IPv6-DRET Implementation Guide

Pascal ANELLI (Pascal.Anelli@masi.ibp.fr)
Benoit BRODARD(Benoit.Brodard@inria.fr)
FranK ZAGO (Franck.Zago@masi.ibp.fr)

December, 1996

Contents

Introduction

IPv6-DRET is a public implementation of the next generation Internet Protocol, IPv6, funded by the DGA/DRET (French Military Research Agency) and co-developped by INRIA Sophia-Antipolis and MASI laboratories.
Fig. 0.1 shows the scope of the project which includes the implementation of the IPv6 network protocol for the Linux operating system and the implementation of the RIPng routing protocol inside the GateD daemon.

0.1 IPv6

IPv6 is the new version of the Internet Protocol. It should replace progressively the current IPv4 routing protocol which is responsible for interconnecting the gateways of the thousands of networks across the worldwide Internet.
IPv6 was primarily designed to extend the current address space which has become too small to accomodate the increasing growth of the network. IPv6 addresses are 128-bit long, thus multiplying the potential Internet size by a factor of 296. This allows for different addressing schemes, new types of applications, etc... As of January 97, the addressing scheme to be used is actively discussed by the IETF working groups.
IPv6 specifications have also been tuned to support the following:

- Plug-and-play configuration
- Efficient packet handling/routing
- End-to-end flow identification for integrated service
- Standard security mechanisms
- Mobility
- Smooth IPv4-to-IPv6 transition.

While native IPv6 stacks and native IPv4 stacks will not be able to interoperate directly, IPv4-compatibility mechanisms have been introduced in IPv6 to ensure a smooth transition, i.e. allow isolated IPv6 islands to communicate via the IPv4 internet through "tunnels".

0.2 About this document

This guide is intended to be the reference tool to contribute to the IPv6-DRET source code. In each chapter, global concepts are first introduced, followed by the list of the corresponding source files. Then, we present in detail data structures and procedures using a line-by-line style approach.
The remainder of the implementation guide is structured as follows. Chapter 1 describes IPv6 basics: raw sockets, etc... The IPv6-DRET routing mechanisms are presented in Chapter 2. Chapter 3 contains the documentation of the Neighbor Discovery (ND) code for unreachability detection and address resolution. The ICMPv6 protocol implementation is further explained in Chapter 4. As was mentioned above, a Simple Interface Transition interface is needed to insure reverse compatibility with the IPv4 protocol. The SIT

IPv6-DRET Implementation Status
December 10, 1995

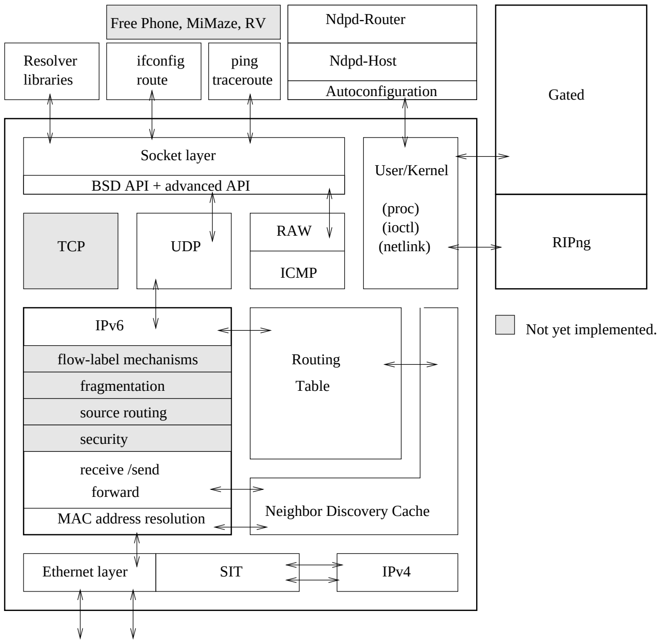

Figure 0.1: The IPv6-DRET project

interface is presented in Chapter 5.

Chapter 1

IPv6 mechanisms

This section introduces the IPv6 networking mechanisms. This portion of the stack is responsible for processing outgoing packets from the transport layer or a user application downward to the interface layer. Incoming packets are taken from a packet queue, processed according to the options and flags set in the IPv6 header and finally demultiplexed to a suitable protocol or user application. The most generic IPv6 features are presented here, while more specific mechanisms such as multicast and fragmentation will be explained in an other section .

1.1 Code introduction

File	Description
linux/in.h	Generic IP declarations
linux/ipv6.h	IPv6 specific declarations
net/ipv6.h	IPv6 specific declarations
ipv6/ipv6_output.c	IPv6 output functionalities
ipv6/ipv6_input.c	IPv6 input functionalities

1.2 File : include/linux/ipv6.h

This file contains the declaration of the IPv6 packet.

```
    struct ipv6hdr {
            __u32 vrprflow;             /* ipv6 version, priority and flow label */
            __u16 payload;
            __u8  next_hdr;
            __u8  hop_limit;
            struct in6_addr saddr,      /* ipv6 source address */
                            daddr;      /* ipv6 destination address */
};
```

1.3 Global variables

Variable	Datatype	Origin	Description
ip6_statistics			snmp structures where IPv6 statistics are stored

1.4 Statistics

Name	Description
IpInReceives	# of IPv6 packets received
IpInHdrErrors	# of IPv6 packets arrived with erroneous header
IpInAddrErrors	# of IPv6 packets arrived with erroneous address
IpOutRequests	# of IPv6 output requests
IpOutNoRoutes	# of IPv6 packets without any route
IpOutDiscards	# of outgoing packets discarded

1.5 IPv6 packets

-Describe IPv6 packets: TCP, UDP, raw + header

-Packet type as seen from the interface layer:

```
static struct packet_type ipv6_packet_type =
{
        0,      /* MUTTER ntohs(ETH_P_IPV6),*/
        NULL,   /* All devices */
        ipv6_rcv,
        NULL,
        NULL,
};
```

1.6 ipv6_rcv Function

1. **Description** This function processes a network buffer and sends it upward to either the protocol layer or a user application.
2. **Input parameters**

Variable	Origin	Description
*skb	linux/sk_buff.h	sk_buff containing incoming packet with IPv6 header
*dev	linux/netdevice.h	main device on which the packet was received.
pt	linux/netdevice.h	packet type

3. **Output parameters**

Variable	Origin	Description
*skb	linux/sk_buff.h	sk_buff containing packet sent to protocol layer.
dev	linux/netdevice.h	actual interface the packet was sent to.

4. **Returns** 0.

ipv6_input.c

```
102 int ipv6_rcv(struct sk_buff *skb, struct device *dev, struct packet_type *pt)
103 {
104 struct sock *raw_sk=NULL;                       /* raw socket         */
105         struct inet6_protocol *ipprot;                  /* inet prot. struct. */
106         struct ipv6hdr *iph = skb->h.ipv6h;
107 struct in6_addr d;
108
109         unsigned char hash;
110 unsigned char flag = 0;
111
112 #ifdef IPV6_DEBUG
113 printk("[ipv6_rcv] IN");
```



```
114 #endif
115
116         ip6_statistics.IpInReceives++;
117
118 /*
119  * Tag the ip header of this packet so we can find it
120  */
121         skb->ipv6_hdr = iph;
122
123 /*
124  *
125  * Is the datagram acceptable?
126  *
127  * 1. Length at least the size of an ip header
128  * 2. Version = 6.
129  * 3. Doesn't have a bogus length
130  *      for now...
131           */
132 if (skb->len<sizeof(struct ipv6hdr) || ((IPV6_VERSION & iph->vrprflow)!=IPV6_VERSION) || skb->
133 {
134 ip6_statistics.IpInHdrErrors++;
135 kfree_skb(skb, FREE_WRITE);
136 #ifdef IPV6_DEBUG
137 printk("[ipv6_rcv] OUT, IpInHdrErrors");
138 #endif
139 return(0);
140 };
```

ipv6_input.c

102-140 This section initialize `iph` and `skb->ipv6_hdr` to point at the beginning of the data region in the buffer. It actually corresponds to the beginning of the IPv6 header. Then, routine checks are performed. The incoming packet is discarded if its length is smaller than the minimum IPv6 header length, if its version is different from 6, or if its size is smaller than the announced payload.

152-163 Next, the network buffer is trimmed to get rid of any extra bytes added by the lower layers for padding. According to the IPv6 payload definition, a data area of length equal to the packet payload plus header size is conserved. In case the device that received the packet has more than one interface, i.e. protocol address, the device structure corresponding to the destination address of the incoming packet is retrieved using the `net_alias_*` functionalities.

ipv6_input.c

```
152 skb_trim(skb,(ntohs(iph->payload)+sizeof(struct ipv6hdr)));
153
154 /*
155  * Try to select closest <src,dst> alias device, if any.
156  * net_alias_dev_rcv_sel32 returns main device if it
157  * fails to found other.
158
159 #ifdef CONFIG_NET_ALIAS
160 if (iph->daddr != skb->dev->pa_addr && net_alias_has(skb->dev))
161 skb->dev = dev = net_alias_dev_rcv_sel32(skb->dev, AF_INET, iph->saddr, iph->daddr);
162 #endif
163 */
```

ipv6_input.c

167-218 If the packet was sent to us, it is delivered to the protocols matching the header field `next_hdr`. This is achieved by two loops. The code first loops over the list of raw sockets accessed in `sock_array` with index `hash` (l.196-218). If the list is not empty, every socket in the list is checked through `get_sock_raw6()` to determine if it should be passed a copy of the incoming packet. For all such sockets except the first one, a copy of the buffer is made via `skb_clone` and delivered to the socket by `raw_rcv6()`.

ipv6_input.c

```
167 COPY_ADDR6(iph->daddr, d);
168         if ( SAME_ADDR6(iph->daddr,skb->dev->pa_addr)) /*|| (brd = ip_chk_addr(iph->daddr)) !=
169 /* daddr = device interface address. Packet is for us. */
170 {
171 /*
172  * Point into the IP datagram, just past the header.
173  */
174 skb->ipv6_hdr = iph;
175 skb->h.raw += sizeof(struct ipv6hdr);
176                 /*
177  *  skb->h.raw now points at the protocol beyond the IP header.
178  */
179
180                 /*
181  * Deliver to raw sockets.
182  */
183 hash = iph->next_hdr & (SOCK_ARRAY_SIZE-1);
184
185 /*
186  *  Is there a raw socket?
187  */
188 if((raw_sk=raw_prot.sock_array[hash])!=NULL)
189 {
190 struct sock *sknext=NULL;
191 struct sk_buff *skb1=NULL;
192 raw_sk=get_sock_raw6(raw_sk, iph->next_hdr,
193                                                         &iph->saddr, &iph->daddr);
194 if(raw_sk)                  /* At least one... */
195 {
196 do
197 {
198 /* Others?... */
199 sknext=get_sock_raw6(raw_sk->next,
200                                                                 iph->next_hdr,
201                                                                 &iph->saddr,
202                                                                 &iph->daddr);
203 if(sknext)
204 {
205 skb1=skb_clone(skb,GFP_ATOMIC);
206 if(skb1)
207         raw_rcv6(raw_sk, skb1, dev,
208                                                                     &iph->saddr, &d);
209                                         }
210 else
211 /*
212                                         * One pending raw socket left to
213                                         * avoid 1 copy.
214                                         */
215                                              break;
216 raw_sk=sknext;
217 }
```



```
218 while(raw_sk!=NULL);
```

ipv6_input.c

228-275 Once all raw socket have been serviced except the first one, the code loops over the protocol list (l. 229-259) accessed in the array `inet6_protos` at index `hash`. If the current internet protocol entry, `ipprot`, of the list corresponds to the protocol indicated in the `next_hdr` field of the packet header, the packet is delivered to that protocol layer using the protocol handler provided in `ipprot`. A copy of the buffer is made if a raw socket is still pending, or if the copy bit in `ipprot` is set, indicating that other protocols still need a copy of the packet. Otherwise, `ipprot` indicates the last protocol the packet should be delivered to and `skb` itself is passed to the handler.

