IPv6 COMPRESSED
1.5.0
Research Paper
peter-thoemmes.org research

© Peter Thoemmes
Weinbergstrasse 3a
D-54441 Ockfen, Germany

December 5, 2010

Abstract

This paper is a compressed guide through the IPv6 world. The things explained inside are the essence of the authors research work in the years 2008-2010. This paper is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY, without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Contents

1 Introduction

As usual I don't want to waste this paper telling general stuff about the actual subject, as almost nobody expects this kind of information from my side. For general information about IPv6 you should surf the Internet. Here you will find compressed information about what the ideas behind IPv6 are.

Outline The remainder of this document is organized as follows. Section 2 lists all the RFC documents from which I picked the theoretical information. Section 3 shows the important things to know about IPv4 addresses and Section 4 does the same for IPv6. Section 5 is about IPv6 packets, Section 6 about IPv6 Routing and Section 7 about IPv6 Protocols. Just to mention it, I show the idea of MPLS in Section 8. The biggest part of this paper is Section 9, where I show the techniques used to provide meaningful IPv4/IPv6 interoperability. In Section 10 I show the usage of the basic commandline tools and in Section 11 basic programming examples using the Socket API.

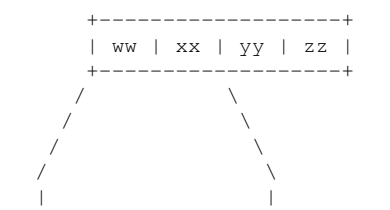
2 References

RFC 2460: IPv6 Specification (Dec 1998/Internet Engineering Task Force)
RFC 4291: IPv6 Addressing Architecture
RFC 3484: IPv6 Default Address Selection
RFC 3879: IPv6 - Deprecating Site Local Addresses
RFC 3307: IPv6 - Allocation Guidelines for Multicast Addresses
RFC 1321: MD5 (Message-Digest algorithm 5)
RFC 3031: MPLS (Multiprotocol Label Switching) Architecture
RFC 0790: ASSIGNED NUMBERS (IPv4)
RFC 0791: INTERNET PROTOCOL (IPv4)
RFC 1112: Host Extensions for IP Multicasting
RFC 0792: ICMP (Internet Control Message Protocol)
RFC 2463: ICMPv6 (Internet Control Message Protocol for IPv6)
RFC 2131: DHCP (Dynamic Host Configuration Protocol)
RFC 3315: DHCPv6 (Dynamic Host Configuration Protocol for IPv6)
RFC 3376: ICMP (Internet Group Management Protocol, Version 3)
RFC 3810: MLD (Multicast Listener Discovery for IPv6, Version 2)
RFC 0826: ARP (Ethernet Address Resolution Protocol)
RFC 2461: ND (Neighbor Discovery for IPv6)
RFC 1933: Transition Mechanisms for IPv6 Hosts and Routers
RFC 2765: Stateless IP/ICMP Translation Algorithm (SIIT)
RFC 2767: Dual Stack Hosts Using "Bump-In-the-Stack" (BIS)
RFC 3338: Dual Stack Hosts Using "Bump-in-the-API" (BIA)
RFC 2766: Network Address Translation - Protocol Translation (NAT-PT)
RFC 3142: An IPv6-to-IPv4 Transport Relay Translator (TRT)
RFC 2529: Transmission of IPv6 over IPv4 Domains without Explicit Tunnels
RFC 2893: Transition Mechanisms for IPv6 Hosts and Routers
RFC 4213: Basic Transition Mechanisms for IPv6 Hosts and Routers
RFC 3056: Connection of IPv6 Domains via IPv4 Clouds
RFC 3068: An Anycast Prefix for 6to4 Relay Routers
RFC 4214: Intra-Site Automatic Tunnel Addressing Protocol (ISATAP)
RFC 3053: IPv6 Tunnel Broker
RFC 4380: Teredo: Tunneling IPv6 through NATs using IPv4 UDP

RFC 0793: TCP (Transmission Control Protocol)
RFC 0768: UDP (User Datagram Protocol)
RFC 2960: SCTP (Stream Control Transmission Protocol)
RFC 3350: RTP (Real-time Transport Protocol)
RFC 3261: SIP (Session Initiation Protocol)

3 IPv4 addresses

An IPv4 address is made of 4 bytes (32 bits). Up to 2 bytes (the most significant ones) are used to classify the address:




```
--- SPECIAL: ------

0111 1111 0000 0000 Localhost            127.0.0.1

--- GLOBAL: -------

0... .... .... .... Class A    1.x.y.z - 126.x.y.z  NOT 127.x.y.z
10.. .... .... .... Class B  128.x.y.z - 191.x.y.z  NOT 172.16.y.z-172.31.y.z
110. .... .... .... Class C  192.x.y.z - 223.x.y.z  NOT 192.168.y.z
1110 .... .... .... Class D  224.x.y.z - 239.x.y.z  => MULTICAST
1111 .... .... .... Class E  240.x.y.z - 254.x.y.z  NOT 255.x.y.z)

--- PRIVATE: ------

0000 1010 .... .... Class A private       10.x.y.z
1010 1100 0001 .... Class B private      172.16.y.z - 172.31.y.z
1100 0000 1010 1000 Class C private      192.168.y.z
```

4 IPv6 addresses

IPv6 addresses are made of 128 bit = 16 byte = 8 words. Writing is done using HEX numbers separated by colon:

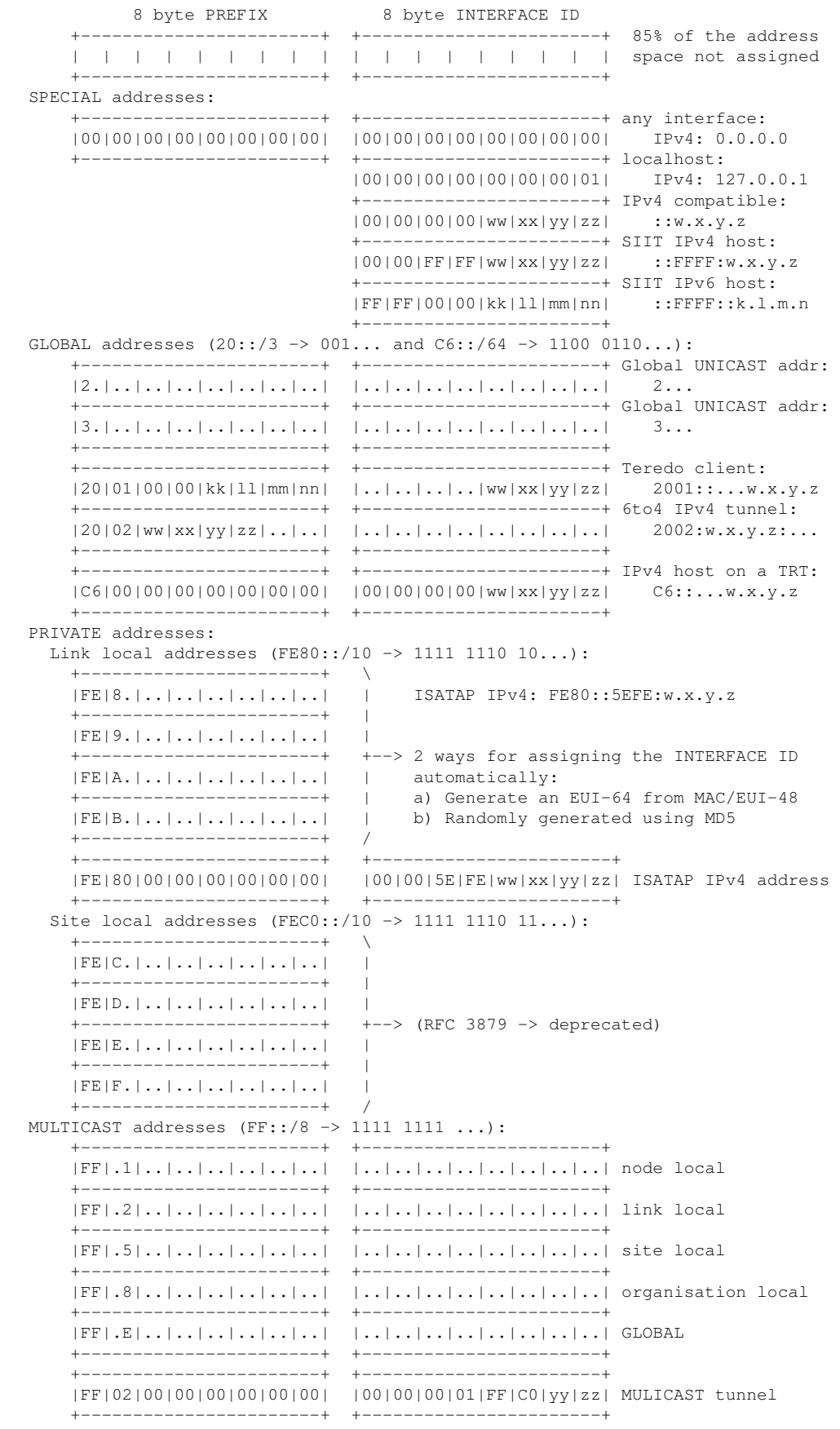
```
xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx
```

