preempted. Now suppose the second process gets scheduled and sets up another source-route. Then the first process has no way to know, that the route it had set up is no longer in effect. Similar problem would occur had the `connect` been allowed multiple times. So we made this setsockopt a one-time operation on the socket.

But this prevents the application from using the very powerful feature of the datagram sockets i.e., the ability to multiplex traffic to different destinations from the same socket. The problem of having a separate system call only to specify the source route is that it actually splits the logical operation of "sending data to a destination *via* a route." into two system calls: identifying a route and sending the data. But these two operations are not logically separable and so we should do both the identification of source-route and sending data via that route in a single system call.

To achieve that we should either develop a new system call like `sendtovia` which accepts the source route also as a parameter, or make the `send to` accept the source route too. Since the `send to` accepts the length of the address as a separate parameter and takes in a *pointer* to the destination socket address instead of the *structure* it self, it is easy to pass a source route as an array of socket addresses with an appropriate length. Besides providing a way to specify the source route once for the life time of the socket (the `IPV6_SRCROUTE` socket option), we allowed the route to be passed through the sendto system call too.

We also provided several ioctl calls to manipulate various kernel tables specific to IPv6. These will be explained later in the report.

4.2 Initializer for INET6

In this section we describe how the IPv6 module fits into the larger framework of the kernel and how it gets initialized.

As mentioned earlier, the `sock_init` function initializes all the configured protocol families. A list of such families is built by initializing the `protocols` list at compile time. A new entry is added to this list specifying the protocol family as `INET6` and the initializer function is `inet6_proto_init`.

```
struct net_proto protocols []  =  {
  .
  .
  .
{"INET", inet_proto_init},
{"INET6", inet6_proto_init}};
```

During the boot up, the network subsystem initializer function `sock_init` steps through this list and calls `inet6_proto_init` which initializes the IPv6 module, by taking the following steps.

- Prepares a `proto_ops` structure `inet6_proto_ops` with the protocol family ID `(AF_INET6)` and pointers to various functions that serve as entry points. Calls the `socket_register` with `AF_INET6` and `inet6_proto_ops` as parameters. This gets the `inet6_proto_ops` into the global `pops` list.
- Initializes various caches tables and counters of the module.
- Call the initializers of the member protocols. As of now we support the raw IPv6. As more and more protocols like TCP/IPv6 and UDP/IPv6 are added to the family, their initializers should also be called along with the `rawv6_init`.

4.3 Initializer for raw IPv6

IPv6 is a member of INET6 protocol family and can be accessed through a raw socket, so we refer the simple IPv6 as raw IPv6 on the same lines as raw IP.

The initializer `rawv6_init` on being called, from `inet6_proto_init` does the following things.

- Raw sockets by default receive all the IPv6 packets destined for the host. So every packet received is copied to all of them. This requirement simplifies the organization of raw sockets and a simple linear linked list is sufficient. Actually this is not the list of the `sockets`, but is a list of all the PCBs of the raw v6 sockets. The initializer initializes this list `ipv6_sk_list` to null. The `socket` system call adds PCBs to this list.
- Prepares a `packet_type` structure `ipv6_packet_type` for IPv6 packet and registers it with the device layer. The two important fields of the `packet_type` are: a protocol ID the packet carries in its Ethernet header and a handler function to be called whenever such a packet is received. For IPv6, the protocol ID is defined as `ETH_P_IPV6` with a value 0x86DD in <`linux/ifether.h`>. The handler is `ipv6_rcv`. The tables of the device layer are set up with this information when the call to `dev_add_pack` is made with `ipv6_packet_type` as a parameter.
- Creates two special `proc` files `ipv6` and `ipv6_route` in `/proc/net` directory. These provide file system interface for various kernel tables specific to IPv6 module. These are explained in more detail in Section 7.1

Having seen how the INET6 protocol family, with raw IPv6 as the constituent protocol, is fitted into the kernel we will move onto the implementation of various system calls in the next chapter.

Chapter 5

System calls

In this chapter we implementation of the system calls and also give a detailed picture of the processing done on the clock interrupts.

5.1 Socket

Socket system call is used to open a socket. The standard interface to `socket` system call takes three parameters: Address/Protocol family, type of the socket and the protocol to be used. To open an IPv6 socket the first parameter should be `AF_INET6` or `PF_INET6`. As we support only the raw IPv6 as of now, the second and third parameters should be `SOCK_RAW` and a zero respectively. So, to open a raw IPv6 socket the application should use the `socket` system call in the following way. `socket(AF_INET6, SOCK_RAW, 0)`.

The system call wrapper in the standard C library after setting up the parameters either on the stack or in the registers, traps the kernel. The trap handler runs a generic system call handler, depending on the system call number retrieves parameters from the user space if necessary, and calls an appropriate function. In case of `socket` system call, the control gets transferred to `sys_socket` which takes the following steps.

- Does sanity checks like, making sure that the supplied socket type is valid etc.
- Searches the `pops` list for a match with the address family chosen by the application. if no match occurs, the error `-EINVAL` is returned. Otherwise, it keeps a pointer to the pops entry that matched.
- Gets a free in-core inode. Allocates memory for a `socket` structure and initializes various fields in it. A pointer to the previously matched `pops` entry is saved in `ops` field of the `socket` structure. The in-core inode structure `inode`, has a field of type void pointer to allow the file system code to store a pointer to its private data. A pointer to the newly allocated and initialized `socket` is stored in this field.
- Allocates an entry in the system-wide file table and initializes it. A pointer to the newly allocated in-core inode is saved in this entry. Mode is set to read/write. The file table entry carries a pointer `f_op` to a structure `file_ops` that holds pointers to functions that actually implement the file system. The socket layer too has one such set of function pointers called `socket_file_ops` whose address is marked in the `f_op` field of the file table structure.
- Allocates a free file descriptor from the file table in the user area and sets up a pointer to the file table entry allocated in the previous step.
- Selects the function pointed to by the `create` field in the `pops` structure of the selected protocol family (`socket->ops->create`) and calls it, in effect giving control to the protocol family.
- Returns success if the family specific `create` returns successfully. otherwise, undoes every thing and returns failure.

