IPv6 Implementation Study

A study of IPv6 Implementation in Linux

Emanuela Lins
Saurabh Agarwal

14 May 2003

Contents

1 Introduction **1**
- 1.1 IPv6 . . . 1
- 1.2 Networking Implementations . . . 1
- 1.3 Networking API . . . 1
- 1.4 Linux . . . 1
- 1.5 Conventions . . . 2

2 An Overview **2**
- 2.1 Socket Communication . . . 3

3 Network Buffers **4**
- 3.1 sk_buff Structure . . . 4
- 3.2 skb API . . . 6

4 Socket Families, Protocols and Packets **7**
- 4.1 Socket Families . . . 7
- 4.2 Protocols . . . 7
- 4.3 Packets . . . 8

5 INET6 **9**
- 5.1 INET6 Layer . . . 9
- 5.2 inet6_protocol Structure . . . 10
- 5.3 proto Structure . . . 10

6 IPv6 **11**
- 6.1 IPv6 Addresses . . . 11
  - 6.1.1 Host Address Configuration . . . 11
- 6.2 IPv6 Header . . . 12
  - 6.2.1 Extension Headers . . . 12
- 6.3 IPv6 Output Processing . . . 12
- 6.4 Input Processing . . . 13

7 Other Protocols **13**
- 7.1 ICMPv6 . . . 13
- 7.2 Neighbor Discovery . . . 13
- 7.3 IPSec . . . 14
- 7.4 Tunneling and Translation Mechanisms . . . 14

8 Conclusion **14**

A Linux Source Code Organization **17**

B Structures 18
B.1 INET6 proto_ops Structures 18
B.2 INET6 proto Structures 19

C Control Flow 21
C.1 Output Processing Control Flow 21
C.2 Input Processing Control Flow 22

1 Introduction

This project is a study of IPv6 implementation in Linux. The main goal is to look at a concrete implementation of IPv6, understand the software architecture and its internal working.

1.1 IPv6

IPv6 is the next generation network protocol in the TCP/IP protocol stack. The IPv6 protocol standards are being developed by the IPv6 working group in IETF. The current state of standards is available at their website, http://www.ietf.org/html.charters/ipv6-charter.html. The main features of IPv6 are the increased address space, auto-configuration of hosts and integration of IPSec. IPv6 implementation is still under development in various operating systems. The website http://playground.sun.com/ipv6/ipng-implementations.html contains information regarding implementations of IPv6.

1.2 Networking Implementations

There are two major styles of network protocol management in unixen, *BSD Sockets* and *streams*[1]. BSD sockets were introduced with 4.2 BSD in 1983, and streams with SVR3 in 1986. While streams provide a very nice API to add more protocols and device drivers[2], they tend to lose on performance when compared to sockets.

1.3 Networking API

In order to use networking protocols among other things, 4.2 BSD introduced the socket API. SVR3 also introduced another API called TLI[3]. However they are independent of networking implementation inside the kernel. A TLI API actually can be provided over sockets implementation and a socket API can be implemented on streams[4].

1.4 Linux

Linux implements the networking subsystem based on BSD sockets. The networking architecture of Linux is fairly object oriented. It is designed to support many different communication protocols. A typical Linux kernel can be running TCP/IP, IPX, DECNET, UNIX, and many more communication protocols simultaneously. The IPv6 implementation in Linux is still under development. The current stable kernel, Linux 2.4, is missing many IPv6 extensions and does not confirm to all drafts and RFCs. This project actually is a study of the Linux 2.5 kernel. While writing this report, the current release of the beta kernel is Linux 2.5.69.

[1] The term streams has nothing to do with standard I/O streams (`fopen` etc).

[2] In fact, the entire terminal I/O in streams derived kernels is written on streams.

[3] Now transformed into XTI.

[4] Thus there is no incompatibility in APIs. A client implemented with TLI can communicate with a server implemented with socket API and vice versa.

1.5 Conventions

Throughout this report we assume that the reader is familiar with fundamentals of operating systems and network programming. The following conventions are used in this report.

a. The snippets from the source code are written in `courier` monospace font.

b. ⟨directory/filename⟩ is used to represent a file in the Linux source code tree. A file name such as ⟨net/ipv6/icmp.c⟩ represents the file icmp.c in the net/ipv6 directory.

c. A figure which represents a structure `s` in a file ⟨f⟩ is named `s` ⟨f⟩.

2 An Overview

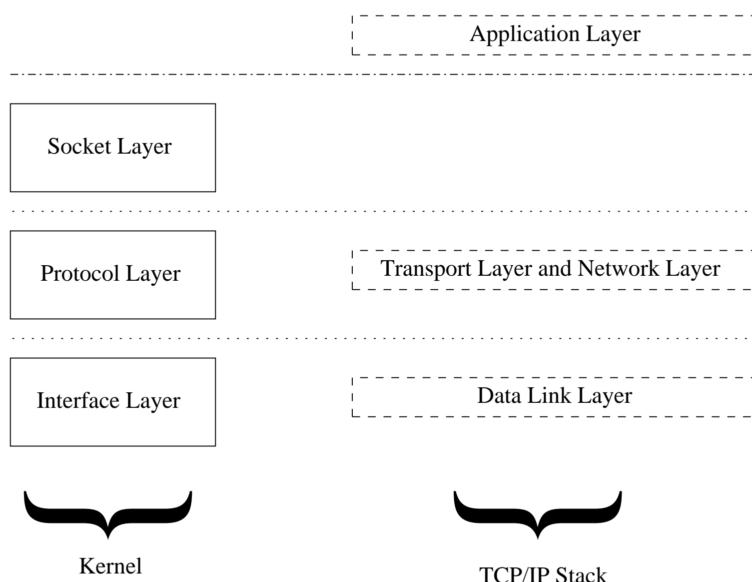

Figure 2.1: The networking subsystem inside the kernel and the TCP/IP stack

The networking subsystem inside the kernel can be viewed in three layers. Figure 2.1 shows these layers and where the TCP/IP stack (on right) lies with respect to them. The socket layer is a protocol-independent interface to the protocol-dependent layer below[1]. All system calls start at the protocol-independent socket layer. The protocol layer contains implementation of individual protocol families. Each protocol family may have its own internal structure within the protocol layer. The interface layer contains device drivers that communicate with network devices. Most of the protocol families do not require that the communicating hosts should be different machines. Thus sockets can also be used for IPC on the same machine using any protocol[2].