Typically zero-words (0000) can be left out (::) and leading zero-words are written in compressed form (::). The use of "::" indicates one or more groups of 16 bits of zeros and can only appear once in an address. It can be used to compress leading or trailing zeros.

```
xxxx:0000:0000:0000:yyyy:yyyy:yyyy:yyyy => xxxx::yyyy:yyyy:yyyy:yyyy

0000:0000:0000:0000:xxxx:xxxx:xxxx:xxxx => ::xxxx:xxxx:xxxx:xxxx

0000:0000:0000:0000:xxxx:xxxx:0000:yyyy => ::xxxx:xxxx:0:yyyy
```

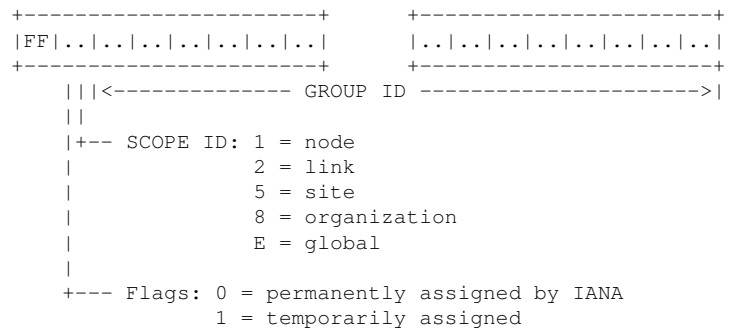
On the next page there is an overview of the classification of IPv6 addresses. I put it all onto one page, so the reader can print this out and stick it to the wall at his working place.

In general an IPv6 address is build/classified like this:



4.1 IPv6 MULTICAST addresses

Each multicast consist of 1 word prefix and 7 words GROUP ID. So there is space for plenty of groups.



There are some special permanent assigned GROUP IDs resulting in following MULTICAST addresses:

```
FF::              reserved
FF0x::1           addresses all nodes
FF0x::2           addresses all routers
FF0x::101         addresses all NTP servers
FF0x::1:FFyy:yyyy  addresses all nodes having the address ending yy:yyyy
```

There are no special BROADCAST addresses defined in IPv6. Instead there is the SCOPE ID (low half of the second byte) in a MULTICAST address.

4.2 IPv6 subnetting

Subnetting terminology is similar to the one used for IPv4 classless interdomain routing (CIDR):

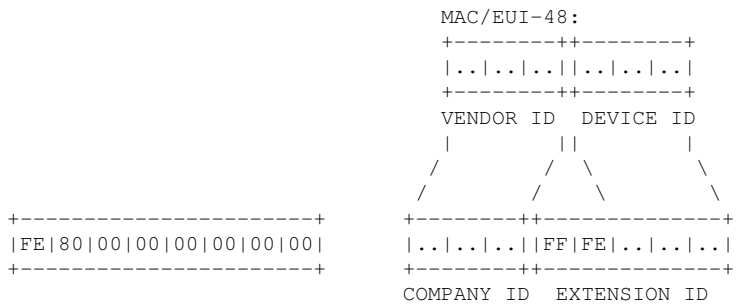
```
<net-addr>/<num-bits-netmask>

Where <net-addr> is the first IP address in the range of the network.

Examples:

    IPv4:
          192.168.5.64/26
          +--> net-addr ..: 192.168.5.64
               netmask ...: 11000000.10101000.00000101.01
               range .....: 2^6 addresses

    IPv6:
          2345:0B87:0230::/44
          +--> net-addr ..: 2345:0B87:0230:0000:0000:0000:0000:0000
               netmask ...: 0010001101000101:0000101110000111:000000100011
               range .....: 2^84 addresses

          20::/3
          +--> net-addr ..: 2000:0000:0000:0000:0000:0000:0000:0000
               netmask ...: 001
               range .....: 2^125 addresses
```

4.3 IPv6 automatic link local addresses

For a link (link = subnet = network segment = every node up to the next router) one has 2 options for assigning the INTERFACE ID automatically:

1. Generate an EUI-64 from the MAC/EUI-48

   The MAC address if build according to IEEE's EUI-48 specification consists of 3 bytes VENDOR ID (= OUI = Organisational Unique ID) and 3 bytes DEVICE ID. The resulting INTERFACE ID (EUI-64) takes over those two IDs:



2. Generate it randomly using MD5

```
1.) Read the last valid INTERFACE ID or build an EUI-64 the first time:

    xxxx:xxxx:xxxx:xxxx

2.) Run an MD5 on that INTERFACE ID ---> 128 bit MD5 check sum (16 bytes)

    yyyy:yyyy:yyyy:yyyy:zzzz:zzzz:zzzz:zzzz

3.) Cut off the least significant 8 bytes of the MD5 and set the U/L
    flag (Universal/Local flag) to 0 (local):

    INTERFACE ID = zzzz:zzzz:zzzz:zzzz & FDFF:FFFF:FFFF:FFFF
                    |
                   / \
                  /   \
                 ....  ..U. ---> Mask = FD
```

Automatic IPv6 INTERFACE ID generation is done, when there is no DHCPv6 server available (so called stateless addressing). The PREFIX is received from the router or if no router available it is simply set to '::'. An address can have one of 4 states here:

```
- tentative ...: in the state of being verified
- preferred ...: valid, meaning preferred address
- deprecated ..: still valid, but the first period of the lifetime is over
- invalid .....: invalid, meaning end-of-life
```

When going from 'tentative' to 'preferred', a 'solicitation message' is sent to locate and inform the ROUTER. The ROUTER answers with an 'advertisement message' including auto-configuration parameters.
For link local addresses IPv6 defines a so called SCOPE ID, which uniquely defines the link inside the organization's Intranet:

```
<addr>%<scope-id>

Example:

      FE80::00D0:5CFF:FE02:FCCC%1
```

4.4 IPv6 link local addresses by DHCP

The DHCP (Dynamic Host Configuration Protocol) does two things:

```
- assigning a dynamic IP address
- registering the hostname to the name server (DNS entry)
```

For doing so the DHCP client sends a UDP message to a special MULTICAST address which is

```
FF02::2:1 ---> special link-local multicast
```

By this the client targets the next DHCP Relay Agent or the directly the DHCP server. DHCP Relay Agents to delegate the request to the DHCP server by sending it to another special MULTICAST address:

```
FF02::5:3 ---> special site-local multicast
```

So if a local link does not have a DHCP server it needs at least a DHCP Relay Agent to make things working with DHCP. The ports used for the communication are:

```
DHCP client ..............: UDP port 546
DHCP server/relay agent ...: UDP port 547
```

The message send by the client is a solicit message to locate the server. The server responses by an advertisement message. The client then sends a request message and the server response is the IP address and the configuration. There is more handshakes happening thereafter, but not explained here.