In case of `PF_INET6`, the `sys_socket` gives control to the family specific create routine `inet6_create`. One of the main functions of this is to allocate and initialize the

Protocol Control Block (PCB). Ideally all the member protocols should define their own PCBs and memory should be allocated in the protocol specific create routine. But the `INET` family, uses a common PCB for all the three protocols UDP, TCP and IP, though most of it is TCP specific. We decided to use the same PCB structure `sock` for INET6 also for two reasons. First, utility routines to manipulate this structure are already available Second, TCP and UDP irrespective of the underlying protocol will need the same PCB and most of the IP specific portion can be re-used for IPv6. So now, that all the member protocols are going to have the same PCB, we should allocate it in the common family specific create routine. The `inet6_create` does the following things.

- Checks if any specific protocol is requested by the application (in the third parameter of the `socket` system call). If so, ensures consistency between the socket type and the protocol. If the application did not choose any protocol specifically, an appropriate protocol is decided upon based on the socket type. A pointer to the protocol operations structure struct `proto` of the chosen protocol is remembered.
- Allocates the PCB (`sock` structure) and initializes it. Store the pointer to the `proto` struct of the protocol in it. Link this PCB to the socket struct by saving a pointer to it in the `data` field of the socket.
- Lets the protocol do its initializations allocations and other protocol specific processing by calling its `create` method.

The opening of a raw IPv6 socket results in a call to `rawv6_init` at the lowest level. This simply links the PCB into a list of raw ipv6 sockets after properly increasing the priority level. The success indication from this function propagates upwards to the application.

5.2 Close

The processing of `close` is trivial. It basically involves freeing of all the structures allocated and releasing of the table entries used. The control first reaches down to the raw IPv6 handler for `close` - the `rawv6_release`, via the family level handler `inet6_release`.

The `rawv6_release` takes the following steps.

- Checks and releases any un-read buffers queued up in the protocol control block.
- Removes the PCB from the list of raw IPv6 PCBs after taking care of atomicity of the operation.

The `inet6_release` frees the PCB and gives control back to the socket level function which frees the `socket` and returns to the VFS level. The VFS level handler, frees the in-core inode and releases the entries in system-wide and the per-process file tables and returns to the application.

5.3 Sendto

This is one of the most complicated system calls. The other two system calls that pass data to the socket `write` and `send` can be implemented as special cases of `sendto`.

The VFS level routines simply follow the pointer `socket_file_ops` and call the sendto handler of the `INET6` family - `inet6_sendto`. This checks if the protocol has any handler for sendto, and if available hands over the control to it, otherwise returns `-ENOTSUPP`.

In the case of a raw IPv6 socket, the control thus reaches the `rawv6_sendto`. The processing in this function can be divided into the following five major steps.

1. Sanity checks
2. Deciding the next hop IPv6 address and the corresponding outgoing interface.
3. Allocating buffers for the packet.
4. Constructing IPv6 header and any other extension headers.
5. Forwarding the packet to the device layer.

We explain each of these steps in greater detail in the following paragraphs.

5.3.1 Sanity checks

The raw IPv6 in this implementation does not support any flags, so the routine first makes sure that all the bits in the supplied flags are zero.

Since the raw ipv6 socket is a datagram socket, packets to different destinations can be multiplexed. So for an unconnected socket, the application must supply the destination address. This can be checked by ensuring the size of the address is at least the size of the `sockaddr_in6` structure.

If the supplied address size is more than that of one `sockaddr_in6`, it is inferred that the application is passing a source route in which case the size of the address should be an integral multiple of the size of `sockaddr_in6`. Application can specify source route in sendto only if it has not already specified one, through the `IPV6_SRCROUTE` socket option. An error is returned if there is a saved source route in the PCB.

5.3.2 Next hop determination

The next hop determination algorithm uses several tables which are described below.

- **Prefix cache**: This table contains all the prefixes that are known to be local. i.e., if a destination's address matches with any of the prefixes, the destination is directly reachable and the next hop address is same as the destination address. This table has three fields : 16 byte long IPv6 prefix, length of the prefix, and a pointer to the device through which the destination can be reached. We have provided an ioctl command `SIOCSV6PFCENT` that makes an entry in this table. This can be used only by the super-user. Actually, this table should be filled in automatically by the host, as part of router advertisement processing. For this, full fledged neighbor discovery [TEA96] should be supported. This table is implemented as a doubly linked list sorted in the decreasing order of the prefix length.

- **Default router cache**: This table maintains a list of default routers that are willing to accept traffic destined for out-side of the network. In future, this will also be filled from the router advertisements. However we have provided another ioctl command `SIOCSV6DEFRTR` which allows the super-user to make entries in this table. This table is maintained as a doubly linked circular list. This is done to allow a round robin selection of routers.

- **Routing table**: This table is used only in routers. In hosts this will be a null list. This is similar to the old IPv4 except for the longer addresses. This table maps a prefix to a next hop router address. The major fields are: prefix, prefix length, and the next hop router address. We have retained other standard fields like the routing metric, flags etc., to make the deployment of the adapted versions of the standard routing protocols like, RIPv6, OSPFv6. We also extended the semantics of the ioctl command `SIOCADDRT`, so that if it is attempted on an IPv6 socket, an IPv6 route is added to this cache.