[1]The definition of various layers is taken from [9].

[2]Linux also provides the ability to create virtual interfaces. See ⟨Documentation/networking/tuntap.txt⟩ for more information.

2.1 Socket Communication

Communication using sockets follows the client server model. The server process creates a socket and *listens* on the socket. For a protocol such as TCP/IPv6, the client process creates a socket and *connects* to the server socket. Once the connection is established, the processes can send messages to each other using `send` or `write` system call. After the data has been written on the socket, the following actions take place within the kernel.

a. The socket layer copies the data from the user-space to a buffer in the kernel-space and calls the protocol layer output routine.

b. This routine calls the TCP output routine.

c. The TCP output routine processes the TCP header and adds it to the front of the data in the buffer and calls the IPv6 output routine.

d. The IPv6 output routine processes the IPv6 header and adds it to the front of the TCP header in the buffer and calls the interface layer output routine.

e. The interface output routine adds the datalink layer header to the front of the IP header and sends the packet to a device driver for transmission on the physical media.

The input processing is different from the output processing because the input is *asynchronous*. The reception of a packet is triggered by an interrupt to a device driver and not by a system call issued by the process. The actions on a packet reception are as follows.

a. The kernel processes the interrupt and schedules the device driver to run.

b. The device driver reads the data bytes from the device and stores the data in a buffer.

c. The device driver then passes the buffer to a general interface level input routine.

d. This routine determines which protocol layer should receive the packet.

e. In case the packet is an IPv6 packet, the buffer is added to the input queue of IPv6 and a software interrupt is raised.

f. This interrupt causes the IPv6 input process routine to run.

g. This routine looks at the type of IPv6 packet and the corresponding packet processing routine is called. If the packet is a TCP packet, then the TCP input routine is called.

h. The TCP input routine verifies the fields in the TCP header, and determines whether or not the process should receive the packet.

i. If yes, the data along with the IP address and port number of the sender are put into the input queue of the socket of the process.

The above assumes that the host does not *forward* packets. If it does, the IPv6 input routine will also decide whether the packet can be forwarded or not. If yes, the IPv6 output processing routine will be called. Otherwise all packets which are not meant for the host may be dropped.

3 Network Buffers

Network buffers are control structures with a block of shared memory which store incoming or outgoing packets. One of the main use of buffers is to avoid unnecessary copy of data. In the kernel a complete copy of the packet is done only a minimum required number of times, once from the user-space to the kernel-space and then from the kernel-space to the interface. In between, protocols look at the whole packet only if it is needed[1].

3.1 sk_buff Structure

The network buffers in Linux are called `sk_buff`, and commonly called skb. The first few members of `sk_buff` are shown in figure 3.1. Buffers are maintained as doubly linked lists. The layers act on these lists and process packets one by one.

Each buffer has a pointer `list` to the head of the list, and a pointer `sk` to the socket who owns the buffer. `stamp` is used for storing timestamps such as the packet arrival. The Network device on which the packet was received or will be sent is pointed by `dev`. Typically each of these members are initialized when the buffer is created[2].

```
struct sk_buff {
        /* These two members must be first. */
        struct sk_buff          *next;
        struct sk_buff          *prev;
        struct sk_buff_head     *list;

        struct sock             *sk;
        struct timeval          stamp;
        struct net_device       *dev;
```

Figure 3.1: `sk_buff` ⟨include/linux/sk_buff.h⟩

Buffers also have pointers to the transport header `h`, network header `nh` and link layer header `mac` of the packet (figure 3.2). This reflects the typical usage of skb. Protocols generally have nothing to do with data[3]. All they manipulate are headers. And this is also what happens inside the kernel. For outgoing packets, protocols in the protocol layer just add the appropriate headers and for incoming packets they remove[4] the headers. This is where the buffers play a central role. A packet never moves inside the kernel. It is the control that shifts from one layer to another layer. The main focus of networking implementation is to make this control efficient and transparent across various protocols.

There are many other members in the `skb_buff` structure which are used for various general functions and book-keeping. Some of them are shown in figure 3.3.

The `pkt_type` and `protocol` members are used to keep track of which protocols will operate on the buffer. Other members are quite difficult to explain without going into details of the `sk_buff`

[1] To compute checksum, to encrypt data etc.
[2] `dev` is initialized only for incoming packets. For outgoing packets this decision is taken later.
[3] Except for checksum or for encryption/decryption of IPSec packets.
[4] Not always. They just manipulate pointers. An entire buffer is typically destroyed in one go.

```
union {
        struct tcphdr   *th;
        struct udphdr   *uh;
        struct icmphdr  *icmph;
        struct igmphdr  *igmph;
        struct iphdr    *ipiph;
        unsigned char   *raw;
} h;

union {
        struct iphdr    *iph;
        struct ipv6hdr  *ipv6h;
        struct arphdr   *arph;
        unsigned char   *raw;
} nh;

union {
        struct ethhdr   *ethernet;
        unsigned char   *raw;
} mac;
```

Figure 3.2: sk_buff ⟨include/linux/sk_buff.h⟩

```
char                    cb[48];
⋮
unsigned int            len,
                        data_len,
⋮
unsigned char           pkt_type,
⋮
unsigned short          protocol,
⋮
unsigned char           *head,
                        *data,
                        *tail,
                        *end;
```

Figure 3.3: sk_buff ⟨include/linux/sk_buff.h⟩

implementation. See ⟨include/linux/skbuff.h⟩ for the complete structure.