Once the loop over all protocols is over, the pending raw_socket, if any, is passed `skb`. The variable `flags` is only there to check that the packet has been delivered at least once. If not, the sk_buff is freed and the corresponding statistic is incremented.

ipv6_input.c

```
228 hash = iph->next_hdr & (MAX_INET_PROTOS -1);
229 for (ipprot = (struct inet6_protocol *)inet6_protos[hash];
230                                 ipprot != NULL;
231                                 ipprot=(struct inet6_protocol *)ipprot->next)
232 {
233 struct sk_buff *skb2;
234 if (ipprot->protocol != iph->next_hdr)
235         /* this is not our protocol */
236 continue;
243 if (ipprot->copy || raw_sk)
244 {
245 skb2 = skb_clone(skb, GFP_ATOMIC);
246 if(skb2==NULL)
247 continue;
248 }
249 else
250 {
251 skb2 = skb;
252 }
253 flag = 1;               /* means we found a matching protocol */
254         /*
255 * Pass on the datagram to that protocol
256                         */
257 ipprot->handler(skb2, dev, NULL /*opt*/, &d, ntohs(iph->payload),
258                                  &iph->saddr, 0, ipprot);
259 }
260 /*
261  * All protocols checked.
262  */
263 if(raw_sk!=NULL) /* Shift to last raw user */
264 raw_rcv6(raw_sk, skb, dev, &iph->saddr, d);
265 else if (!flag) /* Free and report errors */
274 kfree_skb(skb, FREE_WRITE);*/
275 }
```

ipv6_input.c

1.7 ipv6_build_xmit Function

1. **Description** This function handles a network buffer passed by a user application or a transport protocol at the network level. It is responsible for building a correct IPv6 header and passing the buffer down to the Linux kernel device layer.

2. **Input parameters**

Variable	Origin	Description
sock	sock.h	struct
getfrag	raw.h	get a buffer in user space into kernel space
frag	const void	user's buffer
length	unsigned short int	fragment length
d	net/ipv6.h	destination IPv6 address
us	net/ipv6.h	source address provided by user
opt	????	options associated with this packet.
flags	int	flags associated with this packet.
protocol	int	protocol or port number
noblock	int	blocking option

3. **Output parameters**

Variable	Origin	Description
skb	linux/sk_buff.h	network buffer in which the incoming packet is stored.

4. **Returns** 0 or -ERROR, where ERROR can be ENETUNREACH.

5. **Postconditions:** the sk_buff `skb` must be correctly filled in.

`123-133` `ipv6_output` takes 9 parameters in input. Among them , we find the sock structure `*sk` that contains information about the transport and network layers involved in sending packets. The function getfrag, as its name indicates, is responsible for getting and formatting the fragment to send. In the case of raw ip communication, this function must get the data pointed to by `*frag` from user space into kernel space. The length of the data fragment is given by `length`. Other parameters are the destination address, a source address if provided by the user, the IPv6 protocol that must be used and different options. Flags possibly contains MSG_DONTROUTE. (IPv4). The function getfrag will most generally copy data starting at `const void *` to `char *`. The length of copied data is given by the last argument of the function. Report to get_rawv6_frag(), get_icmpv6_frag, etc... for detailed information.

ipv6_output.c

```
204 int ipv6_build_xmit(struct sock *sk,
205                       void getfrag (const void *, struct in6_addr *,
206                                     char *, unsigned int, unsigned int),
207                      const void *frag,
208                      unsigned short int length,        /* fragment's length */
209                      struct in6_addr *d,               /* destination addr. */
210                      struct in6_addr *us,              /* user source addr. */
211                      struct options * opt,      /* keep this for now but... */
212                      int flags,
213                      int protocol,
214                      int noblock)
215 {
%============= code insertion =============
```

ipv6_output.c

`216-` From `*sk` the network protocol information is retrieved: `net_pinfo.af_inet6`. If it already contains a destination, the corresponding destination structure is obtained with `ipv6_check_dest`, otherwise a destination lookup is performed using the destination `*d` passed as parameter and `ipv6_dest_route`. If no route

is found, an error is returned. Destination cache lookup is described in chapter [???]. In the same time, pointer to the neighbor and device structures for that destination are stored in `*hh` and `*dev`.

ipv6_output.c

```
216         struct destination  *rt = NULL, **rt_cache= NULL;
217         struct neighbor *hh = NULL;
218 struct device   *dev = NULL;
219         struct ipv6hdr  *iph;
220 struct ipv6_pinfo *info = &sk->net_pinfo.af_inet6;    /* from sock.h */
221
222 struct in6_addr *s = NULL;                            /* source      */
223         struct in6_addr *r = NULL;                            /* raddr       */
224         /* struct in6_addr *td = d; */ /* true daddr */
225
226         int plen = length;       /* actual packet length, with hdr & options */
227
228 #ifdef IPV6_DEBUG
229 printk("[ipv6\_output] IN");
230 #endif
231 ip6_statistics.IpOutRequests++;                      /*ipv6.h, snmp.h*/
232         /*
233          *  Destination ?
234          */
235         if(info->dest)
236   { /*
237      * check parameters (info->dest)
238             */
239   rt = ipv6_dest_check(info->dest,
240                                 sk->localroute || (flags&MSG_DONTROUTE));
241           }
242         else
243   { /*
244             * find destination structure for *d
245             */
246           rt = ipv6_dest_route(rt_cache, d, sk->localroute);
247           }
248
249
250
251         if (info->dest == NULL)
252 /* No route found... */
253                 {
254                         ip6_statistics.IpOutNoRoutes++;
255                         return(-ENETUNREACH);
256                 }
257         else
258   {
259           info->dest=rt;
260           }
261         /*
262          * Neighbor ?
263          */
264         hh = rt->n;
265          /*
266          * Device ?
267          */
268         dev=hh->dev;
```

ipv6_output.c

`269-293` Then the source address is set, either from the address given by the user, or from the address in `info`. If both methods fail, the IPv6 address list of the device is looked up and a source address is chosen. It must have a scope corresponding to the destination address scope (IFA_HOST, IFA_LINK, IFA_SITE, IFA_GLOBAL). The next hop, or router, address is given by the neigbhor structure. The actual length of the fragment, `plen` is increased by `sizeof(struct ipv6hdr)` if a header must still be added to it. Note that in any case, the IPv6 header length is not taken into account to compute packet payload. Options are not processed as of 05-23-96.

ipv6_output.c

```
269         /* set user source */
270         if (us)
271     P_COPY_ADDR6(us,s);
272         else
273         {
274            if (!IS_ANYADDR6(info->saddr) && (!IS_LOOPADDR6(info->saddr)
275                                                     || IS_LOOPADDR6(*d)))
276               COPY_ADDR6(info->saddr, *s);
277         }
278         if !(s)
279 {
280   /* depending on daddr, get saddr.
281             4 scopes: host, link, site, local
282    */
283         /* set router address */
284         COPY_ADDR6(rt ? hh->naddr : *d, *r);
285
286         /*
287          *      Now compute the buffer space we require
288          */
289         if (!sk->ip_hdrincl)
290    /* header not included... must build it */
291            plen += sizeof(struct ipv6hdr);
292    /* no option for now... */
293
```

ipv6_output.c

`294-328` For now, (may 96), we only service data fragments that do not need fragmentation. In other words, their length, IP header included, must be less than the destination PMTU. First, an sk_buff is allocated, large enough to store the data fragment and the hard header. Its generic fields are filled in: the device pointer, the link protocol type in network byte order, the current time, and the pointer to the sock structure. [1]kb->free... [2]kb->arp... Then, the call to `ipv6_build_mac_header` builds the link header based on the neighbor structure `hh`. A space of size `plen` is reserved in the sk_buff via `skb_put` and is intended to store the data fragment together with the IP header.

ipv6_output.c

```
294         /*
295          *      "Discard" multicast, long packets....
296          */
297         if(plen <= dev->mtu && !IS_MULTIADDR6(*d))
298                                 /* &&  !SAME_ADDR6(*d, dev->pa_brdaddr)*/
299         {
300                 int error;
301                 /*
302                  *  alloc sk_buff for sk. and fill in main fields
```

[1] s

[2] s


```
303                  */
304                 struct sk_buff *skb=sock_alloc_send_skb(sk,
305                            plen+15+dev->hard_header_len, 0, noblock, &error);
306                 if(skb==NULL)
307                 {
308                         ip6_statistics.IpOutDiscards++;
309                         return error;
310                 }
311                 skb->dev=dev;
312                 skb->protocol = htons(ETH_P_IPV6);
313                 skb->free=1;
314                 skb->when=jiffies;
315                 skb->sk=sk;
316                 skb->arp=0;         /* How is it used with IPv6? TOHANDLE */
317                 COPY_ADDR6(*s, skb->saddr6);
318                 COPY_ADDR6(*r, skb->raddr6);
319 /*
320                  * retrieve mac addr.
321                  */
322                 if (hh)
323                    ipv6_build_mac_header(skb, hh, plen);
324
325 /*
326                  *  ip header, setup skbuff area
327                  */
328                 skb->ipv6_hdr=iph=(struct ipv6hdr *)skb_put(skb,plen);
```

ipv6_output.c

`334-366` If no IP header has been provided, `*iph` is filled in using the input parameters. In this case, `length` is the true payload of the IP packet. It is stored in network byte order. `getfrag()` can then get the data fragment and place it into the sk_buff, following the IPv6 header and options if any. Note that `getfrag` may vary depending on the function that called `ipv6_build_xmit()`, e.g. `raw6_output` or `icmpv6_reflect`.The sk_buff is now ready to be transmitted to lower layers with `dev_queue_xmit`.

ipv6_output.c

```
334                 if(!sk->ip_hdrincl)
335                 {
336                         iph->vrprflow  = IPV6_VERSION |
337                                          IPV6_SET_FLOWLABEL(info->flow_lbl) |
338                                          IPV6_SET_PRIORITY(info->priority);
339                         iph->payload   = htons(length);
340                         iph->next_hdr  = protocol;
341                         iph->hop_limit = IPV6_DEFAULT_HOP_LIMIT;
342                         COPY_ADDR6(*s, iph->saddr);
343                         COPY_ADDR6(*d, iph->daddr);
344                 }
345                 getfrag(frag, s,((char *)iph) + plen - length, 0, length);
346                 dev_unlock_list();
347
348                 if(dev->flags&IFF_UP)
349         /* interface is up: send... */
350                         dev_queue_xmit(skb, dev, sk->priority);
351                 else
352                 {
353                         ip6_statistics.IpOutDiscards++;
354                         kfree_skb(skb, FREE_WRITE);
```

```
                }
#ifdef IPV6_DEBUG
printk("[ipv6\_output] OUT");
#endif
                return 0;
        }
        else
  {
          printk("ipv6_output: invalid raw packet - 10-05-96");
          return -1;
  }
}
```

ipv6_output.c

1.8 ipv6_build_mac_header Function

1. **Description** This function retrieves a hard header, given an sk_buff and a neighbor structure.
2. **Input parameters**

Variable	Origin	Description
*skb	linux/skbuff.h	struct
n	net/nd.h	struct neighbor for the sk_buff being sent
len	int	fragment length

3. **Output parameters**

Variable	Origin	Description
skb	linux/sk_buff.h	network buffer in which the hard header was stored.