4.5 IPv6 global address assignment

IPv6 addresses are having a lifetime, which can be finite or infinite. As long as not end-of-life the address is used as the PREFERRED one, but then as DEPRECATED.

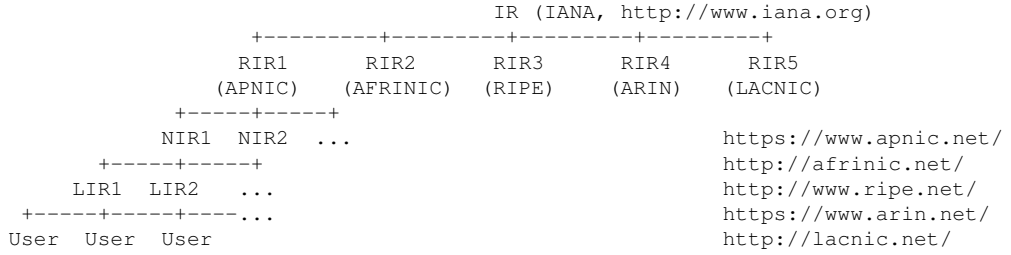
IPv6 foresees more then just one address per adapter. So each adapter may have

```
- one link local address (e.g. FE80::00D0:5CFF:FE02:FCCC)
- one or more global addresses
```

A global address can be assigned to more then just one node. This may be done in a cluster environment for load balancing (web server) or if a failover from one to another node requires the same IP address on both nodes.

The adapters of routers are assigned ANYCAST addresses, one ANYCAST address for each subnet.

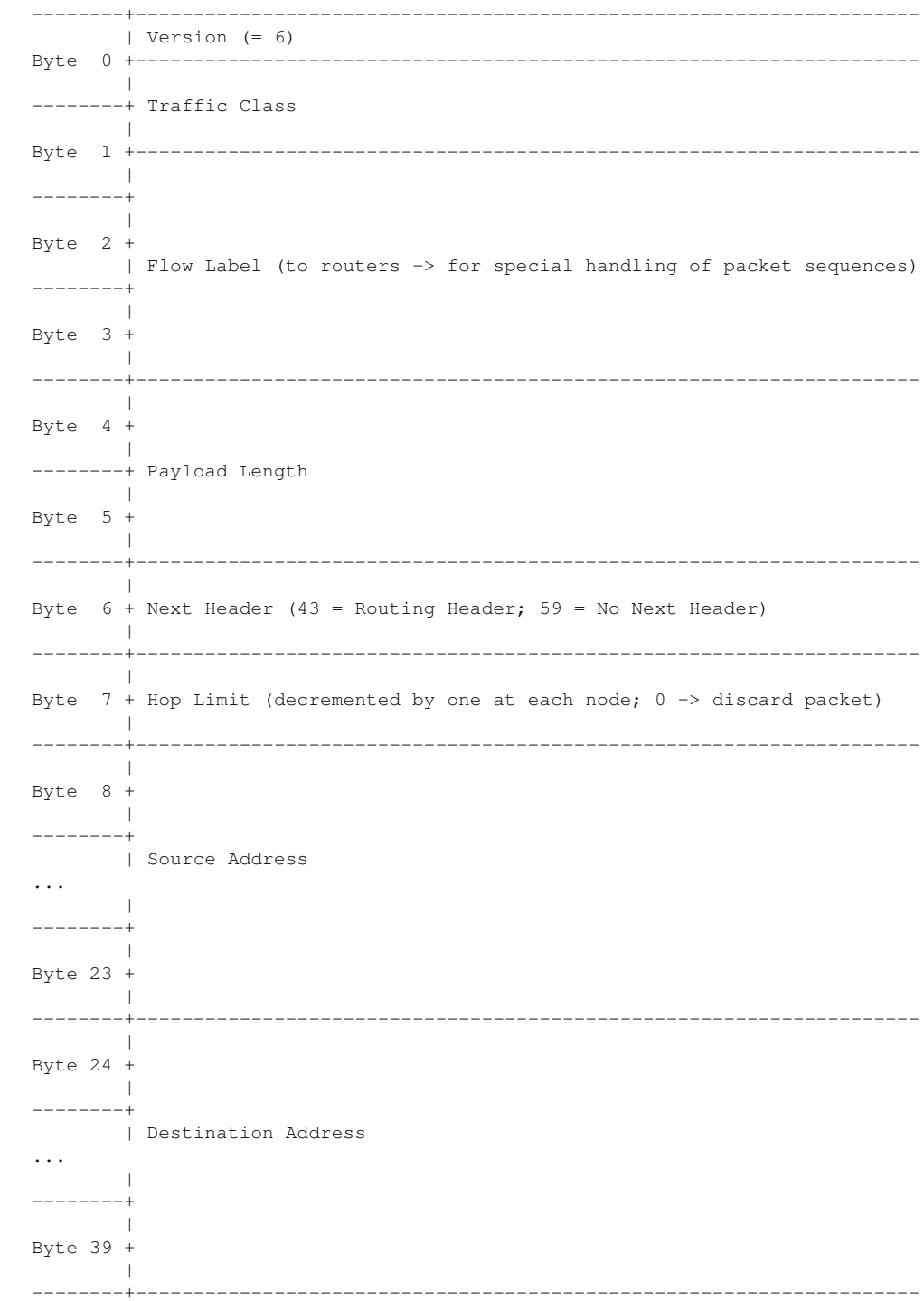
IP addresses are assigned by the IANA (Internet Assignment Numbers Authority) which is a so called IR (Internet Registry). Assignment is done hierarchical, meaning on top there is the IR, which allocates address ranges to Regional Internet Registries (RIR). Those serve and represent large geographical regions and do have the National Internet Registries (NIR) as members which are the Local Internet Registries (LIR), which are typically ISPs providing addresses to end users and address spaces to other ISPs.