The cb member is a control buffer which is used by different protocols. This is typically used by protocols to store private protocol specific variables. The IPv6 module uses it to store the offsets of extension headers from the main header, except for the member iff which is used to store the index of the device on which the packet was received. The IPv6 specific cb is called inet6_skb_parm and is shown in figure 3.4.

```
struct inet6_skb_parm
{
        int                     iif;
        __u16                    ra;
        __u16                    hop;
        __u16                    auth;
        __u16                    dst0;
        __u16                    srcrt;
        __u16                    dst1;
};
```

Figure 3.4: inet6_skb_parm ⟨include/linux/ipv6.h⟩

3.2 skb API

The buffer implementation of Linux comes with a great API. The primary goal of this API is to provide a consistent and efficient buffer-handling mechanism for all of the network layers [4]. There are two sets of functions in the API. One to manipulate the buffers and the memory attached to them and another one to manipulate the doubly linked list of buffers. Some auxiliary functions are also provided which act on various parts of the packet and can be used by the protocols. The API is implemented and fairly well documented in the files shown in table 3.1[1].

file	Description
include/linux/skbuff.h	Structure declarations, macros and API declarations
net/core/skbuff.c	Implementation of main API functions
net/core/datagram.c	Some auxiliary functions

Table 3.1: sk_buff API related files

[1]One should consider reading these files in order to have a good understanding of protocol implementation and internal peculiarities of buffers.

4 Socket Families[1], Protocols and Packets

4.1 Socket Families

The socket API groups related protocols into a *domain*. This implies both an address family and a set of protocols which implement various socket *types*[2] within a domain. An address family is identified by a constant like `AF_xxx` and a protocol family by a constant like `PF_xxx`. The *INET6* domain or INET6 socket-family includes IPv6, ICMPv6, TCPv6[3], and UDPv6[4]. The INET6 domain is identified by `AF_INET6` and `PF_INET6` which are declared in ⟨include/linux/socket.h⟩. The socket layer keeps track of all socket families in an array of `net_proto_family` structures. This structure is shown in figure 4.1.

```
struct net_proto_family {
        int         family;
        int         (*create)(struct socket *sock, int protocol);
        ⋮
```

Figure 4.1: `net_proto_family` ⟨include/linux/net.h⟩

The socket family is identified by `family`. The `create` function is used to create the socket. There are some more fields which have not been shown. Each family has to declare such a structure, and *register* itself with the socket layer using the `sock_register` function declared in ⟨net/socket.c⟩. The INET6 family specific structure is shown in figure 4.2.

```
struct net_proto_family inet6_family_ops = {
        .family = PF_INET6,
        .create = inet6_create,
};
```

Figure 4.2: `inet6_family_ops` ⟨net/ipv6/af_inet6.c⟩

4.2 Protocols

When a socket is to be created, it is supplied with an argument to identify its type. Various types of sockets supported by the Linux kernel are listed in the `socket(7)` man page[5]. Socket type specifies the communication semantics of the socket. INET6 support three socket types. Stream sockets, datagram sockets and raw sockets. Stream sockets are used for TCPv6, datagram sockets are used for UDPv6 and raw sockets are to send IPv6 packets directly[6]. The socket API specifies some basic

[1] The original BSD name for this concept is domain, which is also what is specified in IEEE 1003.1 standard. *socket-family* is the corresponding concept in Linux. See `socket(2)` and `socket(7)` man pages.

[2] See `socket(2)` man page.

[3] The names TCPv6/UDPv6 are used to specifically mean TCP/UDP over IPv6

[4] And also any other protocol which uses IPv6 and is below the transport layer.

[5] Section 7 man pages for Linux do not come as a part of kernel. They are available as separate package from ftp://ftp.kernel.org.

[6] They are also used internally by protocols such as ICMPv6.

operations on sockets such as *bind*, *accept*, etc. The implementation is free to add more functions. For each such a function, the protocol family can specify which function to call within the kernel[1] for each type of socket it supports. This is done via a structure called `proto_ops`. Three important members of `proto_ops` are shown in figure 4.3

The `sendmsg` and `rcvmsg` functions are used to send packets from the protocol family layer to the socket layer, and receive packets from the socket layer to the protocol family layer respectively. INET6 declares two `proto_ops` structures. One is `inet6_stream_ops` for stream sockets, and the other one is `inet6_dgram_ops` for datagram and raw sockets. These are shown in appendix B.1.

```
struct proto_ops {
        int             family;
        ⋮
        int             (*sendmsg)   (struct kiocb *iocb, struct socket *sock,
                                      struct msghdr *m, int total_len);
        int             (*recvmsg)   (struct kiocb *iocb, struct socket *sock,
        ⋮
};
```

Figure 4.3: `proto_ops` ⟨include/linux/net.h⟩

4.3 Packets

When a packet arrives on the interface, the device driver receives it. After processing the packet it has to place the packet in the input queue of the appropriate protocol. Various data link level protocols define ways of identifying network protocols. For example, IEEE 802.3 identifies IPv6 packets with a constant 0x86DD (hex) in the ethernet header. The constant in Linux is called `ETH_P_IPV6` and is defined in ⟨include/linux/if_ether.h⟩.

packet_type

```
struct packet_type
{
        unsigned short          type;   /* This is really htons(ether_type) */
        struct net_device      *dev;    /* Device on which packet can be recieved
                                           NULL is a wildcard here       */
        int                   (*func) (struct sk_buff *, struct net_device *,
                                       struct packet_type *);
                                        /* Packet Type Handler. */
        void                   *data;
                                        /* Private to the packet type */
        struct packet_type     *next;
};
```