4. **Returns** the length of the hard header.
5. **Postconditions:**

80-116 The device structure
tt *dev to be considered is retrieved using the neighbor structure `*n` passed in argument. If a hard header length is indicated, a space of the same size [3]lus ??? is reserved in the skbuff. If available, `dev->hard_header` is used to retrieve the header. The hard header length is returned.

ipv6_output.c

```
80 static int __inline__ ipv6_build_mac_header(struct sk_buff *skb,
81                                             struct neighbor *n,
82                                             int len)
83 {
84         int mac;
85         int hdrlen = 0;
86         struct device *dev = n->dev;
87 
88         skb->arp = 1;
89 
90         /* ignore multicast for now ... */
91 
92         /*
93          *
94          */
95         if (dev->hard_header_len)
```

[3]p


```
96         {
97                 skb_reserve(skb, (dev->hard_header_len + 15) & ~15);
98                 if (dev->hard_header)
99                 {
100                        mac = dev->hard_header(skb, dev, ETH_P_IPV6,
101                                               NULL, NULL, len);
102                        if (mac < 0)
103                        {
104                                hdrlen = -mac;
105                                skb->arp = 0;
106                        }
107                        else
108                        {
109                                hdrlen = mac;
110                        }
111                }
112                else
113                        hdrlen = dev->hard_header_len;
114        }
115        return hdrlen;
116 }
```

ipv6_output.c

Chapter 2

The AF_INET6 Family

This section proposes detailled description of the implementation of the
PF_INET6 familly. It presents the code that handle the PF_INET6 familly (system calls and intialization).

2.1 Data structures for socket's implementation

Most of the new structures presented here are coming from the draft about the BSD API (draft-ietf-ipngwg-bsd-api-05).

2.1.1 File : include/asm-i386/types.h

Two small changes within this file to include the definitions of the types *u_int16_t* (defined as an unsigned short) and *u_int32_t* (defined as an unsigned int) respectively on lines 21 and 28. Those two definitions are used in some structures definition like **sockaddr_in6**.

2.1.2 File : include/linux/in.h

Let first consider the version number :

IPV6_VERSION : value 6

A number of new socket options are defined for `IPv6`. All of these new options are at the **IPPROTO_IPV6** level.

- IPPROTO_IPV6 : value 41
- IPPROTO_ICMPV6value 58

The constant name prefix **IPV6_** is used in all of the new socket options to identify explicitely these options as applying to `IPv6`.
A new data structure to hold a single `IPv6` address, **in6_addr** is defined as :

```
struct in6_addr {
        union {
                u_int32_t u6_addr32[4];
                u_int8_t  u6_addr8[16];
        } u6_addr;
#define s6_addr32       u6_addr.u6_addr32
#define s6_addr8        u6_addr.u6_addr8
#define s6_addr         u6_addr.u6_addr8
};
```


The #define allows us to use transparently the data structure in respect of the draft.
The `IPv6` address is stored in network byte order.
The **sockaddr_in** structure is the protocol-specific address data structure for IPv4. It is used to pass address between applications and the system in the socket functions. We've defined a new structure which can support the new format of `IPv6` address as **sockaddr_in6** in the 4.3 BSD system.
The `IPv6` header contain a 24-bit field to hold a "flow label", and a 4 bit field to hold a "priority" value. So implementations provide two bitmask constant declarations to help applications select out the flow label and priority fields. These constant are :

- **IPV6_FLOWINFO_FLOWLABEL** which select the label field of the header.
- **IPV6_FLOWINFO_PRIORITY** which select the priority field of the header.

On the sending side, applications are responsible for selecting the flow label value. The system provides constant declarations for the `IPv6` priority values. These constant are :

- IPV6_PRIORITY_UNCHARACTERISED : value 0
- IPV6_PRIORITY_FILLER : value 1
- IPV6_PRIORITY_UNATTENDED : value 2
- IPV6_PRIORITY_RESERVED1 : value 3
- IPV6_PRIORITY_BULK : value 4
- IPV6_PRIORITY_RESERVED2 : value 5
- IPV6_PRIORITY_INTERACTIVE : value 6
- IPV6_PRIORITY_CONTROL : value 7
- IPV6_PRIORITY_8 : value 8
- IPV6_PRIORITY_9 : value 9
- IPV6_PRIORITY_10 : value 10
- IPV6_PRIORITY_11 : value 11
- IPV6_PRIORITY_12 : value 12
- IPV6_PRIORITY_13 : value 13
- IPV6_PRIORITY_14 : value 14
- IPV6_PRIORITY_15 : value 15

There are some function definition to get and set the flow label and the priority field, and to get the version of `IPv6` within the `IPv6` header as following :

- IPV6_GET_FLOWLABEL(x) (ntohl(x) & 0x00ffffff)
- IPV6_GET_PRIORITY(x) ((ntohl(x) >> 24) & 0xf)
- IPV6_GET_VERSION(x) ((ntohl(x) >> 28) & 0xf)
- IPV6_SET_FLOWLABEL(x) (htonl(x) & IPV6_FLOWINFO_FLOWLABEL)
- IPV6_SET_PRIORITY(x) (htonl((x & 0xf) << 24))

The application has control over the loose/strict source routing flag by setting or clearing the loose/strict flag contained in the *sin6_flowinfo* field. The implementation provide a set of constant definitions to simplify getting and setting the loose/strict flag from the *sin6_flowinfo field* :

IPV6_FLOWINFO_SRFLAG : Value 1

In addition, two constants are provided which represent the two states of this flag :

- IPV6_SRFLAG_STRICT : value 1
- IPV6_SRFLAG_LOOSE : value 0

The argument type of some `setsockopt()` options is the **ipv6_mreq** structure at line 60.

In order to allow applications to easily declare buffers of the proper size to store IPv4 and `IPv6` addresses in string form, implementations should provide the following constants :

- INET6_ADDRSTRLEN : value 46
- INET_ADDRSTRLEN : value 16

While the `bind()` function allows applications to select the source IP adress of UDP packets and TCP connections, applications often wish to let the system select the source address for then. In IPv4, this is done by specifying the IPv4 address represented by the symbolic constant INADDR_ANY in the `bind()` call, or by simply by skipping the `bind()` entirely. But since the `IPv6` address type is a structure (struct *in_addr*), a symbolic constant can be used to initialize an IPv address variable. The extern definition of this variable is :

extern struct in6_addr ipv6addr_any

The other version is a symbolic constant :

IPV6ADDR_ANY_INIT {0, 0, 0, 0}

Applications may need to send UDP packets to, or originate TCP connection to, services residing on the local node by using the constant IPv4 address INADDR_LOOPBACK. `IPv6` also provides a loopback address which can be used to contact local TCP and UDP services. Like the unspecified address, the `IPv6` loopback address is provided in two forms -- a sybolic constant and a global variable :

- IPV6ADDR_LOOPBACK_INIT {0, 0, 0, 1}
- extern struct in6_addr ipv6addr_loopback

2.1.3 File : include/linux/socket.h

A new address family macro, named **AF_INET6**, has been defined to distinguish between the original ***sockaddr_in*** address data structure, and the new ***sockaddr_in6*** data structure. A new protocol family macro, named **PF_INET6**, has been defined . Like most other protocol family macros, this will usually been defined to have the same value as the corresponding address family macro :

- #define AF_INET6 10
- #define PF_INET6 AF_INET6

An `IPv6` application that is passed an open socket from an unknown process may use the **IPV6_ADDRFORM** `setsockopt()` option to "convert" the socket to **PF_INET6**.

#define IPV6_ADDRFORM 22

A new `setsockopt()` option is used to control the hop limit used in outgoing unicast `IPv6` packets. This option is :

#define IPV6_UNICAST_HOPS IP_TTL (value 2)

A few `setsockopt()` options at the IPPROTO_IPV6 layer are used to control some of the parameters of sending multicast packets. These options are optionnal : applications may send multicast packets without using these options :

- IPV6_MULTICAST_IF : set the interface to use for outgoing multicast packets (value : IP_MULTICAST_IF : 32)
- IPV6_MULTICAST_HOPS : set the limit to use for outgoing multicast packets (value : IP_MULTICAST_TTL : 33)
- IPV6_MULTICAST_LOOP : controls whether outgoing multicast packets sent should be delivered back to the local application (value : IP_MULTICAST_LOOP : 34)
- IPV6_ADD_MEMBERSHIP : join a multicast group (value : IP_ADD_MEMBERSHIP : 35)

- IPV6_DROP_MEMBERSHIP : leave a multicast group (value : IP_DROP_MEMBERSHIP : 36)

The **IPV6_RECVSRCT** option controls the reception of source routes. Implementations provide the **IPV6_SR_MAXADDR** constant declaration in order to allow applications to simply declare storage for the largest possible source route.

A new `setsockopt()` option named **IPV6_RECVIF** is provided at the IPPROTO_IPV6 level. If this option is enable, the system returns an additionnal *sockaddr_in6* structure to the application, holding the `IPv6` address of the receiving interface, in the `recvfrom()`, `recvmsg()`, or `accept()` functions.

Applications may also need to specify the outgoing interface that originated UDP or TCP packets should use. This is accomplished like source route selection. The **IPV6_SENDIF** option is needed so that the system can differentiate between the application's specification of an outgoing interface address and a source address.