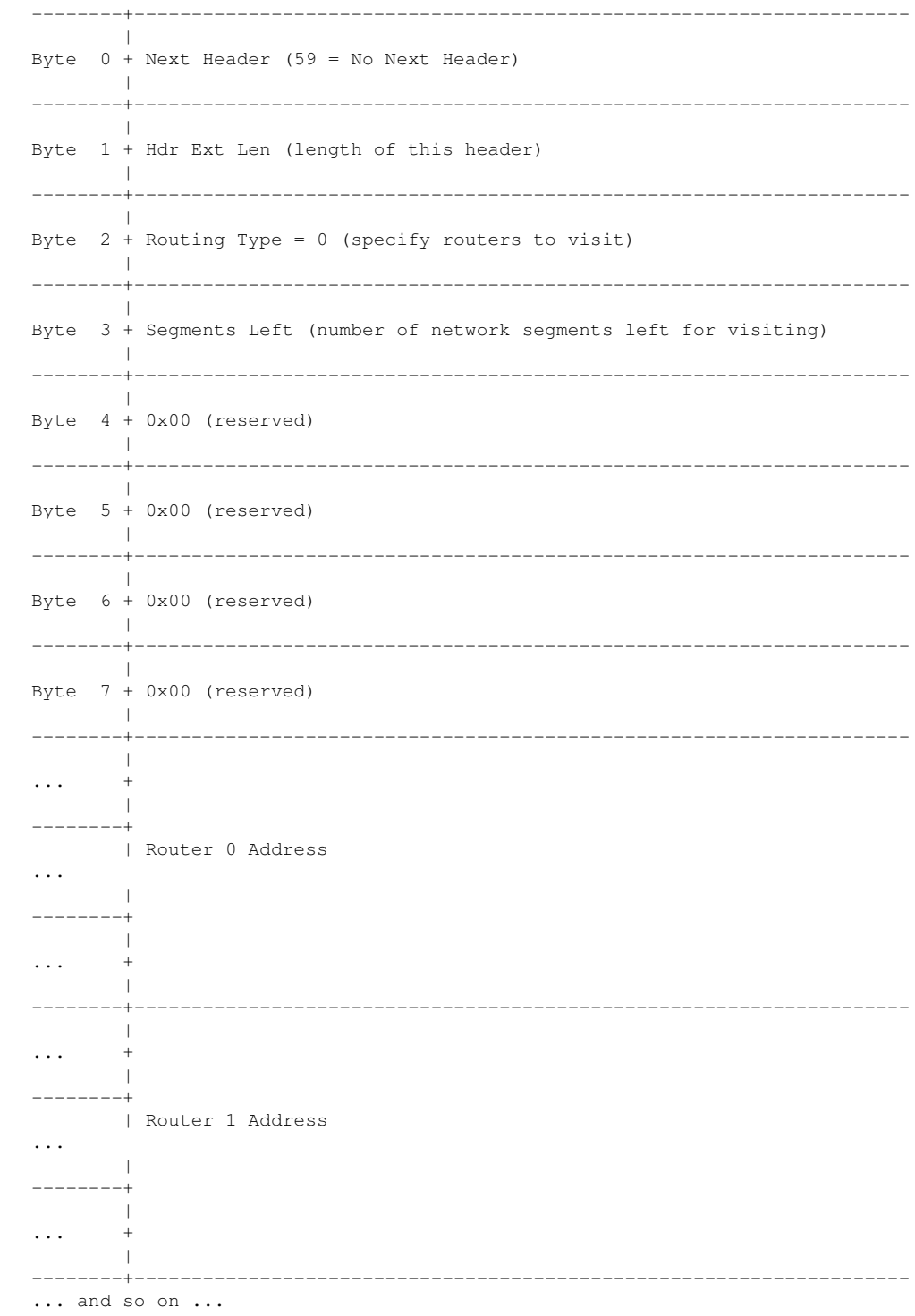
5 IPv6 Packets

The IPv6 packet is composed of three main parts: the fixed header, optional extension headers and the payload.

The fixed header has a length of 40 bytes. It can be extended and is always 8 byte aligned (64 bit aligned). A 16 bit payload length field tells the length of the payload. By setting the payload length to zero a so called 'Jumbo payload' is identified. Then the 'Jumbo Option Header' is appended to the fixed header. 'Jumbo payload' is for transporting big chunks of multimedia streams.



An interesting optional extension header is the Routing Header (Next Header field value 43). That header allows to specify one or more intermediate routers to visit, when Routing Type is 0:



Each router decrements the field 'Segments Left' and sends the packet to the next Router Address until 'Segments Left' is 0. Then it process the next header according to the 'Next Header' field.

6 IPv6 Routing

The IPv6 routing table consists of one or more routes for forwarding packets. Each route entry consists of a full configuration set:

```
- Destination ...: prefix to match for applying this route
- Next Hop ......: address to forward the packets to
- Flag ..........: flags...
           U (route is up)
           H (target is a host)
           G (use gateway)
           R (reinstate route for dynamic routing)
           D (dynamically installed by daemon or redirect)
           M (modified from routing daemon or redirect)
           A (installed by addrconf)
           C (cache entry)
           ! (reject route)
- Met ...........: metric -> distance to the target in hops
- Ref ...........: references to this route (not used in Linux kernel)
- Use ...........: lookup count -> route cache hits (-C) or misses (-F)
- If ............: outgoing interface for this route
```

Under Linux you can watch the routing table entries like this:

```
$ route -n -6 | grep -v 'lo'
Kernel IPv6 routing table
Destination                    Next Hop                   Flag Met Ref Use If
fe80::/64                      ::                         U    256 0     0 eth0
ff00::/8                       ::                         U    256 0     0 eth0

$ ip -f inet6 route
fe80::/64 dev eth0  proto kernel  metric 256  mtu 1500 advmss 1440 hoplimit ...
```

7 IPv6 Protocols

7.1 Protocols of the IP Stack

The Internet Control Messages Proctocol (ICMP) was defined for IPv4 (RFC 0792) and for IPv6 (RFC 2463). The same is true for the DHCP protocol (RFC 2131 and RFC 3315).

The discovery of MULTICAST listeners is done using IGMP (Internet Group Management Protocol, RFC 3376) in an IPv4 environment and MLD (RFC 3810) in IPv6. Both do use ICMP messages for that.

The discovery of NEIGHBORS is done in IPv4 using ARP (Ethernet Address Resolution Protocol, RFC 0826) and is done in IPv6 using ND (Neighbor Discovery, RFC 2461). ND uses ICMP messages for that.

7.2 Protocols on top of the IP Stack

The Transmission Control Protocol (TCP, RFC 0793) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks.

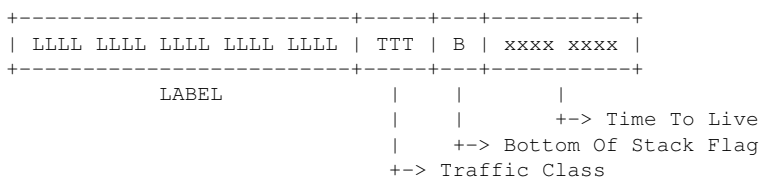
The user datagram protocol (UDP, RFC 0768) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks.

The real-time transport protocol (RTP, RFC 3350) provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services.

The session initiation protocol (SIP, RFC 3261) is an application-layer control (signaling) protocol for creating, modifying, and terminating sessions with one or more participants. These sessions include Internet telephone calls, multimedia distribution, and multimedia conferences.

8 MPLS (Multiprotocol Label Switching)

This is an additional network layer that is put right under the IP or any other network protocol layer. All packets are prefixed by one or even more labels (cascaded). Each label consists of 4 byte (32 bit):



By the bottom of stack flag the receiver can determine the number of prefixed labels.

So any network layer can be transported by an MPLS network and so companies can combine their private site networks (Intranets) using an MPLS network.

9 IPv4/IPv6 Interoperability

There are a few more or less standardized ways to interconnect IPv4 and IPv6 hosts or networks:

```
- Interconnection using a Dual Stack Translator using SIIT (RFC 2765)
- IPv4 host using SIIT - BIS (RFC 2767)
- IPv4 host using SIIT - BIA (RFC 3338)
- Interconnection using a NAT Protocol Translator (RFC 2766)
- Interconnection using a Transport Relay Translator (RFC 3142)
- Automatic Tunneling using IPv4 compatible addresses (RFC 2893)
- Automatic Tunneling using ISATAP (RFC 4214, RFC 4213)
- Automatic Tunneling of MULTICASTS (RFC 2529)
- Configured Tunneling (RFC 4213)
- 6to4 Relay Router Tunneling (RFC 3056, RFC 3068)
- Tunneling using Tunnel Broker (RFC 3053)
- Teredo Tunneling through NATs using UDP (RFC 4380)
```

In the following the big picture of each of that techniques is shown.