Figure 4.4: `packet_type` ⟨include/linux/netdevice.h⟩

The interface layer defines a structure called `packet_type` (figure 4.4). This structure declares a function which is used to transfer the control of the packet from the interface layer to the protocol

[1] The kernel provides a default do nothing function!

layer. For the interface layer all INET6 packets are of IPv6 type. The INET6 packet type structure is shown in figure 4.5. It specifies to the interface layer that packets of type `ETH_P_IPV6` are handled by `ip6_rcv`.

```
static struct packet_type ipv6_packet_type =
{
        .type = __constant_htons(ETH_P_IPV6),
        .dev  = NULL,                           /* All devices */
        .func = ipv6_rcv,
        .data = (void*)1,
};
```

Figure 4.5: `ipv6_packet_type` ⟨net/ipv6/ipv6_sockglue.c⟩

5 INET6

Although INET6 is a family of protocols which sends IPv6 packets to the interface layer, neither IPv6 is completely a part of the INET6 *framework*, nor the entire INET6 framework is a part of IPv6. In this section we look at organization of various protocols in INET6.

5.1 INET6 Layer

The entire INET6 family together with its protocols can be seen as a protocol layer in the kernel (see figure 5.1).

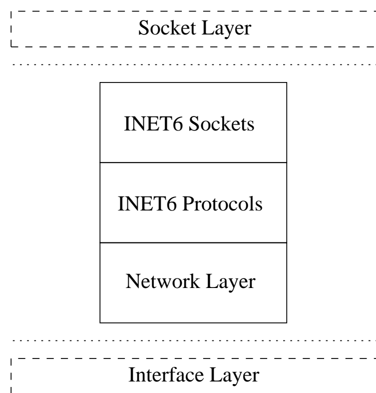

Figure 5.1: INET6 layer

INET6 sockets are sockets belonging to the INET6 family. As mentioned before INET6 supports stream, datagram and raw sockets. INET6 protocols represent various IPv6 packet *types*. Extension headers are also treated as different protocol inside INET6 layer. The network layer take packets from INET6 protocols, add the IPv6 header to them, and sends them to the interface layer. This layer is also responsible for receiving incoming packets from the interface layer and sending them to INET6 protocols.

5.2 inet6_protocol Structure

INET6 keeps track of various protocols in the INET6 layer, in an array of inet6_protocol (figure 5.2) structures called inet6_protos.

```
struct inet6_protocol
{
        int      (*handler)(struct sk_buff **skb, unsigned int *nhoffp);
        void     (*err_handler)(struct sk_buff *skb, struct inet6_skb_parm *opt,
                               int type, int code, int offset, __u32 info);
        unsigned int    flags;  /* INET6_PROTO_xxx */
};
```

Figure 5.2: inet6_protocol ⟨include/net/protocol.h⟩

Every protocol within the INET6 layer has to declare this structure and register it with the INET6 layer using the inet6_register_protosw function. When an IPv6 packet arrives on the interface it is passed to the IPv6 input processing function. This function gives the control of the packet to the appropriate protocol through the corresponding handler. Protocols declared currently in the INET6 layer are shown in table 5.1.

inet6_protocol	Description	file
destopt_protocol	Destination header processing	net/ipv6/exthdrs.c
rthdr_protocol	Routing header processing	net/ipv6/exthdrs.c
icmpv6_protocol	ICMPv6 packet processing	net/ipv6/icmp.c
frag_protocol	Fragmentation processing	net/ipv6/reassembly.c
tcpv6_protocol	TCP packet processing	net/ipv6/tcp_ipv6.c
esp6_protocol	Encapsulation header processing	net/ipv6/esp6.c
udpv6_protocol	UDP packet processing	net/ipv6/udp.c
ah6_protocol	Authentication header processing	net/ipv6/ah6.c

Table 5.1: INET6 protocols

5.3 proto Structure

The proto structure is the transition point between the socket layer and the protocol layer. It specifies functions which are used to communicate with the proto_ops structure in the socket layer. INET6 declares separate proto structures for TCP, UDP and RAW sockets (see table 5.2).

proto	Description	file
rawv6_prot	RAW proto structure	net/ipv6/raw.c
tcpv6_prot	TCP proto structure	net/ipv6/tcp_ipv6.c
udpv6_prot	UDP proto structure	net/ipv6/udp.c

Table 5.2: proto structures for different socket types in IPv6

6 IPv6

IPv6 alone does not make a host IPv6 capable. That is because ICMPv6, ND, PMTU, etc are integral parts of IPv6. In order to be able to use any application level feature, every IPv6 implementation needs to have TCPv6 and UDPv6. Moreover a complete IPv6 implementation requires IPSec as well. A working IPv6 host needs a complete set of user level tools to be IPv6 aware. Finally for an IPv6 host to be used in the current network environment, it would need implementation of mechanisms to communicate with IPv4 hosts.