2.1.4 File : include/linux/netdb6.h

Two new functions analogous to `gethostbyname()` an `gethostbyaddr()` have been defined which support both IPv4 and `IPv6` addresses.The names of the new functions are `hostname2addr()` and `addr2hostname()` :

- struct hostent *addr2hostname(const char *,int,int *)
- struct hostent *addr2hostname(const void *,int,int,int *)

`IPv6` addresses are returned by `hostname2addr()` and `addr2hostname()` via the *hostent* structure. This h_addr_list element of this structure points to an array of *ipv6_hostent_addr* type structure :

```
struct ipv6_hostent_addr {
    /* amount of time in seconds that address is valid for */
    u_int16_t iha_lifetime;

    /* the IPv6 address being returned */
    struct in6_addr iha_addr;
}
```

The hostent and address structures returned both of these functions is allocated by the library. Applications use the `freehostent` function to return this storage to the library after they are done using it :

void freehostent(struct hostent *);

2.1.5 File : include/linux/inet6.h

BSD UNIX provides two functions, `inet_addr()` and `inet_ntoa()`, to convert an IPv4 address between binary and printable form. `IPv6` applications need similar functions. The following two functions convert both `IPv6` and IPv4 addresses :

- int ascii2addr(int,const char *,void *);
- char *addr2ascii(int,const void *,int,char *);

The IPv4-mapped `IPv6` address format is used to represent IPv4 addresses as `IPv6` addresses. Most applications should be able to manipulate `IPv6` addresses as opaque 16-octet quantities, without needing to know whether they represent IPv4 addresses.A few applications may need to determine whether an `IPv6` address is an IPv4-mapped adress or not. The following function is provided for those applications :

int inet6_isipv4addr(const struct in6_addr *);

2.1.6 File : include/linux/ipv6.h

This file contains the declaration of the IPv6 packet.

```
    struct ipv6hdr {
            __u32 vrprflow;           /* ipv6 version, priority and flow label */
            __u16 payload;
            __u8  next_hdr;
            __u8  hop_limit;
            struct in6_addr saddr,    /* ipv6 source address */
                            daddr;    /* ipv6 destination address */
};
```

2.1.7 File : include/linux/skbuff.h

This file contains the definition of the structure sk_buff. That structure has been altered to handle the IPv6 adressing.

We add in the union h of skbuff :

```
struct ipv6hdr  *ipv6h;
```

and in the main structure

```
struct in6_addr saddr6; /* IPv6 source address */
struct in6_addr daddr6; /* IPv6 target address */
```

2.1.8 File : include/net/af_inet.h

Because we reuse most of the functions that handle the AF_INET familyto process the AF_INET6 family, we need to have an external declaration. That's this file contains.

2.2 AF_INET6

2.2.1 Code introduction

Here are the files directly concerned with the processing of the AF_INET6 family.

File	Description
include/linux/skbuff.h	struct sk_buff
include/net/protocol6.h	struct to manipulate the IPv6 upper-layer protocols
net/protocol.c	now handle IPv6 protocol
net/ipv6/af_inet6.c	main functions for AF_INET6 family
net/ipv6/protocol6.c	registration of IPv6 upper-layer protocols

2.2.2 Statistics

Today (96/05/22), format of mib isn't defined. So we use the old ip4 mib format. The new mib is *ip6_statistics* and is defined in *include/net/snmp.h*.

2.2.3 `struct sk_buff` Structure

It's not the purpose of this document to presents the sk_buff. We will just show the changes of the structure to handle the IPv6 protocol.

include/linux/skbuff.h


```
struct sk_buff
{
...
union
{
...
#ifdef CONFIG_NET_IPV6_DIM
struct ipv6hdr  *ipv6h;
#endif
...
} h;

...
#ifdef CONFIG_NET_IPV6_DIM
struct in6_addr saddr6; /* IPv6 source address */
struct in6_addr daddr6; /* IPv6 target address */
#endif
...
};
```

include/linux/skbuf.h

2.2.4 `sk_inuse6()` Function

1. **Description** This function check if a socket exists at the given port. It is used by *get_new_socknum6* to find an unused port.
2. **Input parameters**

Variable	Origin	Description
struct proto *	include/net/sock.h	the socket's protocol
int		the port number

3. **Returns** this function return 0 if the port is not used, else 1.
4. **Preconditions** None.
5. **Postconditions** None.

2.2.5 `get_new_socknum6()` Function

1. **Description** This function return a port that isn't used. The port is placed at the best place: at an empty place in the sockarray, or at the smallest distance of the sockarray (you should remember that sockarray is an array of pointers to linked lists).
2. **Input parameters**

Variable	Origin	Description
struct proto *	include/net/sock.h	the socket's protocol
unsigned short base		minimum port number required

3. **Returns** this function returns the unused port number, always equal or superior to *base*.
4. **Preconditions** None.
5. **Postconditions** None.

2.2.6 get_sock6() Function

1. **Description** This function search for a socket into the protocol's sockarray, given its port number.
2. **Input parameters**

Variable	Origin	Description
struct proto *prot	include/linux/sock.h	point to the sockarray
unsigned short num		the port corresponding to the socket
struct in6_addr *raddr	include/linux/in.h	the remote adress
unsigned short rnum		the remote port in host order
struct in6_addr *laddr		the local port

3. **Returns** this function return a pointer to the found socket, NULL else.
4. **Preconditions**
   - The port number must be given in network order

2.2.7 put_sock6() Function

1. **Description** This function add a socket into the protocol's sockarray, given its port number.
2. **Input parameters**

Variable	Origin	Description
unsigned short		a free port
struct sock *sk	include/linux/socket.h	the socket to add

3. **Returns** this function has no return.
4. **Preconditions**
   - The socket must exists.
   - The port number must be free (eg no socket there).
   - The port number is given in host order
5. **Postconditions** The socket is a the linked list head at its position in the sockarray.

2.2.8 remove_sock6() Function

1. **Description** This function removes a socket from a sockarray.
2. **Input parameters**

Variable	Origin	Description
struct sock *	include/linux/socket.h	the socket to remove

3. **Returns** this function has no return.
4. **Preconditions** describes which conditions must be fulfilled before this function is called.
   - The socket must exists.
5. **Postconditions**
   - The socket is no longer in the sockarray, but still exists.

2.2.9 destroy_sock6() Function

1. **Description** This function cleans and destroyes a socket.
2. **Input parameters**

Variable	Origin	Description
struct sock *	include/linux/socket.h	the socket to destroy

3. **Returns** this function has no return.
4. **Preconditions**
   - The socket must exists and be in a sockarray
5. **Postconditions**
   - The socket and packets in its queues no longer exist.

2.2.10 inet6_autobind() Function

1. **Description** This function gives a port number to an socket, if it has not one yet (when port number is 0).
2. **Input parameters**

Variable	Origin	Description
struct sock *	include/linux/socket.h	the socket

3. **Returns** this variable 0 on succes, -EAGAIN if it can give a port number (for instance, no more room).
4. **Preconditions** None.
5. **Postconditions**
   - If return is 0, the socket has a port number.

2.2.11 inet6_listen() Function

1. **Description** This function puts a socket in a listening state, if not yet. If the socket has no local port (ie no bind has been done), ***autobind()*** is called.
2. **Input parameters**

Variable	Origin	Description
struct socket *	include/linux/socket.h	the socket familly

   The pointer to the socket is stored in the data field of the socket familly parameter.
3. **Returns** this variable return 0 on succes, -EAGAIN on error (if it can allocate a port number to the socket).
4. **Preconditions** None.
5. **Postconditions**
   - The socket has a port number and is a listening state.

2.2.12 inet6_create() Function

1. **Description** This function creates a new socket.
2. **Input parameters**

Variable	Origin	Description
struct socket *	include/linux/socket.h	
int		protocol

3. **Returns** this function returns 0 on success, or a standard error code.
4. **Preconditions** None.
5. **Postconditions** None.

2.2.13 inet6_bind() Function

1. **Description** This function binds a socket to an adress.
2. **Input parameters**

Variable	Origin	Description
struct socket *	include/linux/socket.h	the socket familly protocol
struct sockaddr	include/linux/socket.h	the adress to bind to
int		length of previous structure

3. **Returns** this variable returns 0 on succes, standard error codes on error.
4. **Preconditions** None.
5. **Postconditions**
   - The socket has now a valid inet6 adress.

2.2.14 connect6() Function

1. **Description** This function initiates a connection on a socket.
2. **Input parameters**

Variable	Origin	Description
struct socket *	include/linux/socket.h	the socket familly
struct sockaddr	include/linux/socket.h	overlaid struct sockaddr_in6
int		length of previous structure
int		user given flags

3. **Returns** this variable returns 0 on succes, standard error codes on error.
4. **Preconditions** None.
5. **Postconditions** The socket is a connected state.

2.3 Upperlayer AF_INET6 protocols handling

Each PF_INET6 upper-layer protocol is stored in a hash table, which contains their names and some entry points.

At the beggining, protocols are staticly stored in a linked list, *inet6_protocol_base*. Then, when the kernel boots, *inet6_proto_init()* (in `net/ipv6/af_inet6.c`) is called. That function reads this list and stores the protocol entry in a hash table,*inet6_protos[]*.

2.3.1 Code introduction

The following files are used :

File	Description
net/ipv6/protocol6.c	the code to handle the array
include/net/protocol6.h	the prototypes and protocol structure

2.3.2 Global variables

We present the global variables as follows:

Variable	Datatype	Origin	Description
inet6_protocol_base	struct inet6_protocol *	include/net/protocol6.h	beginning of the list
inet6_proto[]	struct inet6_protocol *	include/net/protocol6.h	the hash tabel

2.3.3 inet6_protocol Structure

include/net/protocol6.h

```
struct inet6_protocol {
int (*handler)(struct sk_buff *skb, struct device *dev,
       struct options6 *opt_ipv6,
       struct in6_addr daddr,
       unsigned short len,
       struct in6_addr saddr,
       int redo,
       struct inet6_protocol *protocol);

void (*err_handler)(int type,
    int code,
    unsigned char *buff,
    struct in6_addr daddr,
    struct in6_addr saddr,
    struct inet6_protocol *protocol);

void (*proto_init)(void);

struct inet6_protocol *next;
unsigned char protocol;
unsigned char copy:1;
void *data;
const char *name;
};
```

include/net/protocol6.h

- *handler()* is the receive function of the protocol (eg udp6_rcv()),
- *err_handler()* is the error function (eg udp_err()),
- *proto_init()* is a new entry and allow to make a more generic code. That function initialize (register in /proc, clear the sockarray, ...) a protocol (eg raw_init()),
- *next* points on the next protocol,
- *protocol* is the protocol number (eg IPPROTO_TCP),

- *copy* if equal to 1, means that there is another protocol in the list with the same protocol number,
- *data* Not used (yet ?) in our implementation,
- *name* is the name of the protocol (the one you see at the kernel starting).

2.3.4 inet6_get_protocol() Function

1. **Description** This function finds a protocol in the array given its protocol number.

   Function not used.

2.3.5 inet6_add_protocol() Function

1. **Description** This function adds a protocol handler to the hash table.
2. **Input parameters**

Variable	Origin	Description
struct inet6_protocol *	include/net/protocol6.h	the handler to add

3. **Returns** this function has no return.
4. **Preconditions** None.
5. **Postconditions** None.

2.3.6 inet6_del_protocol() Function

1. **Description** This function removes a protocol from the hash table.
2. **Input parameters**

Variable	Origin	Description
struct inet6_protocol *	include/net/protocol6.h	the handler to remove

3. **Returns** this function returns -1 on error (eg: protocol not present in hash table), else 0.
4. **Preconditions** None.
5. **Postconditions** None.

2.4 Raw

Unlike the ipv4 raw protocol, ipv6 raw protocol is now considered as an PF_INET6 protocol (stored in the inet6_protool table). It is different as others protocols because it doesn't have a receive handler. A raw socket can send and receive a packet with a protocol number from 1 to 254. A new (for linux, but not BSD) features is that a raw socket listening the protocol 0 will receive every packet that correctly arrive at the IPv6 layer.

2.4.1 Code introduction

The code of the raw6 protocol is contained in only one file, whatever it uses some other files.