This is also built as a doubly linked list sorted in the descending order of the prefix length, so that a match with longer prefix is attempted first. This guarantees the longest match, which is the major requirement of the CIDR style of routing, if we commence our search always at the start of the list.

- **Destination cache**: The next hop computation is saved in this table. This has three important fields: destination address, its next hop address and the outgoing device. In later-versions this table will serve as a one-place repository for all the destination specific information.

In two cases the packet should be sent out only if the destination is directly connected: one is when the socket option `SO_DONTROUTE` is set and the second is during the routing header processing. This kind of routing is called local routing.

If a source-route is specified for this packet, the first address in it, specifies the destination of the packet as far as the routing at this node is concerned. So we should route the packet based on the first address found in the source-route, instead of the destination address specified by the application. The actual destination address is put in the routing header as the last address. The first hop address based on which the routing is done, is kept in the main IPv6 header as the destination address.

The function `ipv6_route` computes the next hop address. It takes two input parameters: the destination address and a flag which when set indicates that only local routing should be attempted. On successful completion, it outputs the next hop address via an output parameter and returns a pointer to the `device` structure of the outgoing device.

The algorithm for determining the next hop is shown in Figure 3

It first searches the destination cache, to see if any previously decided next hop address for the given destination is available. If so, the next hop and the device are taken from the matched entry. Otherwise, the algorithm checks to see if the destination is on-link, by searching in the prefix cache. If a match occurs, the next-hop is same as the destination and the device is taken from the matched entry.

If no prefix in the prefix cache matches, destination is decided to be off-link and if non-local routing is requested, a router is selected. If the routing table is not empty, it is tried first. If a route is found the next hop is the one specified in the routing table entry, otherwise a router from the default router list is picked up as the next hop. Again the outgoing device for this next hop address is found by matching it with one of the prefixes in the prefix cache and selecting the corresponding outgoing device.

In both prefix cache and routing table, match with the longest prefix is guaranteed since we alway commence our search from the heads of the lists and they are organized in decreasing order of the prefix length.

5.3.3 Buffer allocation

The next step is to allocate buffers for the packet. As mentioned earlier, in Linux, the entire packet with all the headers are stored in a single linear buffer called `sk_buff`. So we need to calculate the maximum size of the packet before allocating the buffers.

For raw IPv6 the space required is the sum of the lengths of MAC header IPv6 header, any extension headers and data. The MAC header length can be found from the `device` structure of the outgoing device selected by the routing. The field `hard_header_len` contains the length of the MAC header the driver is going to build. Size of the IPv6 header is always 40 bytes. During the sanity checking phase a flag is set if the packet is going to contain a routing header. The routing header size is easily calculated by the formula `no_of_addresses * 16 + 8`. Data length comes as a parameter from the application. The total buffer space required is calculated by adding all these sizes.

```
ipv6_route:
Input   :  dst_addr and local_flag.  Output  :  nexthop_addr.
Returns :  pointer to the device structure on success and NULL on failure.

/* Look for a hit in the destination cache.  */
dst_cache_entry = lookup_dst_cache(dst_addr);
if(found) /* Cache hit.  */
   nexthop_addr = next hop address from dst_cache_entry;
   save in destination cache and return (device from dst_cache_entry)

device = null;
/* Cache miss - try matching the onlink-prefixes.  */
device = lookup_prefix_cahce(dst_addr);
if(device) /* destination is directly reachable.  */
   copy dst_addr into nexthop_addr;
   save in destination cache and return (device);

if (local_flag)      /* only local-routing.  */
   return(null); /* Destination unreachable.  */

/* Destination is off-link.  try finding a route to it.  */
nexthop_addr = null;
route_entry = lookup_route_table(dst_addr);
if(route_entry)
   nexthop_addr = router address in route_entry;
else
   /* No route try pickup a default router.  */
   nexthop_addr = get_default_router();

if(nexthop_addr)
   /*
    * Found the next hop, This must be on-link
    * so get the address from prefix cache.
    */
   device = lookup_prefix_cahce(dst_addr);
   save in destination cache and return(device)
else
   /* Destination unreachable */
   return(null);
```

Figure 3: Algorithm for next hop determination.

But before allocating the `sk_buff` we need to make one more check. Our implementation does not support fragmentation and re-assembly. Therefore the total size of the packet should not exceed the MTU of the outgoing device. The MTU of the device is also available in the `device` structure.

It is checked, if the allocation of memory to the new packet will violate the maximum write-buffer space that can be allocated for this socket. If so, we will put the process to interruptible sleep, if the socket is a blocking one. The drivers after sending a packet, release the `sk_buff` and wake up any processes that are sleeping for the write-buffers. If the socket is non-blocking the error `EWOULDBLOCK` is returned.

After making sure that we can allocate more write-buffers without exceeding the limits, we'll try to allocate an `sk_buff`. This can fail if the system is running low on memory, in which case an error `ENOBUFFS` is returned. Otherwise the buffer allocation function completes successfully.

5.3.4 Building the headers

Once the buffers are allocated, the sendto handler proceeds to build the headers. As the current implementation supports only, Ethernet, we use the terms MAC header and Ethernet header interchangeably. The function `ipv6_build_header` is the top most function that the `rawv6_sendto` uses.