6.1 IPv6 Addresses

An IPv6 address is an 128-bit address stored in network byte order in a structure called `in6_addr` (figure 6.1). The main IPv6 address is an array of sixteen 8-bit elements. The embedded union is used to force the desired alignment level. See [7] for further information about IPv6 addressing.

```
struct in6_addr {
        union {
                __u8            u6_addr8[16];
                __u16           u6_addr16[8];
                __u32           u6_addr32[4];
        } in6_u;
#define s6_addr                 in6_u.u6_addr8
#define s6_addr16               in6_u.u6_addr16
#define s6_addr32               in6_u.u6_addr32
};
```

Figure 6.1: `in6_addr` ⟨include/linux/in6.h⟩

Within the socket layer this address is carried by the `sockaddr_in6` structure which is shown in figure 6.2. This is implemented in such a way that it can be cast into the protocol independent `sockaddr` structure [5].

```
struct sockaddr_in6 {
        unsigned short int      sin6_family;    /* AF_INET6 */
        __u16                   sin6_port;      /* Transport layer port # */
        __u32                   sin6_flowinfo;  /* IPv6 flow information */
        struct in6_addr         sin6_addr;      /* IPv6 address */
        __u32                   sin6_scope_id;  /* scope id (new in RFC2553) */
};
```

Figure 6.2: `sockaddr_in6` ⟨include/linux/in6.h⟩

6.1.1 Host Address Configuration

Linux supports auto-configuration of host addresses [11]. This is enabled by default. When INET6 is initialized in the kernel, the `addrconf_init` function declared in ⟨net/ipv6/addrconf.c⟩ is called. This function auto-configures all network devices.[1]. The configuration protocol is implemented in

[1] Linux 2.5.69 auto-configuration facility is available only for ethernet devices.

the same file. A manual configuration of IPv6 addresses can be done using the `ifconfig` utility[1].

6.2 IPv6 Header

The IPv6 header is described in [6], and it is represented by the `ipv6hdr` structure which can be seen in figure 6.3.

```
struct ipv6hdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u8                    priority:4,
                                version:4;
#elif defined(__BIG_ENDIAN_BITFIELD)
        __u8                    version:4,
                                priority:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
        __u8                    flow_lbl[3];
        __u16                   payload_len;
        __u8                    nexthdr;
        __u8                    hop_limit;
        struct  in6_addr        saddr;
        struct  in6_addr        daddr;
 };
```

Figure 6.3: `ipv6hdr` ⟨include/linux/ipv6.h⟩

There is an inconsistency. The traffic class consists of the first four bits of `flow_lbl` and the four bits of `priority`. This will be changed soon. The `version` field is always set to `6`. The length of payload including the extension headers (if any) is specified in `pay_load_len`. The type of the next header is stored in `nexthdr`. If the host forwards packets then it has to decrement the value of `hoplimit` and discard the packet if it is 0. The 128-bit source address and destination address are stored in `saddr` and `daddr` respectively. Note that if the routing header is specified, the destination address is not necessarily the final recipient.

6.2.1 Extension Headers

Most of the extension headers are also declared in ⟨include/linux/ipv6.h⟩. Linux treats extension headers as separate protocols inside the INET6 layer. Table 5.1 shows all the different extension headers supported by Linux and the files in which they are implemented. We do not go into more details about any of the extension-header processing.

6.3 IPv6 Output Processing

The core IPv6 output processing routines are declared in ⟨net/ipv6/ip6_output.c⟩. The basic job of these functions is to fill the IPv6 header and transmit the packet to the interface layer. Even

[1]This utility is a part of the net-tools package available from http://www.tazenda.demon.co.uk/phil/net-tools and is *not* part of the Linux kernel.

though there could be in principle only one function to handle this, the code is distributed and even repeated whenever doing so is beneficial to performance. Some functions that are called while transmitting a TCP packet are shown below.

function	Description
ip6_xmit	Receives the packet from TCP module, fills the IPv6 header and calls ip6_output
ip6_output	Sends the packet to loopback and/or calls ip6_output_finish
ip6_output_finish	Fills the final destination address and sends the packet to the interface

There are more functions for building the fragmentation header, for sending raw packets, for forwarding packets if the routing option is enabled, etc.

6.4 Input Processing

Section 4.3 mentions that IPv6 packets are received from the interface layer by the ip6_rcv function declared in ⟨net/ipv6/ip6_input.c⟩. This function performs various sanity checks on the packet, for example, it checks if the protocol version of the packet is correct, if the header length is correct, if the packet is complete, etc. If any of the sanity checks fail, the packet will be dropped. If packet forwarding is enabled, then packets with nexthop header may be forwarded, else they will also be dropped. The control then goes to the ip6_input_finish function, which either raw delivers the packet, or sends the packet to the appropriate registered protocol, using the inet6_protocol structure or it just drops the packet.

7 Other Protocols

7.1 ICMPv6

ICMPv6 is an integral part of IPv6 implementation. ICMPv6 implementation follows [3]. It deals with various informational and error messages exchanged between hosts which communicate using IPv6. ICMPv6 packets are IPv6 packets with next header value set to 58. In Linux, ICMPv6 is a separate INET6 protocol. It registers itself with INET6 protocols, and defines a packet handler icmpv6_rcv for incoming ICMPv6 packets. This function processes the incoming ICMPv6 packets and takes appropriate action[1]. Various other functions are declared are used to construct and send send ICMPv6 messages to other hosts.

7.2 Neighbor Discovery

Neighbor Discovery defines the protocol for router discovery, prefix discovery, address resolution, neighbor reachability and route redirection [10]. The protocol uses ICMPv6 and specifies a set of ICMPv6 message formats for this purpose.

[1] Reply to echo request, take corrective steps for error messages or ignore.

Neighbor discovery is not treated as a different protocol in Linux. While initialization a special raw socket, `ndisc_socket` is created. All message transmission is done through this socket. For incoming packets, the `icmpv6_rcv` function calls `ndisc_rcv` function and hands the buffer with packet to this function, which takes appropriate action. The functions specific to neighbor discovery are declared in ⟨net/ipv6/ndisc.c⟩.

7.3 IPSec

IPSec in Linux is under development. A Basic support for both authentication header and encapsulation header is available[1]. There is a generic crypto API under development which will provide encryption facilities in the kernel. There are some *unofficial* implementations of IPSec available[2]

7.4 Tunneling and Translation Mechanisms

Linux can be easily used as a dual stack machine. There are user space tools to set up tunnels. See IPv6-Howto[3] available at http://www.tldp.org for more details on how to setup tunnels, etc. A general IPv6 over IPv4 *device* is implemented in ⟨net/ipv6/sit.c⟩. This device is initialized at the time of INET6 initialization. Many user space tools used by distributions[4] based on the Linux kernel are already IPv6 aware[5] and others are in process of migrating.