File	Description
net/ipv6/raw6.c	the main code for raw6
include/net/raw6.h	raw6 declarations
include/net/sock.h	structures sock and protocol

2.4.2 Deliver of raw packets

When ipv6_rcv() receive a packet, it calls raw6_deliver(). That function will check if one or more sockets asked for that packet, and deliver it to them by calling raw6_rcv(), except for the first socket found. This socket will be returned to ipv6_rcv(), which will call raw6_rcv() later, to avoid a copy.

2.4.3 raw6_deliver() Function

1. **Description** This function delivers IP packets to good raw sockets.
2. **Input parameters**

Variable	Origin	Description
struct sk_buff *skb	include/net/skbuff.h	the incoming packet
int nexthdr		the protocol number
struct in6_addr *saddr	include/linux/in.h	the source adress
struct in6_addr *daddr	include/linux/in.h	the dest adress

3. **Returns** The function returns a struct sock *, representing the first socket found. The packet has not yet been sent to that socket. ipv6_rcv() will do it later to avoid a copy.
4. **Preconditions**: None.
5. **Postconditions**: The packet is delivered to all matching sockets, except the first one.

2.4.4 raw6_rcv() Function

1. **Description** This function charges a packet to a socket.
2. **Input parameters**

Variable	Origin	Description
struct sock *sk	include/net/sock.h	the socket to charge
struct sk_buff *skb	include/linux/skbuff.h	the packet

3. **Returns** this function returns an int equal to 0, meant all is OK.
4. **Preconditions**: None.
5. **Postconditions**: The packet is delivered or destroyed. The skb is no longer valid.

2.4.5 raw6_sendmsg() Function

1. **Description** This function send a message IPv6 layer. It also use two callback functions (see raw6_getfrag() and raw6_getfragcsum()).
2. **Input parameters**

Variable	Origin	Description
struct sock *sk	net/sock.h	the socket
struct msghdr *msg	linux/socket.h	the message to send
int len		message lenght
int noblock		block/non-block
int flags		user given flags

3. **Returns** this variable returns an int which represents the error code or the size sent.
4. **Preconditions** None.

5. **Postconditions** None.

2.4.6 raw6_close() Function

1. **Description** This function closes a socket.

2.4.7 raw6_recvmsg() Function

1. **Description** This function returns an incoming packet to a listening raw socket.
2. **Input parameters**

Variable	Origin	Description
struct sock *sk	linux/socket.h	the socket
struct msghdr *msg	linux/socket.h	the buffer where to copy
int len		length max to read
int noblock		block/non-block
int flags		user given flags
int *addr_len		0 or size of sockaddr_in6

3. **Returns** this variable returns an int, which is the error code or the length read.
4. **Preconditions** None.
5. **Postconditions** None.

2.4.8 raw6_setsockopt() Function

1. **Description** This function sets raw6 specific options.
2. **Input parameters**

Variable	Origin	Description
struct sock *sk	include/net/sock.h	the socket on which options apply
int level		the option level
int optname		the option name
char *optval		the option value
int optlen		the option lenght

3. **Returns** this function returns 0 on succes, or a standard error code.
4. **Preconditions** none.
5. **Postconditions** none.

   To access to this function, level must be IPPROTO_RAW. Currently, only one option is exists : *RAWIP6_ADDCHECKSUM*. That options says if a checksum must be computed, and if so, where it should be in a packet, starting from the end of the first ipv6 header.

   The **optval* value is put in the raw6 option variable, *ck_offset*. If **optval* is negative, no checksum will be computed, and raw6 option variable *checksum* is set to 0, else it is set to 1.

   Please note that it is not standardised yet and might change.

2.4.9 raw6_getsockopt() Function

1. **Description** This function read the raw6 current options.

2. **Input parameters**

Variable	Origin	Description
struct sock *sk	include/net/sock.h	the socket we want to read options
int level		the option level
int optname		the option name
char *optval		function will put the value here
int *optlen		function will put the value length here

3. **Returns** this function return 0 on succes, else a standard error code.

   Set *raw6_setsockopt()* for details.

2.4.10 raw6_init_sk() Function

1. **Description** This function is called to initialise a newly created raw socket. It fixes the socket options (see *setsockopt()*) to default values: the checksum must be computed and be placed at the offset 2. It is the default values for ping. If the protocol is *IPPROTO_RAW*, the socket options *ip_hdrincl* (means "header included") is set to 1. Else this options stays 0.
2. **Input parameters**

Variable	Origin	Description
struct sock *sk	include/net/sock.h	the socket to initialise

3. **Returns** this function always returns 0, meaning all is good.
4. **Preconditions** the socket has a *SOCK_RAW* type (always).
5. **Postconditions** the socket exists and is initialised.

2.4.11 raw6_init_sk() Function

1. **Description** This function initialise the raw protocol. It is called only once, when the kernel boots. It fill the sockarray with NULL (no socket open yet) and register itself in */proc/net*.

   It takes no parameters and has no return.

2.4.12 raw6_getfrag() Function

1. **Description** This function is a simple callback, called be *ipv6_build_xmit()*. It copies the required fragment stored in one ore more iovec to the buffer given (in fact it is the data zone of a skbuff).
2. **Input parameters**

Variable	Origin	Description
const void *p		the source iovec
struct in6_addr *saddr		unused
char *to		the destination buffer
unsigned int offset		the offset of the required fragment
unsigned int fraglen		the length of the required fragment

3. **Returns** this function has no returns.
4. **Preconditions** offset and fraglen must be correct.
5. **Postconditions** none.

2.4.13 raw6_getfragcsum() Function

1. **Description** This function acts as the previous, but also computes a partial checksum. The function place it at the good place when it is completly computed. It can be done because fragments are built and sent starting from the last fragment. So, the last fragment required is in fact the first in the packet.
2. **Input parameters**

Variable	Origin	Description
const void *p		the source iovec
struct in6_addr *saddr		unused
char *to		the destination buffer
unsigned int offset		the offset of the required fragment
unsigned int fraglen		the length of the required fragment

3. **Returns** this function has no returns.
4. **Preconditions** offset and fraglen must be correct.
5. **Postconditions** none.

2.5 UDP

Chapter 3

ICMPv6

3.1 Basics

ICMPv6 is used by IPv6 nodes for different maintenance services such as error reporting, diagnostic (e.g. Ping) or multicast membership reporting (previously IGMP in IPv4). ICMPv6 messages belong to either of the two following categories: error messages with type identifier comprised between 0 and 127 or informational messages with type identifier comprised between 128 and 255. Fig. ??CMPv6hdr shows the general format common to all ICMPv6 messages.

3.1.1 Code introduction

File	Description
linux/icmpv6.h	Generic ICMPv6 declarations
net/icmpv6.h	ICMPv6 specific declarations
ipv6/icmpv6	ICMPv6 functionalities

3.1.2 Global variables

Variable	Datatype	Origin	Description
icmp6_statistics			snmp structures where ICMPv6 statistics are stored

3.1.3 Statistics

Name	Description
IpInReceives	# of IPv6 packets received
Fill in	Fill in

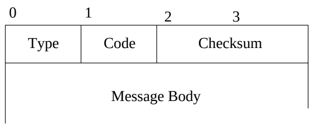

Figure 3.1: ICMPv6 general header

3.1.4 ICMP message types

Name	Description
IpInReceives	# of IPv6 packets received
ICMP6_UNREACH	Destination unreachable.
ICMP6_PKTTOOBIG	Packet too big error.
ICMP6_TIMXCEED	Time exceeded error
ICMP6_PARAMPROB	Parameter problem
ICMP6_SOLICITATION_RT	Router sollicitation.
ICMP6_ADVERTISMENT_RT	Router advertisement.
ICMP6_SOLICITATION_ND	Host sollicitation.
ICMP6_ADVERTISMENT_ND	Host advertisement.
ICMP6_REDIRECT	Redirect message.
ICMP6_GROUPMEM_QUERY	Group membership query.
ICMP6_GROUPMEM_REPORT	Group membership report.
ICMP6_GROUPMEM_TERM	Group membership termination.
ICMP6_ECHO	Echo message.
ICMP6_ECHOREPLY	Reply message.

3.1.5 icmpv6 Structure

The ICMPv6 header format varies with the type of ICMP message. The `icmpv6` structure is a generic representation of all possible ICMPv6 headers.

3-??? The first word of the header contains the type of the message, its code, and its checksum. The ICMPv6 checksum is 16-bit long and is computed from the total ICMP message, including its header with checksum field set to 0, plus a pseudo-header. The pseudo-header contains the IPv6 destination and source addresses for the message, its payload length, a 16-bit zeroed field and the next header type, IPPROTO_ICMPV6 = 58. This is the part of the header common to all ICMPv6 messages.
The second word of the header is message dependent. The union `icmp6_2w` provides a way to accomodate all formats. `pmtu` is used in messages for path MTU discovery, while `pptr` is used in bad parameter error messages. If the second word is meaningless or contains flags, `flags` is used. Then, we can distinguish between echo-reply, group membership and neighbor discovery messages.
[1]

linux/icmpv6.h

```
 3  /*
 4   * Structure of an ICMPv6 header.
 5   */
 6  struct icmpv6 {
 7          /* header - first 32bit word */
 8          __u8        icmp6_type;     /* type of message, see below */
 9          __u8        icmp6_code;     /* type sub code */
10          __16        icmp6_cksum;    /* checksum */
11          /* header - second 32bit word - 6 possibilities */
12          union {
13              __u32       pmtu;           /* next MTU (for discovery) */
14              __u32       pptr;           /* bad parameter pointer */
15              __u32       flags;          /* reserved/flags field */
16              struct id_seq {             /* echo */
17                  __u16   id;
18                  __u16   seq;
19              } idseq;
20              struct grp_msh {        /* group membership */
21                  __u16   mxdel;      /* maximum response delay 4.3 */
```