9.1 Dual Stack Translator using SIIT (RFC 2765)

SIIT stands for stateless IP/ICMP translation and applies temporary bidirectional address mapping. Stateless mans here that each packet is treated without relation to previous packets (no session tracking).

When receiving packets, the VERSION field of the IP header is checked first: if it is 4 (IPv4) then the packet goes through the IPv4 stack, it it is 6 (IPv6) the packet is delegated to the IPv6 stack.

When sending packets the decision on the correct stack is taken by looking at the target address: if 32 bit it goes to the IPv4 stack, if 128 bit ist goes to the IPv6 stack.

The dual stack translator allows only communication between like applications, meaning IPv4 to IPv4 and IPv6 to IPv6.

The actual translation goes like this: The hosts of the IPv6 cloud do have to be assigned special IPv6 addresses. For building those it uses a pool of IPv4 addresses:

```
IPv6 host: based on IPv4 address (k.l.m.n) ---> ::FFFF:0000:k.l.m.n
```

Those IPv4 addresses must not be used in the IPv4 cloud. If an IPv6 host wants to address an IPv4 host following mapping is applied:

```
IPv4 host: based on its IPv4 address (w.x.y.z) ---> ::0000:FFFF:w.x.y.z
```

And the story goes like this:

```
IPv4 cloud    |       TRANSLATOR        |         IPv6 cloud
              |                         |
+---------+ --|-------> k.l.m.n --------|-> +---------------------+
| w.x.y.z |   |                         |   | ::FFFF:0000:k.l.m.n |
+---------+ <-|-- ::0000:FFFF:w.x.y.z <-|-- +---------------------+
              |                         |
```

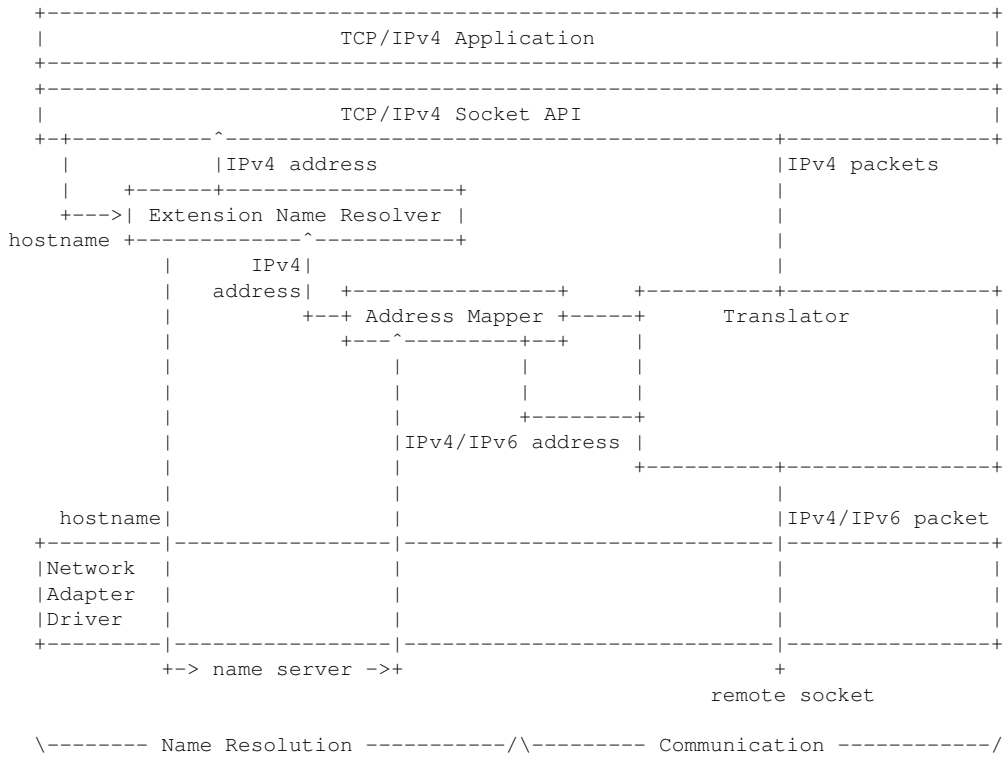
IPv4 to IPv6 translation goes like this: The IPv4 packet HEADERS are converted into IPv6 packet HEADERS. The IPv4 PAYLOAD needs to be fragmented, as typically the IPv6 header is 20 bytes larger then the IPv4 and so the payload portion in IPv6 is smaller. IPv6 to IPv4 translation is symmetrically to this, but there is no need for fragmentation.

The TCP HEADER translation goes like this:

IPv4	IPv6
Version = 4	Version = 6
Internet Header Length IHL = 5	-
Type Of Service (TOS, DiffServ)	Traffic Class
Total Len = Payload Len + IHL	Payload Len = Total Len - IHL
Identification = 0	Flow Label = 0
DF Flag = 1 (no fragmentation)	No 'Fragment' extension header
Time To Live	Hop Limit
Protocol = 6 (TCP), RFC 0790	Next Header = 6 (TCP), RFC 0790

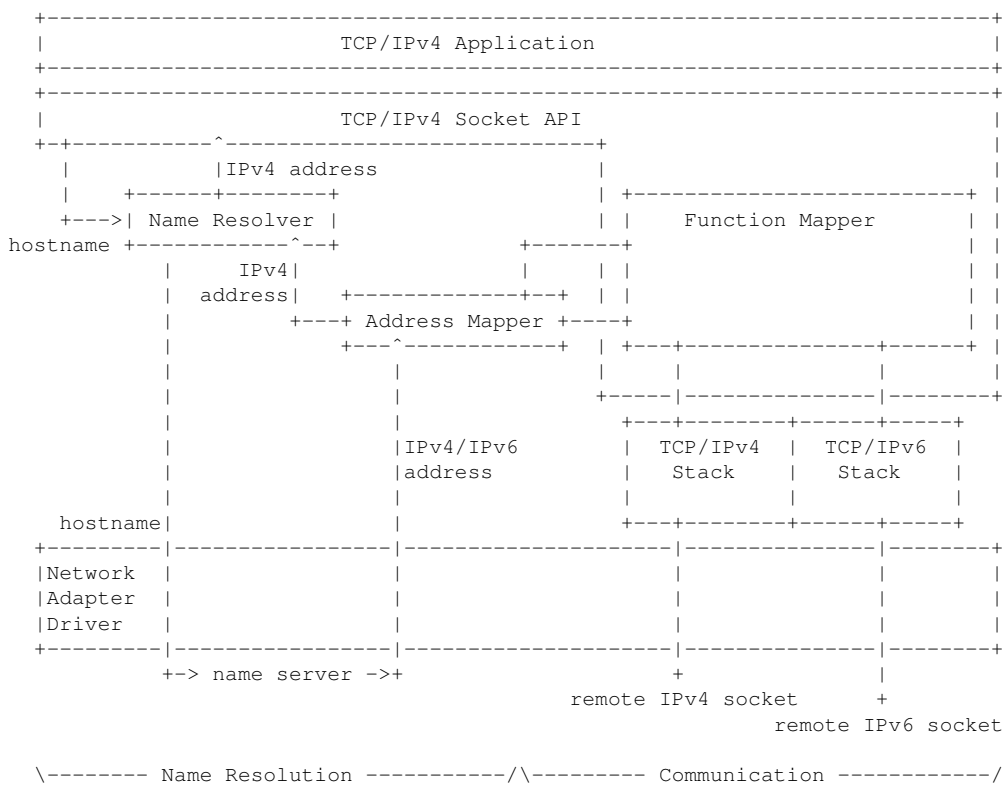
9.2 IPv4 host using SIIT - BIS (RFC 2767)

A special way of the above mentioned stateless IP/ICMP translation (SIIT) is to bump the translation into the TCP/IP stack of an host running IPv4 applications in an IPv6 network (Bump In the Stack = BIS).