Mac header

Firstly, enough space is reserved at the head of the `sk_buff` using `skb_reserve`. To construct the MAC header we need a mechanism which maps the IPv6 addresses to MAC (Ethernet) addresses i.e., a counterpart of ARP in IPv4 domain. In a full-fledged implementation neighbor discovery will take care of this mapping, in the absence of which we resorted to the following solution.

If the address is an IPv4 compatible one, the solution is trivial. Extract the last four bytes and look up in the ARP cache using the `arp_find` function. We can even postpone the construction of the Ethernet header, until the moment it reaches the device driver. The control data of `sk_buff` has two fields to hold IPv4 addresses of the source and the next hop. A Flag is also provided which when set indicates to the driver that the MAC header is not built by the upper-layer protocol, but needs to be built now. in which case, the driver uses the source and next hop addresses from the `sk-buff` and builds the MAC header. So, in the IPv6 module, we need not build the MAC header at all, the module extracts the IPv4 addresses of the source and destination from the supplied IPv4 compatible IPv6 addresses, and records them in the above mentioned two fields of the `sk_buff` control data. Then it sets the flag also, instructing the device driver, to build the MAC header.

If the destination address is a general IPv6 address, the last six bytes are assumed to carry its Ethernet address. The last six bytes are used as the Ethernet address in constructing the Ethernet header. The IPv6 addressing architecture has provision for this kind of the addresses, since they make the auto address configuration easy.

The `ipv6_build_header` uses a function pointer from the `device` structure to construct the MAC header. This function takes as parameters, the MAC addresses of both source and the destination (next hop), and builds the MAC header in the previously reserved space at the head of the packet.

▮ *IPv6 header*

Building the IPv6 header is straightforward. The data area in the `sk_buff` is extended by 40 bytes and the header is built there. Only thing to be noted is that in the presence of the routing header, the IPv6 header should carry the first hop IPv6 address found in the routing header, as the destination address. Actual destination address goes into the routing header as the last hop. The priority and the flow-id are copied as they are, from the destination sockaddr_in6 supplied by the application. No checks or attempts to use these fields are made. The next header field is assigned a value, that indicates the presence/absence of the routing header. The hop-limit is assigned from the value found in the PCB. The hop limit cab be set by the application through a socket option.

Routing header

The source route is stored in the PCB, in three fields: v6srcroute_len carries the size of the source route in 8 octet units, v6srcroute carries the sequence of addresses, and v6srcroute_lsbmp holds the loose/strict bitmap. The amount of space to be allocated in the packet can be computed by the formula : v6srcroute_len * 8 + 8. The extra 8 octets are used for the control data of the routing header. After allocating the space in the sk_buff, it is straight forward to fill in the routing header. The first address in the sequence is not written into the routing header, since it is going to be the destination address in the main IPv6 header and after the last address, the actual destination address is copied as the last address into the routing header.

5.3.5 Forwarding to the device layer

As mentioned earlier, all the devices are accessed through a single interfacing function dev_queue_xmit. Just before forwarding rawv6_sendto checks the status of the device. It looks at the flags in the device and makes sure that the device is up and running. Otherwise it returns an error ENETUNREACH. After this check, the dev_queue_xmit function is called with pointers to the device and the packet. The packet is queued and is sent, when its turn comes. Actually the bottom half takes over here. Every time the network bottom half handler is run, it looks at the packet queues, and if there are any packets waiting to be transmitted, it sends them by invoking the hardware handling routines.

5.4 Recvfrom

There are two parts for this system call, that work asynchronously. The bottom half is responsible for receiving the packets from the net and delivering them to appropriate sockets. When the application requests for data via `recvfrom`, the upper half dequeues packets from the receive queue in the PCB and copies the data to application buffers.

The other two system calls that read data from the socket - `read` and `recv` can be trivially implemented as special cases of `recvfrom`. Therefore we will explain below only the operation of `recvfrom`. We will first explain the the bottom half then proceed to explain the upper half.

5.4.1 Bottom half

When a packet is received from the net, the device driver allocates an `sk_buff` and copies the packet into it. Then it examines the MAC header and extracts the protocol ID from it and stores it in the `protocol` field of the control data part of the `sk_buff`. It then removes the MAC header and queues up the packet for despatch to higher level protocols. When the network bottom half handler runs on the next clock interrupt, it takes up packets one by one from this queue and searches the table that maps the protocol ID to corresponding handler function. After the appropriate handler is identified, it is called with the packet as the argument. In case of IPv6 the packet is passed onto the protocol handler function `ipv6_rcv`.

The function `ipv6_rcv` proceeds along the following steps:

- Extract the IPv6 header from the packet and make sure that version number is six.

- Next step is to see if the packet is actually addressed to this host. Each host keeps a list of addresses it is expected to recognize. The destination address found is matched against each of them. If no match occurs, the packet needs to be forwarded, and so it is passed on to the function `ipv6_forward`.
- If the destination address in the packet matches one of the recognized addresses, the packet is apparently destined to this host only, but there can be a routing header and it can request the packet to be forwarded. So check the next header field in the IPv6 header. If it indicates, that the packet is carrying a routing header, process it by passing it to the `ipv6_process_route_hdr`. After this, the destination address could have changed, so it is again matched with the set of recognized addresses and is forwarded if necessary.
- The IPv6 as well as other extension headers are removed from the packet and then it is passed on to the raw IPv6 receive function `ipv6_rcv`. Once TCP and UDP are added, this function will be extended to call the receive function of the appropriate protocol, depending upon the next header field in the last IPv6 extension header.

The `rawv6_rcv` picks up the PCBs chained in `ipv6_sk_list` and delivers them a copy of the packet. For this, it queues up the packet in the `receive_queue` of the PCB and wakes up all the processes that have slept waiting for data to appear on that particular socket.