[1] Detail this later


```
22                  __u16   unused;
23              } grp_msh;
24              struct nd_msg {         /* neighbor discovery icmp header */
25                  __u8  cur_hlim;
26                  __u8  flags;
27                  __u16 rt_lftime;
28              } nd_msg;
29          } icmp6_2w;
30  #define icmp6_pmtu      icmp6_2w.pmtu
31  #define icmp6_pptr      icmp6_2w.pptr
32  #define icmp6_flags     icmp6_2w.flags
33  #define icmp6_id        icmp6_2w.id_seq.id
34  #define icmp6_seq       icmp6_2w.id_seq.seq
35  #define icmp6_mrd       icmp6_2w.grp_msh.mxdel
36  #define icmp6_unu       icmp6_2w.grp_msh.unused
37  #define icmp6_cur_hlim  icmp6_2w.nd_msg.cur_hlim
38  #define icmp6_ndflag    icmp6_2w.nd_msg.flags
39  #define icmp6_life      icmp6_2w.nd_msg.rt_ltime
```

linux/icmpv6.h

41- The next union, `icmp6_dun` handles all possible data information included in the header

```
41          union {
42              struct in6_addr id6_grp;        /* group management */
43  #ifndef ND6_DRAFT2
44              struct id6_addr {
45                  struct in6_addr  id6_tgt;     /* target address */
46                  struct in6_addr  id6_rdst;    /* (redirect) destination
*/
47              } id6_addr;
48  #else
49              struct id6_raddr {
50                  struct in6_addr  id6_rtgt;     /* target address */
51                  struct in6_addr  id6_rdst;    /* (redirect) destination
*/
52              } id6_raddr;
53              struct id6_saddr {
54                  struct in6_addr  id6_ssnd;     /* sender address */
55                  struct in6_addr  id6_stgt;     /* target address */
56              } id6_saddr;
57  #endif
58              struct id6_nud {                 /* router adv. msg. */
59                  __u32       id6_reach;   /* reachable time */
60                  __u32       id6_retrans; /* retrans timer */
61              } id6_nud;
62              struct ipv6hdr   id6_ip;         /* header of packet in erro
r */
63              char             id6_data[1];    /* echo packet data */
64          } icmp6_dun;
65  #define icmp6_grp       icmp6_dun.id6_grp
66  #ifndef ND6_DRAFT2
67  #define icmp6_tgt       icmp6_dun.id6_addr.id6_tgt
68  #define icmp6_rdst      icmp6_dun.id6_addr.id6_rdst
69  #else
70  #define icmp6_tgt       icmp6_dun.id6_grp
71  #define icmp6_rtgt      icmp6_dun.id6_raddr.id6_rtgt
72  #define icmp6_rdst      icmp6_dun.id6_raddr.id6_rdst
73  #define icmp6_ssnd      icmp6_dun.id6_saddr.id6_ssnd
74  #define icmp6_stgt      icmp6_dun.id6_saddr.id6_stgt
```

```
75 #endif
76 #define icmp6_reach     icmp6_dun.id6_nud.id6_reach
77 #define icmp6_retrans   icmp6_dun.id6_nud.id6_retrans
78 #define icmp6_ip        icmp6_dun.id6_ip
79 #define icmp6_data      icmp6_dun.id6_data
80 };
```

linux/icmpv6.h

3.2 Functions

3.2.1 icmpv6_rcv Function

1. **Description** This function processes incoming ICMPv6 messages. It is called from ipv6_rcv when the packet is labeled with prototype IPPROTO_ICMPV6. It takes action according to the ICMPv6 type of the message.

2. **Input parameters**

Variable	Origin	Description
*skb	linux/sk_buff.h	sk_buff containing incoming packet with IPv6 header
dev	linux/netdevice.h	main device on which the packet was received.
opt		
d	linux/ipv6.h	destination address
len	int	packet length (icmpv6 header + data)
s	linux/ipv6.h	source address
redo	int	???????
protocol	net/protocol6.h	inet6_protocol structure

3. **Returns** 0.

226-269 The first action on the incoming packet is to check that it is a valid ICMPv6 message. If the data length is less than the minimum ICMPv6 data length the packet is automatically discarded. Otherwise, its checksum is computed. `csum_partial` is used to get the ckecksum for the ICMPv6 header and the data. `csum_ipv6_magic` complete the computation by considering the ICMPv6 pseudo-header described in the introduction of this chapter. Actual checksum must be equal to 0 to consider the packet as valid.

linux/icmpv6.c

```
226 int icmpv6_rcv(struct sk_buff *skb,
227                struct device *dev,
228                struct options *opt,
229                struct in6_addr *d,
230                unsigned short len,
231                struct in6_addr *s,
232                int redo,
233                struct inet6_protocol *protocol)
234 {
235    register struct icmpv6 *hdr= (struct icmpv6 *)skb->h.raw;
236    icmp6_statistics.IcmpInMsg++;
237    int icmp_len;
238
239 #ifdef IPV6_DEBUG
240         printk("[ICMPV6]icmpv6_rcv  IN\n");
241 #endif
242    /*
243     * Validate packet (length)
244     */
245    if ((icmp_len= skb->tail - (unsigned char *) (hdr + 1)) < ICMP6_MINLEN )
```



```
246    {
247       icmpv6_statistics.IcmpInErrors++;
248       printk("[ICMPv6]icmpv6_rcv: bad length \n");
249       kfree_skb(skb, FREE_READ);
250       return(0);
251    }
252
253    /*
254     * Validate packet
255     */
256    switch(skb->ip_summed) {
257       case CHECKSUM_NONE:                    /* no checksum computed so far */
258          skb->csum = csum_partial((char *)hdr, len, 0);
259       case CHECKSUM_HW:      /* pseudo header not part of checksum so far */
260          if (csum_ipv6_magic(s, d, len, IPPROTO_ICMPV6, skb->csum))
261          {  /* checksum <> 0: error */
262             icmpv6_statistics.IcmpInErrors++;
263             printk("[ICMPv6]icmpv6_rcv: checksum failed. \n");
264             kfree_skb(skb, FREE_READ);
265             return(0);
266          }
267       default:
268            /* CHECKSUM_UNNECESSARY should not be reached?*/
269    }
270
271    switch (hdr->icmp6_type) {
 ----------------------------------------------------------------
 |                                                              |
 |                                                              |
 |            switch for all message types                      |
 |                                                              |
 ----------------------------------------------------------------
```

linux/icmpv6.c

562-568 When an ECHO message is received, the corresponding statistic count is incremented and the packet reflected to the sender with `icmpv6_reflect`.

linux/icmpv6.c

```
562         case ICMP6_ECHO:
563                 icmp6_statistics.IcmpInEchos++;
564                 hdr->icmp6_type = ICMP6_ECHOREPLY;
565                 /* reflect the packet (reply) */
566                 icmp6_statistics.IcmpOutEchoReps++;
567                 icmpv6_reflect(skb, dev);/* TOHANDLE*/
568                 return(0);
```

linux/icmpv6.c

574-577 Reception of a REPLY message only causes the kernel to increment the corresponding statistic variable, as a user RAW socket should also receive the packet and service it.

linux/icmpv6.c

```
574         case ICMP6_ECHOREPLY:
575                 icmp6_statistics.IcmpInEchoReps++;
576                 break;
577
```

linux/icmpv6.c

3.2.2 icmpv6_reflect() Function

1. **Description** This function reflects echo messages to the sender. It is called by `icmpv6_rcv` upon reception of such messages.

2. **Input parameters**

Variable	Origin	Description
*skb	linux/skbuff.h	Network buffer containing the echo message
dev	linux/netdevice.h	device on which the packet was received

3. **Output parameters** Same as input parameters, updating according to the source and destination addresses of the reply.

4. **Returns** Void.

5. **Preconditions**
   - skb->ipv6hdr points to the beginning of the incoming packet IPv6 header.
   - skb->h.raw points to the beginning of the ICMPv6 header.

6. **Postconditions**
   - skb->ipv6hdr and skb->h are in the same configuration.
   - The IPv6 header source and destination addresses are updated.
   - *dev corresponds to the new source address.
   - The ICMPv6 packet data length may have been reduced to fit the return path MTU.
   - A new checksum has been computed.
   - The destination address must be a unicast one.

91-107 The IPv6 and ICMPv6 headers are first retrieved from the skbuff passed in parameters. The socket used for this connection is icmpv6_socket.

icmpv6.c

```
 91 static void icmpv6_reflect(struct sk_buff *skb, struct device *dev)
 92 {
 93 struct sock     *sk  = (struct sock *) icmpv6_socket.data;
 94 struct ipv6hdr *iphdr = skb->ipv6hdr;
 95 struct icmpv6  *icmphdr = (struct icmpv6hdr *) skb->h.raw;
 96 struct destination *dest, **rt_cache = NULL; /*TOHANDLE*/
 97 struct in6_addr *s, t;
 98 register struct inet6_dev *idev;
 99 register struct inet6_ifaddr *ia;
100
101 int len;
102 int hdrlen;
103
104 #ifdef IPV6_DEBUG
105         printk("[ICMPV6]icmpv6_reflect  IN\n");
106 #endif
107
```

icmpv6.c

108-152 First, we check that the current source address is unicast. If not the packet cannot be reflected. If it is unicast, it replaces the current destination address which is saved in a temporary variable `*t`. Finding the new source address involves more steps. First, we look for `*t` in the IPv6 unicast address list associated with the device passed in parameters. If it is there, we are done. If not, we must find out a source address with a scope similar to the destination address'. This is done through `ipv6_get_saddr`. [2][3] If no such address exists, no reply is issued and the function returns. Otherwise, the new source address is set in the header and the hop limit is initialized to its maximum value, `MAXTTL`.

icmpv6.c

```
108 /*
109  * Return address
110  */
111 if (IS_ANYADDR6(iphdr->saddr) || IS_MULTIADDR6(iphdr->saddr))
112 {
113    printk("[ICMPv6]icmpv6_reflect: bad return address \n");
114    kfree_skb(skb, FREE_READ);
115    return(0);
116 }
117 COPY_ADDR6(iphdr->daddr, t);
118 COPY_ADDR6(iphdr->saddr, iphdr->daddr);
119 /*
120  * Options
121  */
122
123 /*
124  * Source address
125  */
126
127 /* Find inet6_dev struct */
128 /* We may add a pointer in the device structure... */
129 for (idev = inet6_dev_lst; idev; idev = idev->next)
130 {
131    if (idev->dev == dev)
132       break;
133 }
134 /* Find inet6_ifaddr struct */
135 for (ia = idev->addr_list; ia; ia = ia->if_next)
136 {
137    if (SAME_ADDR6(t, ia->addr))
138       break;
139 }
140
141 if (ia == NULL)
142    ia = ipv6_get_saddr(dev, &t);
143 if (ia == NULL)
144 {
145    printk("[ICMPV6]icmpv6_reflect: no source address...\n");
146 #ifdef IPV6_DEBUG
147         printk("[ICMPV6]icmpv6_reflect  OUT\n");
148 #endif
149    return;
150 }
151 COPY_ADDR6(ia->addr, iphdr->saddr);
152 iphdr->hop_limit = MAXTTL;
```

icmpv6.c

[2]A routing table lookup may be better later...
[3]Shouldn't we try to find the address on *dev first? (RFC1885, 2.2)

159-181 The next step is to retrieve the route MTU between the source and destination we just found. If it is shorter than the data length plus header lengths, the data length is set to the PMTU minus the length of the ICMPv6, and IPv6 headers. If this calculation returns a negative result, the packet is not reflected.

icmpv6.c

```
159 dest = ipv6_dst_route(&iphdr->daddr, 0);
160     if (dest == NULL)
161     {
162        printk("[ICMPV6]icmpv6_reflect: no route to host\n");
163        return;
164     }
165     len = skb->tail - skb->h.raw - sizeof(struct icmpv6hdr);
166     hdrlen = sizeof(struct ipv6hdr) + sizeof(struct icmpv6hdr);
167                                  /* + optlen */
168     /*
169      * Compare payload and MTU.
170      * Reduce ICMP content if necessary.
171      * Discard if too small.
172      */
173     if (len + hdrslen > dest->pmtu)
174        len = dest->pmtu - hdrlen;
175     if (len < 0)
176     {
177 #ifdef IPV6_DEBUG
178         printk("[ICMPV6]icmpv6_reflect: MTU too small  OUT\n");
179 #endif
180        return;
181     }
```

icmpv6.c

157-159 Finally, we compute the checksum for the reflected packet. The ICMPv6 header checksum field is temporarily set to 0, and as in `icmpv6_rcv`, we use `csum_partial` on the whole message and `csum_ipv6_magic` on the pseudo-header to get the actual checksum. The sk_buff now contains a correctly formatted reply packet that is sent to `ipv6_build_xmit`. There, a new sk_buff will be allocated to accomodate the hard header length, and useful data will be copied to it. [4]

icmpv6.c

```
185  icmphdr->icmp6_cksum = 0;
   186  len = iphdr->payload;
   187  /* Take care of options later */
   188  skb->csum = csum_partial((char *) icmphdr, len, 0);
   189  icmphdr->icmp6_cksum = csum_ipv6_magic
   190       (&iphdr->saddr, &iphdr->daddr, htons(len), IPPROTO_ICMPV6, skb->csum);
   191  /*
   192   * We now have a skbuff containing both updated icmpv6 and ipv6 header
   193   * The question is: what about hh and available size in sk_buff?
   194   * This is why we need to allocate a new skbuff and go through
   195   * ipv6_build_xmit... Sigh...
   196   */
   197  ipv6_build_xmit(sk, get_icmpv6_frag, skb, len, &iphdr->daddr,
   198                     &iphdr->saddr, NULL/*offset*/, IPPROTO_ICMPV6, 1);
```

icmpv6.c

[4] May be safer and faster to compute the checksum in icmpv6_get_frag and perform the copy at the same time.