9.3 IPv4 host using SIIT - BIA (RFC 3338)

Another special way of the above mentioned stateless IP/ICMP translation (SIIT) is to bump the translation into the TCP/IP socket API of an host running IPv4 applications in an IPv6 network (Bump In the API = BIA). In that case there are 2 full TCP/IP stacks: one for IPv4 and one for IPv6. On top of those the translation is happening by a function mapper, encapsulated into the TCP/IPv4 socket API. The advantage is that no IP header translation is required.



9.4 NAT Protocol Translator (RFC 2766)

NAT-PT is a stateful translation method. It uses a pool of IPv4 addresses for assignment to IPv6 nodes on a dynamic basis as sessions are initiated across IPv4-IPv6 boundaries. The IPv4 addresses are assumed to be globally unique. It provides fully transparent routing. It does, however, require NAT-PT to track the sessions it supports and mandates that inbound and outbound datagrams pertaining to a session traverse the same NAT-PT router.

The big advantage with this technique is that the IPv6 hosts don't need to be assigned to special IPv6 addresses, like with SIIT, where those need to get assigned to addresses formatted like ::FFFF:0000:k.l.m.n with k.l.m.n being IPv4 addresses not used in the connected IPv4 cloud.

9.5 Transport Relay Translator (RFC 3142)

A Transport Relay Translator is made to connect an IPv4 cloud with an IPv6 cloud TCP/UDP wise. It translates TCP,UDP/IPv6 to TCP,UDP/IPv4 and vice versa. It can not translate multicasts, but just single bidirectional connections. The translator needs to implement a dual TCP/IP stack, one for IPv4 and one for IPv6:

```
                                         \
w.x.y.z  <---- C6::w.x.y.z ----          |-> connection initiation
         ---->   k.l.m.n   ----> IPv6    /

                +-----+
           +----+ TRT +----+
           |    +-----+    |
           |               |
      +----+----+     +----+----+
      | TCP/IP  |     | TCP/IP  |
      |   v4    |     |   v6    |
      +----+----+     +----+----+
           |               |
       IPv4 cloud      IPv6 cloud
```

So the translation takes place on top of the stack inside the TCP respectively UDP packets.

All packets from the IPv6 cloud are routed towards the TRT if the address is inside C6::/64, so having the PREFIX C600:0000:0000:0000. The IPv4 address is generated from the IPv4 target w.x.y.z like this: C6::w.x.y.z.

For the IPv4 side, the routing table relays TCP and UDP traffic for specific IPv4 addresses and maps those to real IPv6 addresses in the IPv6 cloud.

9.6 Auto Tunneling/IPv4 compatible addresses (RFC 2893)

IPv6 traffic is tunneled through an IPv4 network by assigning IPv4 compatible IPv6 addresses to the IPv6 hosts. For that the IPv6 hosts need a dual stack.

```
+-----------------+  \
|       APP       |  |
+-----------------+  |
         |           |
+-----------------+  |
| TCP/IP v6 stack |  |-> IPv6 host
+-----------------+  |
         |           |
+-----------------+  |
| TCP/IP v4 stack |  |
+-----------------+  /
         |
     IPv4 cloud
         |
+-----------------+  \
| TCP/IP v4 stack |  |
+-----------------+  |
         |           |
+-----------------+  |
| TCP/IP v6 stack |  |-> IPv6 host
+-----------------+  |
         |           |
+-----------------+  |
|       APP       |  |
+-----------------+  /
```

The IPv6 traffic is encapsulated into IPv4 packets and the SRC and DST addresses are automatically generated:

```
        IPv6 addr                        IPv4 addr

        ::w.x.y.z          <--->          w.x.y.z
    (IPv4 compatible)
```

Of course the IPv4 stacks of the IPv6 hosts need to care for IPv4 address conflicts when assigning the IPv4 addresses.

For details about the encapsulation please refer to RFC 2893.

9.7 Automatic Tunneling using ISATAP (RFC 4214)

ISATAP stands for Intra-Site Automatic Tunnel Addressing Protocol. It describes automatic tunneling through an IPv4 network by assigning special IPv6 link-local addresses to the IPv6 hosts. For that the IPv6 hosts need a dual stack. ISATAP is only made for NBMA (Non Broadcast Multi-Access) IPv6 interfaces and only works in an intranet, as it works with link-local IPv6 addresses. IPv6 ND (Neighbor Discovery) is supported by ISATAP.

The special IPv6 link-local addresses are build as described in the following. The INTERFACE ID is build from the IPv4 address, the host is assigned to:

```
+------------------------+      +------------------------+
|FE|80|00|00|00|00|00|00|      |00|00|5E|FE|ww|xx|yy|zz|
+------------------------+      +------------------------+
                                              \-----------/
                                              IPv4: w.x.y.z
```

The IPv6 traffic is encapsulated into IPv4 packets and the SRC and DST addresses are automatically generated:

```
          IPv6 addr                            IPv4 addr

    FE80::5EFE:w.x.y.z          <--->          w.x.y.z
```

The underlying IPv4 carrier network only has unicast capability, no multicast.

For details about the encapsulation please refer to RFC 4213.

9.8 Automatic Tunneling of MULTICASTS (RFC 2529)

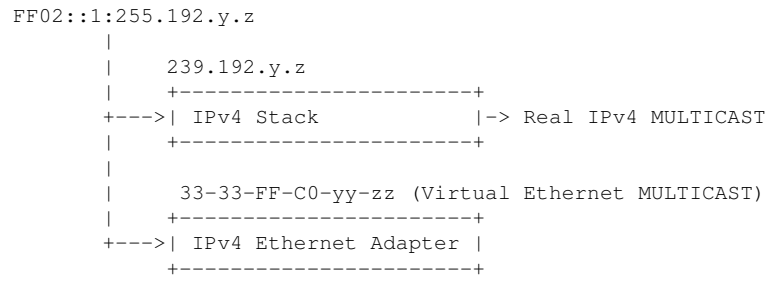
This method describes an IPv4 tunnel for IPv6 MULTICAST messages. Private IPv4 MULTICAST adresses look like this:

```
239.192.y.z   (239.192.0.0/16)
```

Ethernet MULTICAST addresses for IPv6 looks like this (RFC 3307):

```
IPv6 MULTICAST              Ethernet IPv6 MULTICAST

FF02::1:w.x.y.z     --->    33-33-ww-xx-yy-zz
```

Now for this technique special formatted IPv6 MULTICAST addresses are used. The packets are sent to an IPv6 Ethernet MULTICAST address (33-33-ww-xx-yy-zz), which is just a virtual thing as the underlying IPv4 Ethernet adapter does not really do a multicast (it would like a '01-00-5E-xx-xx-xx' for that). But due to a special implementation (and that is the actual multicast tunneler here) those packets are addressed to a real IPv4 MULTICAST network (239.192.y.z):



Sending an IPv6 link local MULTICAST (FF02:...) to the GROUP ID '1:FFxx:yyzz' will by definition address all nodes having the IPv6 address ending 'xx:yyzz'. But sending it to an IPv4 adapter it is used to address the IPv4 MULTICAST group 'x.y.z'.

That allows us even to use Neighbor Discovery for IPv6 (ND) over IPv4, as this is assigned to the special Ethernet address 33-33-00-00-00-00:

```
IPv6 ND                    Ethernet IPv6 ND

FF02::1:0.0.0.0    --->    33-33-00-00-00-00
```

So the ND implementation will send the ICMPv6 messages to the IPv4 Ethernet adapter thinking it was an IPv6 adapter and it will succeed doing the Neighbor Discovery, as the underlying IPv4 stack will do an IPv4 ARP Neighbor Discovery.