▮ *Routing header processing*

We provide support for the type 0 routing header which is described in [Dee95]. The important fields are: `length, segments_left, sl_bmp` and `type` . `length` field gives the length of the `data` i.e., the list of addresses, in units of 8 octets. `segments_left` indicates how many more addresses in the `data` are yet to be visited. `sl_bmp` is the bit map in which bits are numbered from left to right and, k

th bit denotes whether the *k* th hop is to be routed strict or loose i.e., whether the *k* th address need to be a neighbor of *k-1* th address, or not.

The processing of type 0 header follows the steps out-lined below:

- If the routing header type is not a known type i.e., not of type zero and if there are no more segments left, simply ignore the routing header. But,if a routing header of unknown type with a non-zero value in `segments_left` field, is encountered the packet is dropped and an ICMPv6 message needs to be generated.
- Ensure the sanity of the `length` and `segments_left` fields. The length should be an even number and should not exceed 46, since the maximum number of hops that a routing header of type zero is limited by the length of the `sl_bmp` to 23. The `segments_left` should not exceed the total no of hops i.e., `length`/2.
- Compute the offset of the next hop address. Decrement `segments_left` by one and subtract it from the total no of addresses (`length/2`). Multiply this quantity by 16 to get the byte offset of the next address form the start of the `data` field.
- Interchange this address of the next hop and the destination address found in IPv6 header. Return the strict/loose bit corresponding to this hop from the `sl_bmp`.

A return value of zero from this routine indicates that the next hop does not need to be a neighbour. On the other hand any positive value indicates that the next hops must be a neighbor, and hence only local routing should be attempted. So the value returned by this routine is used as the `local_flag` for the `ipv6_route` when forwarding is being attempted.

■ *Forwarding*

The main steps in forwarding algorithm (`ipv6_forward`) are described below.

- Decrement the hop limit in IPv6 header by one and drop the packet if this becomes zero.
- Route the packet and find the outgoing device. The function `ipv6_forward` gets the `local_flag` as parameter from `ipv6_rcv` and passes it on to the `ipv6_route`.
- Check if the new device chosen is the same on which this packet is received. This check may be helpful in future when we augment the ability to generate and handle route redirect ICMPv6 messages.
- Make sure that the MTU of the new device will be able to handle the packet. If it cannot, drop the packet.
- See if the space available at the head of the packet is sufficient to construct the new MAC header, if so go ahead and construct it. If not, allocate new skb and copy the packet and then construct the new MAC header. We always have enough head room since we support only Ethernet.
- Check whether the device is up and running and send the packet off by calling the `dev_queue_xmit`.

5.4.2 Upper half

When the application requests data via the `recvfrom` system call, the control eventually comes to the function `rawv6_recvfrom`.

Similar to `sendto`, in this version, we do not support any flags in `recvfrom` as well. So first it is made sure that they are zero. The function tries to dequeue a packet from the `receive_queue` in the PCB. If the socket is non blocking an error `EWOULDBLOCK` is returned. Other wise the process is put to sleep, until some data appears.

After dequeuing a packet, data bytes are copied to the application buffers upto a maximum of the number of bytes requested by the application. If the packet holds more data, the remaining data is discarded, as the `sk_buff` is freed after copying. It returns the actual number of bytes copied.

5.5 Setsockopt and getsockopt

These system calls copy some data from application to kernel tables or vice versa. The main steps are given below.

- Check whether the socket option is valid or not. This may include checking if the calling process is running with super user privileges or not. Check if the level of the option is correct or not.
- Verify if a read (write) operation on the area specified by the argument of setsockopt(getsockopt) is going to violate the memory protection. If so, return the `EINVAL` error.
- Copy the data from the argument to the kernel data structure in case of setsockopt and from the kernel data structures to the argument in case of getsockopt.

Implementation of ioctl calls is also on the same lines, except for a few obvious differences like the absence of checks for option level.

Chapter 6

Tunneling and 6Bone

In this chapter we discuss the implementation of the pseudo device driver to tunnel IPv6 packet over IPv4 clouds. We also explain how this is used in laying out a IPv6 backbone over the IIT Kanpur campus network.

6.1 Tunneling

The existing IPv4 infrastructure is too vast to be phased out on a flag day. Therefore IPv4 and IPv6 are going to co-exist for reasonably long time over which the IPv6 is gradually phased in in a diffused manner. This makes, tunneling of the IPv6 datagrams over IPv4 routing infrastructure, not just attractive but also imperative. The [RE96] specifies several transition mechanisms. There are two main variants of tunneling: Automatic tunneling and configured tunneling.

In automatic tunneling, the host will decide to tunnel the packet if the destination address is an IPv4 compatible IPv6 address. The IPv4 header that encapsulates the IPv6 packet, carries the IPv4 address that is embedded in the destination IPv6 address, as its destination address. So there is no fixed configured endpoint for tunneling. The endpoint is determined dynamically from the destination IPv6 address, and there is no need of any configuration by the system administrator. But the inherent disadvantage is that this mechanism may not exploit whatever IPv6 infrastructure that is existing.

In contrast to automatic tunneling, configured tunneling works in terms of a point to point pseudo device driver with a fixed endpoint. This requires some setup by the system administrator, but not only allows exploitation of IPv6 infra structure, but also gives the system administrator a finer control in deploying IPv6.

6.1.1 Tunnel driver

We support configured tunneling in our implementation. We developed a pseudo device driver that looks like point to point IPv6 link. This driver can be configured using the same `ifconfig` and the kernel modules access it like any other network device i.e., through `dev_queue_xmit`. Instances of these driver are named `v6tunl1, v6tunl2`, etc.