8 Conclusion

This project gave us the opportunity to understand better and to take a closer look at how it is an actual IPv6 implementation. Studying a developing version of Linux was a challenging task in itself. There were frequent changes in the source code which added both to our confusion and clearing some doubts. The RFC specifications were also very useful to understand the source code.

A networking implementation needs to be efficient, besides supporting the major network protocols, and it tends to share information across the layers in order to improve performance. This makes the concept of layering in the protocol stack quite insignificant in implementation.

Linux networking implementation is fairly object oriented, that is why adding a new protocol in the Linux kernel is relatively easy[6]. The design is efficient and its subsystems are well distributed into functions that are thread safe. Most of these functions share common data with *copy on write*.

The Linux `sk_buff` API is extremely efficient. Linux network buffers are a bit complicated to understand, and can waste a few bytes of memory as well. However the gain in performance is worth wasting this space.

There are some interesting points which we outline below.

[1] Both are available at compile time as separated modules.

[2] Freeswan is the major one. See http://www.freeswan.org for more details on it.

[3] Out-of-date, but useful. Also see man page of `ip` utility from the `iproute` package.

[4] Gentoo, RedHat, Mandrake etc.

[5] They have to be compiled with IPv6 enabled.

[6] Making a protocol RFC compliant takes effort. Making it RFC compliant and efficient takes a lot of effort!

Netfilter

Linux has a general purpose packet filter called Netfilter. One can specify a very complex set of rules to filter packets. Netfilter *hooks* are scattered in the networking source code, and do nothing if Netfilter is disabled.

IPv4 and IPv6

Linux can also be used as a router. It provides a general purpose IPv6 over IPv4 tunnel device which can be used to connect other IPv6 aware hosts in the IPv4 network. As IPv4 and IPv6 can run together on the Linux kernel, a Linux machine can be used as a dual stack host or router and migrating from IPv4 to IPv6 in Linux is not a problem.

Spinlocks

A lot of networking code which share data use spinlocks. Hence it becomes critical to ensure that the code between spinlock is very small and does not sleep at all.

Word boundary

It is a good idea to develop protocols in which packet is aligned with word boundary. Most implementations add extra zeros to the end of the packet[1].

Caches

Caches are used for quick lookup of destination addresses. They are built up while looking for a route to the destination. The TCP module generally adds this information as a hint to IP(v6) modules for packets which are sent later on. When a connection is established, some hosts which communicated with each other before may need to communicate with each other again. Having cache mechanisms improves the performance a lot because the necessary information is already stored.

Fragmentation

Fragmentation can cause buffer overflow which can lead to a severe decrease in performance. It also affects the rate at which a host can transmit packets. From implementation point of view, it is a good decision to not fragment IPv6 packets on routers.

Checksum

Error checking is done in other layers, and it is not necessary an additional error checking in the IPv6 protocol. That is why there is no checksum in IPv6. It avoids the protocol to make an extra pass on the packet and thus it improves the performance.

[1] That is one of the reasons why the payload length field is needed in the header.

IPSec

One of the major reasons why IPSec is still not fully developed in Linux, is that it interfaces badly with the kernel. The distribution of Crypto API is a step forward in the direction of making IPSec implementation as efficient as the rest of the networking subsystem.

Future Work

We believe that this report is a good start point to anyone who is interested in making changes with IPv6 or add/remove any feature to/from IPv6. There are some areas in which this report can be extended.

a. We do not describe much about IPSec. A study of IPSec implementation is worthwhile to pursue.

b. We could not perform any test such as to how well the implementation works. Testing the protocol can be a very challenging task as one needs to design and send carefully crafted packets to a host in order to study their behavior.

c. We could not also look at implementation in great detail to see if all the parts are RFC compliant.

The main source of Linux kernel documentation is its source code itself. It is expected that Linux 2.6 will be fairly well documented. Other good sources of documentation are [12] [2], [1] and [8]

A Linux Source Code Organization

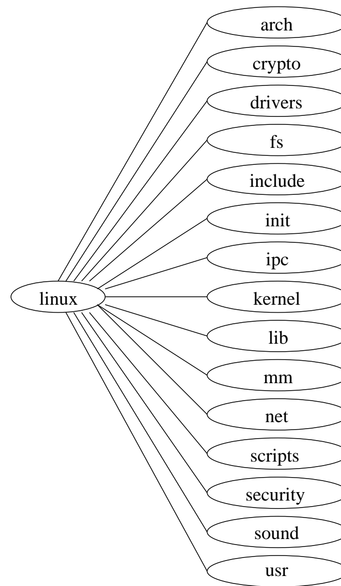

Figure A.1: ⟨linux⟩ Source Code Organization

The main directory concerning networking code is ⟨net⟩. The header files specific for Linux networking implementation are in ⟨include/net⟩. The directory ⟨include/linux⟩ contains header declarations which can be used by application layer to interact with the kernel.

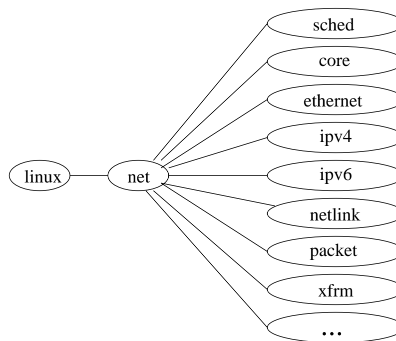

Figure A.2: ⟨linux/net⟩ Source Code Organization

In the ⟨net⟩ directory there is implementation of the socket subsystem, interface layer and protocols. Network device drivers are kept separately in ⟨drivers/net⟩. INET6 specific files are stored in the ⟨net/ipv6⟩ directory. However it shares a lot of code with IPv4 implementation in the ⟨net/ipv4⟩ directory. The directory ⟨net/core⟩ contains files which implement the generic software layer and the interface layer.