For this tunneling technique following definitions have been defined:

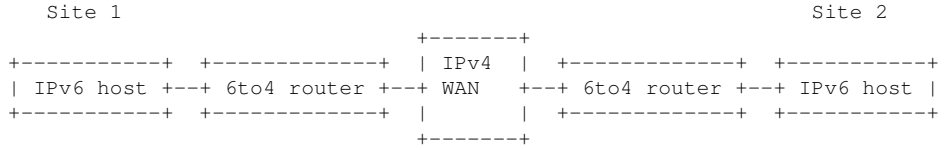
```
- Neighbor Discovery (ND):

        IPv6 address             Virtual Ethernet
      FF02::1:0.0.0.0     --->   33-33-00-00-00-00   --->  IPv4 ARP ND

- All NODES MULTICAST:

        IPv6 address             Virtual Ethernet          IPv4 address
      FF02::1:255.192.0.1 --->   33-33-FF-C0-00-01   --->   239.192.0.1

- All ROUTERS MULTICAST:

        IPv6 address             Virtual Ethernet          IPv4 address
      FF02::1:255.192.0.2 --->   33-33-FF-C0-00-02   --->   239.192.0.2
```

9.9 Configured Tunneling (RFC 4213)

This is the same kind of thing as for the automatic tunneling, but instead using special addresses here we setup explicit tunnels by configuring routers and/or hosts. The entry node of the tunnel (the encapsulator) creates an encapsulating IPv4 header and transmits the encapsulated packet. The exit node of the tunnel (the decapsulator) receives the encapsulated packet, reassembles the packet if needed, removes the IPv4 header, and processes the received IPv6 packet.

9.10 6to4 Relay Router Tunneling (RFC 3056, RFC 3068)

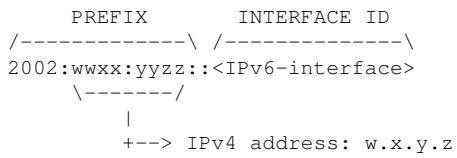
Via 6to4 relay routers a wide area IPv4 network is treated as a unicast point-to-point link layer:



An that's the way it goes:

An interim unique IPv6 PREFIX is defined for each site by embedding a globally unique IPv4 address. The 6to4 relay router extracts that IPv4 address and encapsulates the IPv6 packets into IPv4 packets and send them to that address. The globally unique IPv4 address used might even be combined with an IPv4 Network Address Translator (NAT). This means tunneling with minimal manual configuration.

The format of the IPv6 addresses of the tunnel looks like this: IANA has defined one GLOBAL address (20::/3, means binary 001...) with a 13 bit TLA (Top Level Aggregator) for 6to4 tunneling: 0 0000 0000 0010. That means the global IPv6 address range 2002::/16 is reserved for 6to4 tunneling:



So if a 6to4 relay router sees an IPv6 address starting with 2002, it will extract the globally unique IPv4 address (= the following 32 bit). So a valid 6to4 PREFIX has 48 bit length:

```
2002:wwxx:yyzz::/48   <--->   w.x.y.z
```

RFC 3068 defines a 6to4 ANYCAST ADDRESS in order to simplify the configuration of 6to4 routers:

```
          6to4 ANYCAST ADDRESS

        IPv6                  IPv4

  2002:C058:6301::        192.88.99.1
```

9.11 Tunneling using Tunnel Broker (RFC 3053)

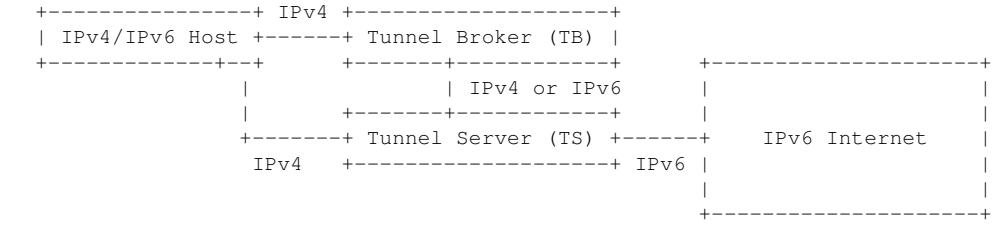
Many users are currently located behind NATs which prohibit the usage of IPv6 in IPv4 tunnels using 6to4 relay routers unless they manually reconfigure their NAT setup which in some cases is impossible as the NAT cannot be configured to forward IPv6 packets to a specific host. There might also be cases when multiple endpoints are behind the same NAT, when multiple NATs are used or when the user has no control at all over the NAT setup. This is an undesired situation as it limits the deployment of IPv6, which was meant to solve the problem of the disturbance in end

to end communications caused by NATs, which where created because of limited address space in the first place.

This problem can be solved easily by tunneling the IPv6 packets over either UDP, TCP or even SCTP. Taking into consideration that multiple separate endpoints could be behind the same NAT and/or that the public endpoint can change on the fly, there is also a need to identify the endpoint that certain packets are coming from and endpoints need to be able to change e.g. source addresses of the transporting protocol on the fly while still being identifiable as the same endpoint. The protocol described here is independent of the transport and payload's protocol. An example could be IPv6 in UDP in IPv4, which is a typical setup that can be used by IPv6 Tunnel Brokers.

The Tunnel Broker idea is an alternative approach based on the provision of dedicated servers, called Tunnel Brokers, to automatically manage tunnel requests coming from the users. This approach is expected to be useful to stimulate the growth of IPv6 interconnected hosts and to allow early IPv6 network providers to provide easy access to their IPv6 networks.

Compared to the 6to4 mechanisms they serve a different segment of the IPv6 community. The Tunnel Broker fits well for small isolated IPv6 sites, and especially isolated IPv6 hosts on the IPv4 Internet, that want to easily connect to an existing IPv6 network. The 6to4 approach has been designed to allow isolated IPv6 sites to easily connect together without having to wait for their IPv4 ISPs to deliver native IPv6 services. This is very well suited for extranet and virtual private networks. Using 6to4 relays, 6to4 sites can also reach sites on the IPv6 Internet.



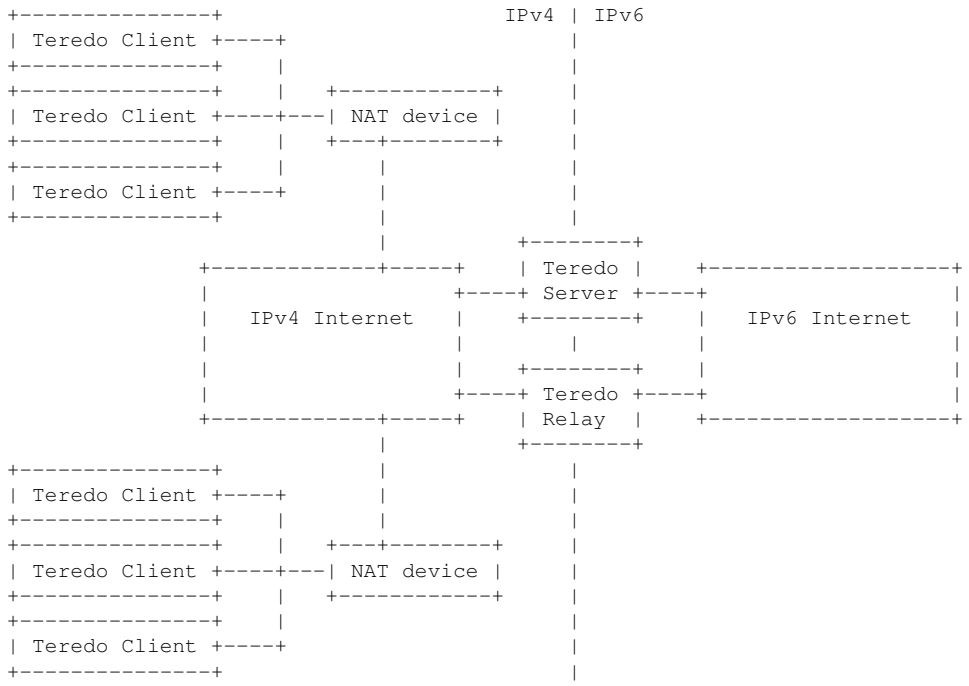
The IPv4/IPv6 dual stack host contacts a well known Tunnel Broker (TB) to get connected to a tunnel. The TB registers that hosts and provides an IPv6 address to the host. It may also register the IPv6 address and name in the DNS. Then it provides access to an appropriate Tunnel Server (TS), which actually serves the tunnel to the host. The TS is an IPv4/IPv6 dual stack router.