All the network device drivers should implement two functions: one initializer and a transmit function. The initializer is called only once, during the boot time. A statically initialized linked list of `device` structures, is employed to keep track of all the devices, in the same as the protocol families are organized. The list is traversed by the `dev_init` when it is called by `sock_init` and the initializer function of each of the device is called. This function initializes and, fills in the rest of the details in the `device` structure. `v6tunnel_init` is the initializer for `v6tunl` devices.

The transmit function is called by the `dev_queue_xmit`, this should transmit the packet on the network. The processing done in the transmit function `v6tunnel_xmit` is explained below.

- Do some sanity checking like verifying the version number in the packet, making sure that the device has been assigned IPv4 addresses for both ends, it is supposed to connect as a point to point link etc.
- Switch on the device busy flag in the `device` struct. This puts the `dev_xmit` on hold.
- Allocate a new `sk_buff` and, copy the packet, leaving enough head room for the new IPv4 header.
- Fill in the IPv4 header
  - The source and destination addresses come from the `device` struct which have been assigned by the system administrator using `ifconfig`.
  - After building the packet, it is going to be handed over to the `ip_forward` to be forwarded and it is going to decrement the TTL by one and drop the packet if it becomes to zero. So, we assign a value one more than what is found in the `hop_limit` field of the IPv6 header.
  - The [JJ94] defines a protocol ID for IPv6 tunneling over IPv4 and it is assigned to the `protocol` field in the IP header.
- Compute the checksum for the new IPv4 header.
- Pass the packet to `ip_forward`.

On the receive side, this packet is going to be received as an IPv4 packet so it is passed to the IPv4 receive function `ip_rcv`. We define a new dummy protocol, IPv6IP, in `INET` family with the protocol ID that is used for this tunneling and a simple receive function. The `ip_rcv` after the normal processing, looks at the protocol ID in the IPv4 header and hands the packet is handed over to the dummy protocol, IPv6IP. Since the actual IPv4 packet is received on the Ethernet device, the device pointer points to the `eth0` device. Therefore, the receive function of the IPv6IP `ipv6ip_rcv` does a little processing on the packet, to make it look as if it had come from the `v6tunl` driver and hands it over to the IPv6 receive function `ipv6_rcv`. From here onwards the normal processing of an IPv6 packet follows.

6.2 6Bone

Since commercial implementations of IPv6 are expected to start appearing in the market soon, work has been started to set up a world-wide IPv6 backbone called "6Bone". This backbone is going to be constructed on the lines of the multicast backbone "Mbone". It's going to connect islands of IPv6 networks together, by means of dedicated IPv6 links where possible or by the configured IPv6 tunnels described above. Since we developed all the required technology, we set out to set up a miniature version of such 6Bone in IIT Kanpur campus network. We could set up the 6Bone with two Linux machines acting as backbone routers. We also had two hosts, one each on CSE and ACES networks. We describe the set up in the following sections and explain the journey of an IPv6 packet through this network with source route.

6.2.1 Local setup

The IIT Kanpur campus network consists of a backbone Ethernet, to which all individual department Ethernets are connected via IPv4 routers. We chose two machines, `godavari` on CSE net and `kusha` on ACES net, to act as IPv6 routers. We joined these two machines with a bidirectional tunnel comprising of two unidirectional IPv6/IPv4 tunnels (`v6tunl`). We also have one simple host on each of the two networks. Addresses assigned to these machines are shown in the following table.

Constituent machines of 6Bone

Name	IPv4 Address	Site-local IPv6 Address
godavari	144.16.162.121	FEC9:6164:6F72:6555:1:0000:C00F:00D9
kusha	144.16.160.231	FEC9:6164:6F72:6555:2:0080:4800:39BE
cspc12	144.16.162.112	FEC9:6164:6F72:6555:1:0000:E8C3:A938
eepc51	144.16.160.151	FEC9:6164:6F72:6555:2:0080:4884:D25F

Since all of these machine can communicate using IPv4 as well, their IPv6 modules recognize the corresponding IPv4 compatible IPv6 addresses also. We use these addresses in the forthcoming examples as they are more compact and readable. The prefix caches, routing tables and default router caches on different machines are shown in the following tables.

Default router cache		
Name	IPv4 Compatible IPv6 Address	Default Router
cspc12	::144.16.162.112	::144.16.162.121
eepc51	::144.16.160.151	::144.16.160.231

Prefix cache				
	IPv4 Compatible	Prefix		
Name	IPv6 Address	Address	Length	Device
godavari	::144.16.162.121	::144.16.160.231	16	v6tunl1
		::144.16.162.0	15	eth0
		::127.0.0.0	13	lo
kusha	::144.16.160.231	::144.16.162.121	16	v6tunl1
		::144.16.160.0	15	eth0
		::127.0.0.0	13	lo
cspc12	::144.16.162.112	::144.16.162.0	15	eth0
		::127.0.0.0	13	lo
eepc51	::144.16.160.151	::144.16.160.0	15	eth0
		::127.0.0.0	13	lo

Routing table				
	IPv4 Compatible	Prefix		
Name	IPv6 Address	Address	Length	Next-hop
godavari	::144.16.162.121	::144.16.160.0	15	::144.16.160.231
kusha	::144.16.160.231	::144.16.162.0	15	::144.16.162.121

Now consider an IPv6 packet being sent from `godavari` to `eepc51` with a source route that specifies an intermediate host `cspc12` to be visited. We see how each machine processes this packet.