Chapter 4

Network programs

4.1 Ifconfig

This section shows the modifications done to support new family in IOCTL's systeml calls, and what has been done to support it in the standard ifconfig.

The ifconfig is mostly used to set/unset an adress to an interface. There is three problems. First, the size of a *struct sockaddr_in6* is greater than a *struct sockaddr*, used for example in the *struct ifreq*. Second, an interface can have more than one adress (usually one link-local and one gloabal). First, no IOCTL code exist to delete an adress. We have to solve all these problems. We separate in that document modifications done in kernel and those made in the ifconfig.

4.1.1 Kernel files

Kernel files modified were

File	Description
net/core/dev.c	handle of device ioctls
include/linux/if.h	structures used to pass ioctl'parameters

4.1.2 Kernel structures

This is the structure usually used to pass ioctl'parameters. An addition has been made in the second union to handle ipv6 adresses. There is no *struct in6_addr* in this structure because that would have increased the ifreq structure, and thus we have lost binary compatibility.

if.h

```
struct ifreq
{

        union
        {
                char    ifrn_name[IFNAMSIZ];      /* if name, e.g. ``en0'' */
        } ifr_ifrn;

        union {
                struct  sockaddr        ifru_addr;
                struct  sockaddr        ifru_dstaddr;
                struct  sockaddr        ifru_broadaddr;
                struct  sockaddr        ifru_netmask;
                struct  sockaddr        ifru_hwaddr;
                struct {
                        unsigned short sa_family;
```



```
                        int len;                /* length of buffer */
                        void *buff;             /* Store infos */
                } inet6;
                short   ifru_flags;
                int     ifru_metric;
                int     ifru_mtu;
                struct  ifmap ifru_map;
                char    ifru_slave[IFNAMSIZ];   /* Just fits the size */
                caddr_t ifru_data;
        } ifr_ifru;
};

#define ifr_len6        ifr_ifru.inet6.len      /* length of buffer     */
#define ifr_buf6        ifr_ifru.inet6.buff     /* store infos          */
```

if.h

Now *ifr_buf6* is used to pass arguments. It will be an array of the following structure:

if.h

```
struct ifreq6
{
        int preflen;            /* prefix length of the above adress */
        struct in6_addr in6;
};
```

if.h

4.1.3 SIOCGIFADDR_1 ioctl call

1. **Description** This call is used to get all the inet6 adresses of an interface.
2. **Usage** First, declare an array of *struct ifreq6* to store adresses. It should be long enough, because no error are returned if it is too small. It's up to you to test if it is full on return. *ifr_buf6* must point to that array. *ifr_len6* is the size in bytes of that array. Don't forget to set the family to *AF_INET6*. On return, the array is fit, and *ifr_len6* contains the size fit in bytes.

example.c

```
        struct ifreq6 req[60];            /* enough room for 60 adresses */
  /* about 1024 bytes */
        struct ifreq ifr;
int numaddr6;                   /* number of adresses found */

        strcpy(ifr.ifr_name, ifname);
        ifr.ifr_addr.sa_family = AF_INET6;
        ifr.ifr_len6 = sizeof(req6);
        ifr.ifr_buf6 = (caddr_t) &req6;
        if (ioctl(skfd, SIOCGIFADDR, &ifr) < 0)
          { error }

/* ioctl is ok */
        numaddr6 = ifr.ifr_len6 / sizeof(struct ifreq6);
        if (numaddr6) {
Store the adresses found somewhere. They are stored in
the req6 array.
        }
```

example.c

4.1.4 SIOCSIFADDR and SIOCDELADDR ioctl calls

1. **Description** These codes respectively set and remove an adress.
2. **Usage** These function works identically. Of course, you can't set an existing adress, and can't remove an inexistant one. Principle is nearly the same that the SIOCGIFADDR, except that you can set or remove only one adresse at the same time. As SIOCDELADDR is a new code, it is implemented ONLY for *PF_INET6* protocol family. So the socket used must have been opened in that family.

example.c

```
        struct in6_addr in6;
        struct ifreq6 req6;
        struct ifreq ifr;

        { ...  fill in6 ... }

        COPY_ADDR6(req6.in6, sin6.sin6_addr);
        req6.preflen=128;
        ifr->ifr_buf6 = &req6;
        ifr->ifr_len6 = sizeof(req6);
        ifr->ifr_addr.sa_family = AF_INET6;

        if (ioctl(inet6_sock, SIOCDELADDR, ifr) < 0) {
error
} else {
succes
}
```

example.c

4.1.5 Ifconfig files

The original ifconfig used is the one included in ***net-tools-1.32-alpha.tar.gz.*** This one is highly generic and flexible. Problems have been encoutered because *struct sockaddr_in6* is larger than *struct sockaddr*.

File	Description
ifconfig.c	the ifconfig main code
config.in	added support for inet6 family
lib/inet6.c	support for inet6 family

4.1.6 ifconfig usage notes

The ifconfig is yet compatible with all family that it supported before. The option *del* has been added to support the deletion of inet6 adresses.
Usage is simple:

1. To add an adress: ifconfig eth0 inet6 12:34::56
2. To remove an adress : ifconfig eth0 inet6 del 12:34::56
3. To see all adresses in all families : ifconfig

Canonical names are not supported because a host will usually have more than one adress, and I don't see how to do the good choice.

4.2 Route

4.3 Ping

4.4 Traceroute

Chapter 5

Simple Interface Transition

5.1 Code introduction

A figure formatted as the one below lists the different files examined in the chapter:

File	Description
sit.c	Simple Internet Protocol : pseudo-device

5.1.1 Global variables

We present the global variables as follows:

Variable	Datatype	Origin	Description
sit_mtu_cache	sit_mtu_info * []	???h	

If necessary, additional information can be given such as:

5.1.2 Static variables

And the statics...

Variable	Datatype	Origin	Description
sit_device	struct device	linux/netdevice.h	initialisation des tous premiers paramètres de la structure device
sit_gc_last_run	unsigned long		???
sit_protocol	struct inet_protocol	inet/protocol.h	initialisation des paramètres protocolaires pour SIT

Once this is done, we may include any additional section to explain any mechanism that requires explanation before we proceed to a detailed code description.

5.2 sit_rcv Function

1. **Description**
Cette fonction est utilisée lors de la remont ée d'un paquet depuis la couche Ethernet. Tout d'abord le paquet passe par la couche IP, on decapsule le paquet puis ce dernier étant du type IPPROTO_IPV6 est envoyé par le biais de la structure inet_protocol (sit_protocol) sur la fonction sit_rcv(). Cette fonction modifie le protocole et envoie le paquet a IPv6 (ipv6_rcv())en lui faisant croire que le paquet vient de lui être passé par la carte elle-même.

2. **Input parameters**

Variable	Type	Origin	Description
skb	struct sk_buff *	linux/skbuff.h	contient toutes les données
idev	struct device *	linux/netdevice.h	device associé à l'adresse de destination par le routage
opt	struct options *	linux/ip.h	Not used
daddr	__u32	asm/types.h	IPv4 destination address
len	unsigned short		Not used
saddr	__u32	asm/types.h	IPv4 source address
redo	int		Not used
protocol	struct inet_protocol *	inet/protocol.h	Not used

3. **Returns** : 0

4. **Preconditions**
   - Les structures device et skbuff ne doivent pas être nulle.
   - La structure skbuff doit contenir les données pour la couche supérieur.
   - L'en tête IPv4 ne doit pas être présent.

5. **Postconditions**
   - Modification du protocole.
   - Appel à la fonction ipv6_rcv().

6. **Pieces of code are introduced as explained above:**

net/ipv6/sit.c

```
306  static int sit_rcv(struct sk_buff *skb, struct device *idev,
307                     struct options *opt,
308                     __u32 daddr, unsigned short len, __u32 saddr,
309                    int redo, struct inet_protocol * protocol)
310  {
311          struct device *dev = &sit_device;
312          struct enet_statistics *stats;
313
314          stats = (struct enet_statistics *)dev->priv;
315
316          /*
317           *      Force checksum for debug purposes
318           */
319          skb->ip_summed = CHECKSUM_NONE;
320
321          skb->h.raw = skb_pull(skb, skb->h.raw - skb->data);
322          skb->protocol = __constant_htons(ETH_P_IPV6);
323
324          skb->dev = dev;
325
326          stats->rx_packets++;
327
328          ipv6_rcv(skb, dev, NULL);
329          return 0;
330  }
```

net/ipv6/sit.c

Comments are then provided with the line numbers which are concerned:

306-309 Déclarations : Le skbuff passé en paramètre contient les données pour la couche IPv6.

321 Séparation des données, extraction du champs raw de la zone de données.

322 Modification du protocole pour faire croire à la couche du dessus que le paquet arrive directement de la couche ethernet.

5.3 sit_xmit Function

1. **Description**
Le squelette de base a été pris par rapport à ce qu'avait fait Roque. Quelques modifications ont du être apportées afin de pouvoir faire du tunneling configuré. En fait la différence entre les deux sortes de tunneling est le fait que l'un des deux est intellignet. Le tunneling automatique se fait au niveau de SIT si l'adresse de destination est une adresse IPv4 compatible. Dans ce cas, on ne cherche pas a savoir ce qui ce passe entre les deux machine, on encapsule le paquet de l'expediteur ayant appelé SIT jusqu'au recepteur (destinataire). Dans le cas du tunneling configuré, on IPv6 nous a fournit le NHOP supportant à la fois les adresses IPv4 et IPv6. Il suffit donc d'encapsuler le paquet IPv6 dans un paquet IPv4 avec comme destination l'adresse du NHOP fournie par IPv6. Pour des rainsons de simplicité, la partie de code concernant le routage a été fait dans SIT. Cela évite d'avoir une surcharge de code en faisant un appel à ip_build_xmit(). On fera donc appel à ip_queue_xmit() afin de fragmenter les paquets si nécessaire.
Les modifications de code se situent dans la partie concernant le tunneling automatique. Cela n'avait pas été prévue dans la derniere version de Roque. A ce propos, j'aimerais dire que la version de roque du routage utilise au sein de la structure des skbuff, une structure nommée neighbour. Cette structure d'après ce que j'ai pu en lire serait utilisée comme une liste chainée afin de connaitre tout les sauts permettant de passer de l'expéditeur vers le destinataire. Cette structure est ensuite utilisée dans site en remplacement du champs raddr6 dont l'adresse peut être trouvée maintenant dant le champs nexthop->addr.