B Structures

B.1 INET6 proto_ops Structures

inet6_stream_ops

```
struct proto_ops inet6_stream_ops = {
        .family =       PF_INET6,

        .release =      inet6_release,
        .bind =         inet6_bind,
        .connect =      inet_stream_connect,          /* ok             */
        .socketpair =   sock_no_socketpair,           /* a do nothing */
        .accept =       inet_accept,                  /* ok             */
        .getname =      inet6_getname,
        .poll =         tcp_poll,                     /* ok             */
        .ioctl =        inet6_ioctl,                  /* must change   */
        .listen =       inet_listen,                  /* ok             */
        .shutdown =     inet_shutdown,                /* ok             */
        .setsockopt =   inet_setsockopt,              /* ok             */
        .getsockopt =   inet_getsockopt,              /* ok             */
        .sendmsg =      inet_sendmsg,                 /* ok             */
        .recvmsg =      inet_recvmsg,                 /* ok             */
        .mmap =         sock_no_mmap,
        .sendpage =     tcp_sendpage
};
```

inet6_stream_ops

```
struct proto_ops inet6_dgram_ops = {
        .family =       PF_INET6,

        .release =      inet6_release,
        .bind =         inet6_bind,
        .connect =      inet_dgram_connect,           /* ok             */
        .socketpair =   sock_no_socketpair,           /* a do nothing */
        .accept =       sock_no_accept,               /* a do nothing */
        .getname =      inet6_getname,
        .poll =         datagram_poll,                /* ok             */
        .ioctl =        inet6_ioctl,                  /* must change   */
        .listen =       sock_no_listen,               /* ok             */
        .shutdown =     inet_shutdown,                /* ok             */
        .setsockopt =   inet_setsockopt,              /* ok             */
        .getsockopt =   inet_getsockopt,              /* ok             */
        .sendmsg =      inet_sendmsg,                 /* ok             */
        .recvmsg =      inet_recvmsg,                 /* ok             */
        .mmap =         sock_no_mmap,
        .sendpage =     sock_no_sendpage,
};
```

All the functions `inet6_xxx` are declared in ⟨net/ipv6/af_inet6.c⟩. However the functions prefixed `inet_xxx` are from ⟨net/ipv4/af_inet.c⟩. This is so because a lot of INET6 code share code with INET code.

B.2 INET6 proto Structures

tcpv6_prot

```
  struct proto tcpv6_prot = {
        .name           =       "TCPv6",
        .close          =       tcp_close,
        .connect        =       tcp_v6_connect,
        .disconnect     =       tcp_disconnect,
        .accept         =       tcp_accept,
        .ioctl          =       tcp_ioctl,
        .init           =       tcp_v6_init_sock,
        .destroy        =       tcp_v6_destroy_sock,
        .shutdown       =       tcp_shutdown,
        .setsockopt     =       tcp_setsockopt,
        .getsockopt     =       tcp_getsockopt,
        .sendmsg        =       tcp_sendmsg,
        .recvmsg        =       tcp_recvmsg,
        .backlog_rcv    =       tcp_v6_do_rcv,
        .hash           =       tcp_v6_hash,
        .unhash         =       tcp_unhash,
        .get_port       =       tcp_v6_get_port,
};
```

udpv6_prot

```
  struct proto udpv6_prot = {
        .name =         "UDP",
        .close =        udpv6_close,
        .connect =      udpv6_connect,
        .disconnect =   udp_disconnect,
        .ioctl =        udp_ioctl,
        .destroy =      inet6_destroy_sock,
        .setsockopt =   ipv6_setsockopt,
        .getsockopt =   ipv6_getsockopt,
        .sendmsg =      udpv6_sendmsg,
        .recvmsg =      udpv6_recvmsg,
        .backlog_rcv =  udpv6_queue_rcv_skb,
        .hash =         udp_v6_hash,
        .unhash =       udp_v6_unhash,
        .get_port =     udp_v6_get_port,
};
```

udpv6_prot

```
struct proto rawv6_prot = {
        .name =         "RAW",
        .close =        rawv6_close,
        .connect =      udpv6_connect,
        .disconnect =   udp_disconnect,
        .ioctl =        rawv6_ioctl,
        .init =         rawv6_init_sk,
        .destroy =      inet6_destroy_sock,
        .setsockopt =   rawv6_setsockopt,
        .getsockopt =   rawv6_getsockopt,
        .sendmsg =      rawv6_sendmsg,
        .recvmsg =      rawv6_recvmsg,
        .bind =         rawv6_bind,
        .backlog_rcv =  rawv6_rcv_skb,
        .hash =         raw_v6_hash,
        .unhash =       raw_v6_unhash,
};
```

The `xxx6_xxx` and `xxx_v6_xxx` functions are INET6 specific, while others are shared from INET.