1. The IPv6 module on `godavari` after receiving the packet from the application, finds that a source route is specified and it picks up the first hop destination, `cspc12` and routes the packet. The second entry (::144.16.162.112/15) in the prefix cache matches, and the corresponding device `eth0` is selected as outgoing device. The packet is sent to `cspc12` with the destination IPv6 address as `cspc12` itself and with a routing header specifying `eepc52` as the next-hop destination.

2. The host `cspc12` receives the packet and successfully matches the destination address with one of its own. It then proceeds to process the header. It finds that there is a routing header which specifies that the packet is to be forwarded to the next destination, `eepc51`. It fails to match any of the on-link prefixes found in its prefix cache. So it decides to send it to a router. Since it is a simple host its routing table is empty. It looks up in the default router cache and selects `godavari` as the next hop.

3. The packet reaches `godavari` with a destination address of `eepc52`. The IPv6 module on `godavari` decides that this packet should be forwarded, since the packet's destination address does not match with any of its own addresses. No prefix from the prefix cache matches the packet's destination address and the routing table is consulted. From the routing table it is found that the router for this host is `kusha`. Hence, the prefix cache is searched with the address of `kusha` and the tunnel `v6tunl1` is chosen as the outgoing device. The packet is sent on this device with `eepc51` as the destination.

4. The machine `kusha` decides that the packet should be forwarded, since none of its addresses match with the destination address found in the packet. It searches in the prefix cache and finds out that the host `eepc51` is directly reachable through the device `eth0`.

5. Finally, `eepc51` receives the packet and matches its address with the destination of the packet. It then checks that there are no more hops specified in the routing header and the packet has reached the final destination.

As mentioned earlier, various tables used by IPv6 will be set up by the neighbour discovery module. Since the current version does not support neighbour discovery, we have provided some utility programs that allow the system administrator to manipulate these tables. In the next chapter we explain these utilities and other tools provided.

Chapter 7

Tools and utilities

In this chapter we describe the facilities we have provided to setup, maintain and test the IPv6 module in the kernel. We explain the `/proc` file system interface to the kernel tables and also give brief descriptions of a few utility programs we have developed.

7.1 The `/proc` file system interface

The `/proc` file system started as a pseudo file system under VFS, with the main aim to provide a file system interface to the virtual memory of the active processes. In this file system, which is traditionally mounted on `/proc`, a process is represented as a file under `/proc` directory with its process ID forming the name. For example, reading the file `/proc/43`, from the beginning, is equivalent to reading the virtual memory of the process with ID 43 starting at the virtual address zero. The process can be manipulated in a variety of ways by executing appropriate ioctl commands on the file descriptor obtained by opening the corresponding file in the `/proc` file system. Thus this interface forms a superset of the process manipulating system call `ptrace`.

Linux extended this concept to cover the kernel tables and some system status related information as well. For example, the file /proc/uptime reports the time for which the system has been up, since the last boot. The file /proc/net/arp can be read to get the kernel's ARP tables. All the major networking protocols have one such file in /proc/net, which can be read to see some important tables and status variables.

We created two such files in the same directory with names ipv6 and ipv6_route. The file ipv6 gives out information like the number of open raw IPv6 sockets, number of datagrams received and transmitted since boot up time, etc. Similarly, the other file ipv6_route, outputs the kernel tables like prefix cache, router cache, routing table etc.

The steps given below are followed to create these two files.

- The whole directory hierarchy for this file system is maintained in memory. So we need to allocate inode numbers for these two files. The include file <linux/proc_fs.h> is edited to include two more inodes in the enumerated list. These numbers are later used to create the in-core inodes when an open is attempted on the corresponding file.
- The module which wants to create a proc file should tell the proc file system module, by giving the inode number. This is done in the protocol initializer for IPv6 ipv6_init. Along with the inode number, a function pointer also needs to be supplied to which all the read requests on the corresponding file converge.
- Two such information supplying functions get_ipv6_info and get_ipv6_-route_info are developed. They simply print the tables and variables into the buffer supplied.

7.2 Packet snooper

A small but powerful snooper utility is also developed to read and (write) packets from (to) the ethernet device directly. This has been of great use to catch and examine the packets generated by the IPv6 module and to feed it with test packets.

This is based on a special type socket `SOCK_PACKET` that can be opened by the super user. This allows direct access to the devices. The device is identified by giving its name as an ascii string in the data part of the `sockaddr`. This is implemented as a member protocol in `INET`. The third parameter in `socket` system call, should identify the hexadecimal constant the packets, the socket is expected to catch, carry in the protocol ID field of their MAC header. A special constant `DEV_P_ALL` makes the socket catch all the packets. Packets are copied into these sockets at two points: the incoming packets in the bottom half handler, and the outgoing packets in the `dev_queue_xmit`. While writing on this socket, the application should construct the MAC header also.

7.3 Miscellaneous utilities

The following miscellaneous utilities have also been developed.

- We had to change the `ifreq` struct to add the IPv6 address also to the union that carries the argument for almost all the ioctl commands, that work on the device drivers. The `ifconfig` depends on these heavily and the changed size creates problems in the existing executable of this program. We provided a new executable that works on the new kernel.
- Small utility programs: `addpf, addrtr, addroute`, are developed to add entries to prefix default router and routing tables respectively. These programs use the special ioctl commands implemented for this purpose.

Chapter 8

Conclusions

We have successfully implemented the basic IPv6 in Linux, as a new protocol family. An enhanced version of the popular BSD API is also provided. Applications could access the protocol module through the raw socket interface of this API and could communicate with each other.