2. **Input parameters**

Variable	type	Origin	Description
skb	struct sk_buff *	linux/skbuff.h	contient toutes les données
dev	struct device *	linux/netdevice.h	device associé à l'adresse de destination par le routage

3. **Returns** : 0. En cas de problème, ajoute dans les statistics une erreur.

4. **Preconditions**
   - Suppose les foinctions de routage, et de ND réalisées.

5. **Postconditions**
   - Réalisation de routage pour un plus grand confort d'utilisation.
   - Appel à la fonction ip_queue_xmit().

6. **Pieces of code are introduced as explained above:**

net/ipv6/sit.c

```
332 static int sit_xmit(struct sk_buff *skb, struct device *dev)
333 {
334   struct enet_statistics *stats;
335   struct sit_mtu_info *minfo;
336   struct neighbour *neigh;
337   struct in6_addr *addr6;
338   unsigned long flags;
339   struct rtable *rt;
340   struct iphdr *iph;
341   __u32 saddr;
342   __u32 daddr;
343   __u32 raddr;
344 /*      int addr_type;*/
345   int mtu;
346   int len;
347
348   /*
349    *    Make sure we are not busy (check lock variable)
350    */
351
352   stats = (struct enet_statistics *)dev->priv;
353   save_flags(flags);
354   cli();
355   if (dev->tbusy != 0)
356     {
357       restore_flags(flags);
358       printk(KERN_DEBUG "sit_xmit: busy\n");
359       return(1);
360     }
361   dev->tbusy = 1;
362   restore_flags(flags);
363
364   neigh = skb->nexthop;
365
366   if (neigh == NULL)
367     {
368       printk(KERN_DEBUG "sit: nexthop == NULL\n");
369       goto on_error;
370     }
371
372   if (IS_IPV4ADDR6(skb->daddr6) || IS_ANYADDR6(neigh->addr))
373     addr6=skb->daddr6;
374   else
375     addr6 = &neigh->addr;
376
377   if (!IS_IPV4ADDR6(*addr6))
378     {
379       printk(KERN_DEBUG "sit_xmit: non v4 address\n");
380       goto on_error;
381     }
382
383
384   daddr = addr6->s6_addr32[3];
385
386
387   len = skb->tail - (skb->data + sizeof(struct ipv6hdr));
388
389   if (skb->sk)
390     {
```



```
391       atomic_sub(skb->truesize, &skb->sk->wmem_alloc);
392     }
393
394   skb->sk = NULL;
395
396   iph = (struct iphdr *) skb_push(skb, sizeof(struct iphdr));
397
398   skb->protocol = htons(ETH_P_IP);
399
400   /* get route */
401
402   rt = ip_rt_route(daddr, skb->localroute);
403
404   if (rt == NULL)
405     {
406       printk(KERN_DEBUG "sit: no route to host\n");
407       goto on_error;
408     }
409
410   minfo = sit_mtu_lookup(daddr);
411
412   if (minfo)
413     mtu = minfo->mtu;
414   else
415     mtu = rt->rt_dev->mtu;
416
417   if (mtu > 576 && len > mtu)
418     {
419       icmpv6_send(skb, ICMPV6_PKT_TOOBIG, 0, mtu, dev);
420       goto on_error;
421     }
422
423   saddr = rt->rt_src;
424   skb->dev = rt->rt_dev;
425   raddr = rt->rt_gateway;
426
427   if (raddr == 0)
428     raddr = daddr;
429
430   /* now for the device header */
431
432   skb->arp = 1;
433
434   if (skb->dev->hard_header_len)
435     {
436       int mac;
437
438       if (skb->data - skb->head < skb->dev->hard_header_len)
439         {
440           printk(KERN_DEBUG "sit: space at head < dev header\n");
441           goto on_error;
442         }
443
444       if (skb->dev->hard_header)
445         {
446           mac = skb->dev->hard_header(skb, skb->dev, ETH_P_IP,
447                                       NULL, NULL, len);
448
449           if (mac < 0)
```



```
450             skb->arp = 0;
451
452           skb->raddr = raddr;
453         }
454
455     }
456
457   ip_rt_put(rt);
458
459
460   iph->version  = 4;
461   iph->ihl      = 5;
462   iph->tos      = 0;                              /* tos set to 0... */
463
464   if (mtu > 576)
465     {
466       iph->frag_off = htons(IP_DF);
467     }
468   else
469     iph->frag_off = 0;
470
471   iph->ttl      = 64;
472   iph->saddr    = saddr;
473   iph->daddr    = daddr;
474   iph->protocol = IPPROTO_IPV6;
475   skb->ip_hdr   = iph;
476
477   ip_send_check(iph);
478
479   ip_queue_xmit(NULL, skb->dev, skb, 1);
480
481   stats->tx_packets++;
482   dev->tbusy=0;
483
484   return 0;
485
486 on_error:
487   kfree_skb(skb, FREE_WRITE);
488   dev->tbusy=0;
489   stats->tx_errors++;
490   return 0;
491 }
```

net/ipv6/sit.c

Fonction utilisée lors de la descente entre IPv6 et IPv4. Pour les non initiés, IPv6 appel un device, et ce suivant le routage. Ceci se traduit par une strucure ayant des pointeurs sur fonctions. Aprés le routage, on sait donc ou on va se diriger. DAns le cas que nous voyons, il s'agit non pas d'un device, mais d'un pseudo-device appelé SIT pour Simple Internet Transition.
Comments are then provided with the line numbers which are concerned:

334-346 Déclarations : l'une des déclarations n'est pas ou n'est plus utile, voila à quoi correspond le commentaire de al ligne 344.

352 Les statistiques.

364 Recherche du NHOP, dans la liste contenant tous les sauts entre l'expéditeur et le destinataire.

366-370 Vérification que les fonction de routage ont bien fait leur travail.

372-375 Modification de ma part. Ceci a été fait pour pouvoir à la fois faire du tunneling automatique et du tunneling configuré. Pour ce qui ne saurait pas la différence entre les deux, je leur conseil de se reporter aux différentes documentations sur le sujet.

377-381 Vérification de la conformité de l'adresse IPv4 pour l'envoie des datagrammes.

384 Récupération de l'adresse dans le dernier champs de la structure d'une adresse IPv6.

396 Création d'un header IPv4.

398 Changement de protocol afin de pouvoir rediriger par la suite les datagrammes de IPv4 vers la carte Ethernet.

402-408 Appel à la fonction de routage IPv4 et verification de la validite de celle-ci.

410-421 Recherche de la valeur de la MTU pour le decoupage ou non du datagramme.

425-428 Recherche de l'adresse du prochain saut (NHOP), au cas ou cette adresse est nulle, l'adresse du NHOP est l'adresse de destination.

434-455 Vérifications de la taille des données et de l'adresse MAC, ajout du NHOP dans la structure skbuff.

460-477 Ajout d'informations dans la structure d'en-tête.

398 Appel a IPv4. Du fait du routage effectué précédemment, il n'est plus nécessaire de passer par la fonction ip_build_xmit(). De plus ceci permet de conserver les informations du skbuff.

Chapter 6

Kernel support

6.1 proc

6.2 ioctl

6.3 netlink

Chapter 7

Host / Router configuration

autoconfiguration

Chapter 8

Dictionnary

8.1 Data Structures

8.2 Globales variables

8.3 Functions

Chapter 9

Installation

compile tracing

Appendix A

man pages

Appendix B

Model

B.1 Introduction

As indicated by its name, this part introduces the IP functionality documented in the chapter. It contains such general information as the reason for this functionialty, how it is used by other source modules, where it is described (RFC, draft. . .), etc....

B.2 Code introduction

A figure formatted as the one below lists the different files examined in the chapter:

File	Description
file_1h.h	description of file_1h.h
. . .	. . .
file_nh.h	description of file_n.h
file_1c.c	description of file_1c.c
. . .	. . .
file_mc.c	description of file_mc.c

B.3 Global variables

We present the global variables as follows:

Variable	Datatype	Origin	Description
var1	type_var1	xxxxx.h	description of variable var1
. . .	. . .	. . .	. . .
varn	type_varnn	yyyyy.h	description of variable varn

If necessary, additional information can be given such as:

B.4 Statistics variables

And the statistics variable for /proc...

Name	Description
stat1	description of stat1
. . .	. . .
statn	description of variable statn

Once this is done, we may include any additional section to explain any mechanism that requires explanation before we proceed to a detailed code description.

B.5 XWZYRT@K

-Such a subsection is inserted anytime it seems necessary to detail a functionality, mechanism, etc...
-Then, important structures are described in detail:

B.6 struct_1 Structure

This is a presentation of the given structure, located in *f*ile.h. The rfh command is defined at the beginning of *j*ournal.tex.

*f*ile.h

```
51  struct route {
52    char          route_size;
        /* other declarations */
53    char          pointer;
54    unsigned long route[MAX_ROUTE];
55  };
```

*f*ile.h

Basically, this is how any piece of code will be included in the documentation. Line numbers are obtained by using `'cat -n file.h`. If some part of code must be skipped (because it need not be explained, for example), it is replaced with an explicit C comment.

B.7 function_1 Function

1. **Description** This function does...
2. **Input parameters**

Variable	Origin	Description
var1	xxxxx.h	description of variable var1
. . .	. . .	. . .
varn	yyyyy.h	description of variable varn

3. **Output parameters**

Variable	Origin	Description
var1	xxxxx.h	description of variable var1
. . .	. . .	. . .
varn	yyyyy.h	description of variable varn

4. **Returns** this variable: . . .
5. **Preconditions** describes which conditions must be fulfilled before this function is called.
   - This...
   - That...
6. **Postconditions** describes which conditions are fulfilled when the functon returns.
   - That...
   - This...

Pieces of code are introduced as explained above:

*f*ile.c


```
155 static void if_getstats(char *ifname, struct interface *ife)
156 {
157   FILE *f=fopen("/proc/net/dev","r");
158   char buf[256];
159   char *bp;
160   if(f==NULL)
161         return;
162   while(fgets(buf,255,f))
163   {
164         bp=buf;
165         while(*bp&&isspace(*bp))
166                 bp++;
167         if(strncmp(bp,ifname,strlen(ifname))==0 && bp[strlen(ifname)]==':')
168         {
169                 bp=strchr(bp,':');
170                 bp++;
          /* scanf statistics */
186                 fclose(f);
187                 return;
188         }
189   }
190   fclose(f);
191 }
```

*f*ile.c

*C*omments are then provided with the line numbers which are concerned:

`157-159` Declarations. `buff` is... etc, etc,

`162-187` The `while` loop, bla, bla, bli, blo....
And so on... for each piece of code we explain.

B.8 Last part

This can be a summary, a conclusion,... Up to you...