C Control Flow

C.1 Output Processing Control Flow

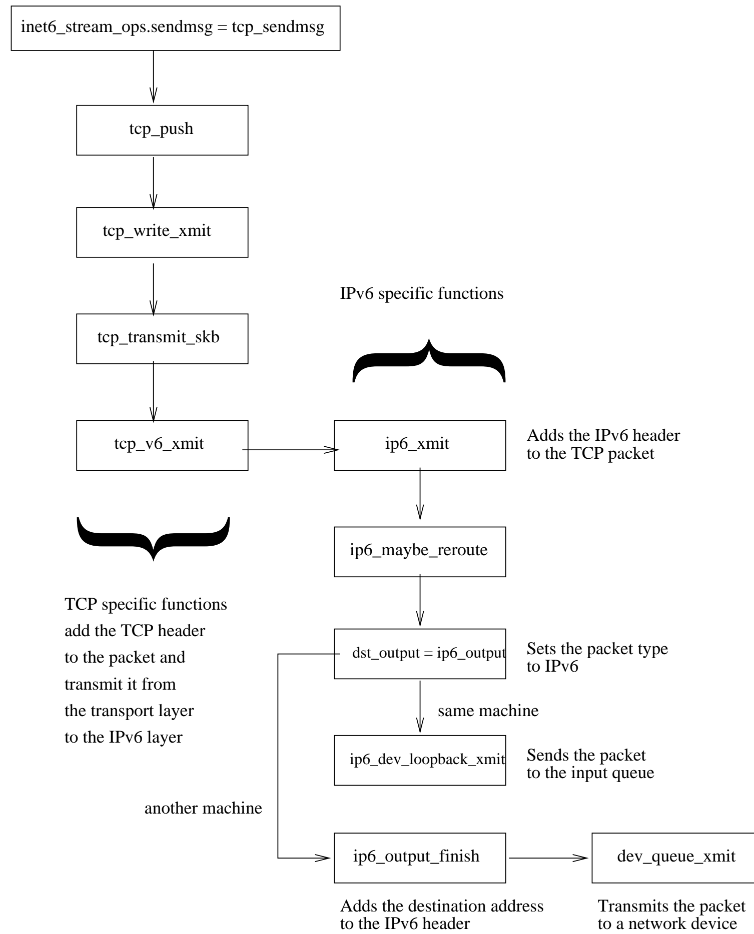

Figure C.1: Output-processing control flow

The `tcp_sendmsg` and `tcp_push` functions are placed in ⟨net/ipv4/tcp.c⟩, and the `tcp_write_xmit` and `tcp_transmit_skb` functions in ⟨net/ipv4/tcp_output.c⟩. Moreover the `tcp_v6_xmit` function can be found in ⟨net/ipv6/tcp_ipv6.c⟩, and finally the `ip6_xmit` function is in ⟨ip6_output⟩.

C.2 Input Processing Control Flow

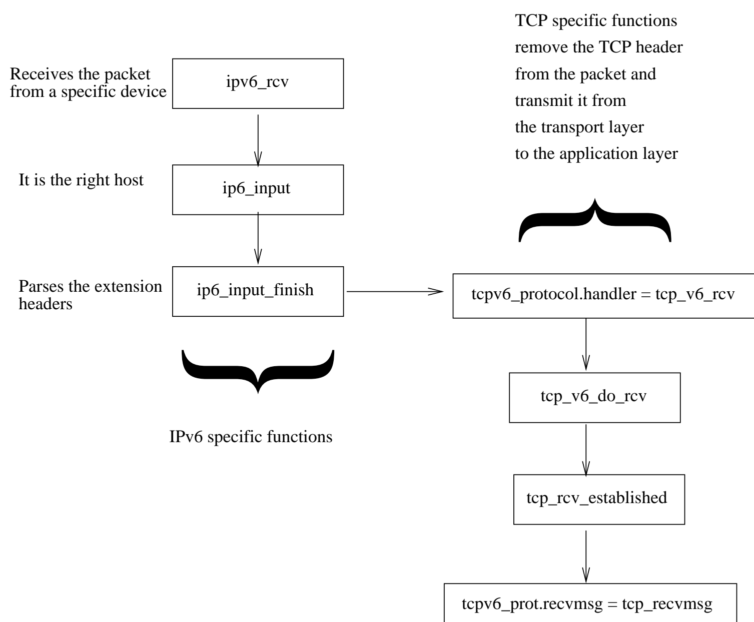

Figure C.2: Input-processing control flow

It is worth explaining some of the functions. The `tcp_rcv_established` function is responsible for receiving packets on sockets with established state, it also queues the packets in sequence at the receive queue and out-of-sequence packets at the out-of-order queue. The `tcp_recvmsg` function is responsible for reading packets from the receive queue and giving them to the application layer.

The `tcp_v6_rcv` and `tcp_v6_do_rcv` functions can be found in ⟨net/ipv6/tcp_ipv6.c⟩. Moreover the `tcp_rcv_established` function is placed in ⟨net/ipv4/tcp_input.c⟩, and the `tcp_recvmsg` function in ⟨net/ipv4/tcp.c⟩.

References

[1] Tigran Aivazian. Linux kernel 2.4 internals. Available at http://tldp.org/guides.html.

[2] Roberto Arcomano. Kernelanalysis-howto. Available at http://www.tldp.org/HOWTO/KernelAnalysis-HOWTO.html.

[3] A. Conta and S. Deering. RFC 2463: Internet control message protocol (icmpv6) for the internet protocol version 6 (ipv6) specification.

[4] Alan Cox. Network buffers and memory management. *Linux Journal*, September 1996.

[5] R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens. RFC 3493: Basic socket interface extensions for ipv6, February 2003.

[6] R. Hinden and S. Deering. RFC 2460: Internet protocol, version 6 (ipv6) specification.

[7] R. Hinden and S. Deering. RFC 3513: Internet protocol version 6 (ipv6) addressing architecture, April 2003.

[8] David A. Rusling. The linux kernel. Available at http://tldp.org/guides.html.

[9] W. Richard Stevens and Gary R. Wright. *TCP/IP Illustrated, Volume 2*. Addison-Wesley, 1995.

[10] E. Nordmark T. Narten and W. Simpson. RFC 2461: Neighbor discovery for ip version 6 (ipv6).

[11] S. Thomson and T. Narten. RFC 2462: Ipv6 stateless address autoconfiguration.

[12] Alavoor Vasudevan. The linux kernel howto. Available at http://www.tldp.org/HOWTO/Kernel-HOWTO/index.html.