We have also implemented a minimal IPv6 router. This router uses CIDR style of routing and could forward packets between IPv6 networks. We provided the functionality of source routing also, which is going to aid in experimenting and conducting research into issues like mobility etc. We could use the existing routing infrastructure by implementing a tunnel driver that tunnel IPv6 packets in IPv4 routing domains. This helps in connecting island of IPv6 networks through the IPv4 networks.

An IPv6 backbone is created n IIT Kanpur campus network with two IPv6 routers. We also developed a set of basic utilities that aid in setting up and managing the IPv6 networks. This backbone allows any host in either CSE network or ACES network to be upgraded to IPv6 without worrying about any other dependencies.

8.1 Future Work

The implementation may be extended and refined in the following ways.

- The ICMPv6 needs to be implemented. This will make Path MTU discovery feasible. Support for fragmentation and re-assembly can follow.
- Multicast capability may be added.
- A few changes in transport layer protocols like TCP and UDP are required. For example the pseudo-header for CRC computation is different in case of IPv6.
- Neighbor discovery should be implemented and the dependency on ARP module should be removed.
- Auto address configuration may be added.
- Support for security and authentication may be a provided.
- Different routing protocols like, RIP, OSPF etc., may be adapted to run on IPv6.
- More research needed to study the flow handling techniques at the intermediate routers and how the resource reservation protocols like RSVP fit into this framework.
- API can be improved to include new multicast related socket options, ways for the applications to get unique flow-ids, etc.

References

[Bac86] Maurice J. Bach. *The Design of The UNIX Operating System.* Prentice-Hall Inc., Englewood Cliffs, N.J., U.S.A., 1986.

[C94] Huitema C. *The H ratio for address assignment efficiency.* RFC: 1715, Network Information Center, INRIA, October 1994.

[Car94] Carpenter B. *IPng White Paper on Transition and Other Considerations.* RFC: 1671, Network Information Center, CERN, Geneva., August 1994.

[CE90] Mogul J C and Deering S E. *Path MTU discovery* . RFC: 1191, Network Information Center, November 1990.

[Dee95] Deering S and Hinden R. *Internet Protocol, Version 6 (IPv6) Specification.* RFC: 1883, Network Information Center, Xerox PARC, Ipsilon Netwoks, December 1995.

[ESJ96] Gilligan R E, Thompson S, and Bound J. *Basic Socket Interface Extensions for IPv6.* Internet draft, Internet Engineering Task Force, Sun, Bellcore, Digital, April 1996. Work in Progress. Available as `draft-ietf-ipngwg-bsd-api-05.txt` at `http://www.ietf.cnri.reston.va.us/`.

[Gro94] Groos P. *A Direction for IPng.* RFC: 1719, Network Information Center, MCI, December 1994.

[Hin95] Hindon, R. and Deering, S. . *IP Version 6 Addressing Architecture.* RFC: 1884, Network Information Center, Ipsilon Networks, Xerox PARC., December 1995.

[IAN95] IANA. *Class A Subnet Experiment.* RFC: 1797, Network Information Center, ISI, University of Southern California., April 1995.

[J81] Postel J. *Internet Protocol.* RFC: 791, Network Information Center, ISI, University of Southern California, September 1981.

[JJ94] Reynolds J and Postel J. *Assigned Numbers.* RFC: 1700, Network Information Center, ISI, University of Southern California, October 1994.

[LMKQ89] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, MA, May 1989.

[Par95] Partridge C. *Using the Flow Label Field in IPv6.* RFC: 1809, Network Information Center, BBN Systems and Technologies, June 1995.

[R87] Callon R. *A Proposal for a Next Generation Internet Protocol.* Proposal to X3S3, December 1987.

[RE96] Gilligan R and Nordmark E. *Transition Mechanisms for IPv6 Hosts and Routers.* RFC: 1191, Network Information Center, Sun Microsystems, Inc., April 1996.

[Rek93] Rekhter Y and Li T. *An Architecture for IP Address Allocation with CIDR.* RFC: 1518, Network Information Center, T. J. Watson Research Center - IBM, Cisco Systems, September 1993.

[Rek95] Rekhter Y and Li T. *An Architecture for IPv6 Unicast Address Allocation.* RFC: 1887, Network Information Center, Cisco systems., December 1995.

[SA95] Bradner S and Mankin A. *The Recommendations for IP Next Generation Protocol.* RFC: 1752, Network Information Center, January 1995.

[Siy92] Siyan K. *An IP Address Extension Proposal.* RFC: 1365, Network Information Center, Siyan Consulting Services., September 1992.

[Tay94] Taylor M. *A Cellular Industry View of IPng.* RFC: 1674, Network Information Center, CDPD Consortium., August 1994.

[TEA96] Narten T, Nordmark E, and Simpson W A. *Neighbor Discovery for IP Version 6 (IPv6).* Internet draft, Internet Engineering Task Force, IBM, Sun, Daydreamer, March 1996. Work in Progress. Available as `draft-ietf-ipngwg-discovery-06.txt` at `http:/www.ietf.cni.reston.va.us`.

[Vec94] Vecchi M. *IPng Requirements: A Cable Television Industry Viewpoint.* RFC: 1686, Network Information Center, Time Warner Cable., August 1994.

[VTJK93] Fuller V, Li T, Yu J, and Varadhan K. *Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggreation Strategy.* RFC: 1519, Network Information Center, BARRNet, cisco, OARnet, September 1993.

[Wan92] Wang Z and Crowcroft J. *A Two-Tier Addess Structure for the Internet: A Solution for the Address Space Exhaustion.* RFC: 1335, Network Information Center, University College, London., June 1992.

[Wiz96] Network Wizards. *Internet Domain Survey.* Technical report, January 1996. Data is available at http://www.nw.com/.