An interesting implementation based on the IPv6 Tunnel Broker is AYIYA (Anything In Anything) using TIC (Tunnel Information & Control protocol). For the user there is the tool AICCU (Automatic IPv6 Connectivity Client Utility) to get automatically connected to a tunnel.

```
AYIYA .....: http://www.sixxs.net/tools/ayiya/
TIC .......: http://www.sixxs.net/tools/tic/
AICCU .....: http://www.sixxs.net/tools/aiccu/
```

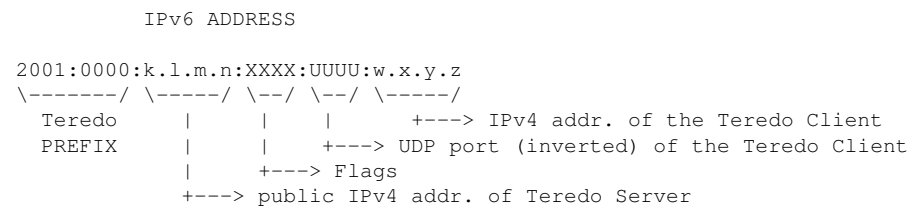
9.12 Teredo Tunneling through NATs (RFC 4380)

Teredo specifies tunneling through IPv6-unaware NAT devices. This is done by encapsulating the IPv6 packets into IPv4 UDP datagrams. 6to4 relay routers cannot be used here, as the hosts behind the NAT do not have globally unique IPv4 addresses. The only way allowing access to 6to4 relay routers would be to implement a 6to4 tunnel endpoint into the NAT device.



Teredo is a special tunnel broker implementation and that's the way it goes:

Teredo uses the fact that NAT devices typically forward IPv4 UDP datagrams without problem. So IPv4 UDP is used for tunneling. The Teredo clients are assigned special GLOBAL IPv6 addresses using the Teredo PREFIX (2001:0000::/32) and looking like this:



Teredo Clients contact the (well known) Teredo Server by its IPv4 address and do request a tunnel to a certain IPv6 address (host) in the IPv6 Internet. The Teredo Server sends out ICMPv6 messages to the IPv6 Internet and get back a reply from the IPv6 host, which is send back using the route over the closest Teredo Relay. Then it tells the requesting Teredo Client the address and the UDP port of the Teredo Relay and the client establishes the IPv4 UDP tunnel to that relay and starts the communication with the IPv6 host through that tunnel. The Teredo Relay then advertises the IPv6 address of the Teredo Client to the IPv6 Internet. The IPv6 host sends back its packets to the given IPv6 address, which is routed to the Teredo

Relay and from there to the Teredo client by using the UDP port (UUUU; inverted) and the IPv4 address (w.x.y.z) from the IPv6 address.

One Teredo Relay can serve only a limited number of IPv6 hosts from the IPv6 Internet to Teredo Clients, as there is only one client connection per tunnel, meaning one client connection per UDP port possible. An example for a public Teredo Server is the one run by Microsoft (USA/Redmond):

```
teredo.ipv6.microsoft.com
```

The nickname 'Teredo' is the name of one species of the shipworm. Shipworms are known to pierce holes into ships, like Teredo pierces wholes (tunnels to the IPv6 Internet) into a NAT device.

Interclient IPv6 connections are possible by Teredo as well, but the clients need to use the same UDP port, they would use when contacting the Teredo server, as the UDP port is part of there IPv6 address.

Examples for Teredo implementations:

```
- Teredo Client support built-in in Microsoft Windows XP SP2
- Full Teredo built-in support in Microsoft Windows Vista and Windows 7
- Miredo for Linux, FreeBSD, NetBSD and Darwin (Mac OS X's core)
- NICI-Teredo for Linux
```

10 IPv4/IPv6 Administration

There is some basic commandline tools for both, IPv4 and IPv6. It is essentially required to be familiar with those tools before starting any kind of socket programming.

To be able to use IPv6 when your machine is part of an IPv4 network or even behind a NAT device, an easy way to get IPv6 connectivity is to get an account at the SixXS web site http://www.sixxs.net/signup.

Remark: As long as you can't login ('User hasn't been activated yet') you will have to wait for the validation email by SixXS.

Once you have a valid account you will need to download the AICCU (Automatic IPv6 Connectivity Client Utility) from https://www.sixxs.net/tools/aiccu/ and install it:

```
- Under Linux/UNIX:

   Build from the source 'aiccu_20070115.tar.gz'. For some distributions
   there is a binary package, i.e. for ubuntu (# apt-get install aiccu).

      To change the configuration:

         # vi /etc/aiccu.conf
         ...
         protocol tic
         server tic.sixxs.net
         username <SixXS-user>
         password <SixXS-passwd>
         ipv6_interface sixxs
         behindnat true
         ...

         # chmod 644 /etc/aiccu.conf

      To start the client and hence setup a tunnel to the IPv6 Internet:

         $ aiccu start

- Under Windows:

   Install driver 'tap32-driver-v9' by running 'addtap.bat'. Then run
   'aiccu-2006-07-23-windows-gui.exe'.
```

A quick an easy IPv6 connection check can be done using the checker at

```
http://www.ipv6forum.com/:
```

You should get a result like this:

```
IPv4/IPv6 Checker:  IPv6: 2a01:198:200:854::2 (Global)
```

Once your tunnel is fine, you can verify different things with the basic commandline tools:

Check the TCP and UDP connections with netstat:

```
$ netstat -tuna -p
Active Internet connections (servers and established)
Proto .. Local Address           Foreign Address      State    PID/Progr
...
tcp6  .. :::22                   :::*                 LISTEN   2379/sshd
tcp6     ::1:631                 :::*                 LISTEN   3124/cupsd
tcp6     2a01:198:200:854::48033 2002:d2ab:e22d::1:80 SYN_SENT 4112/firefox
...
```

Check the interfaces of your machine:

```
$ ifconfig -a
...
sit0      Link encap:IPv6-in-IPv4
          NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
...
sixxs     Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-...-00-00
          inet6 addr: 2a01:198:200:854::2/64 Scope:Global
          inet6 addr: fe80::98:200:854:2/64 Scope:Link
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1280  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 B)  TX bytes:96 (96.0 B)
```

Send ICMPv6 echo messages using the 'ping' and 'ping6' command:

```
--- IPv4: ---

$ ping www.ipv6forum.com
PING www.ipv6forum.com (158.64.50.42) 56(84) bytes of data.
64 bytes from unknown.uni.lu (158.64.50.42): icmp_seq=1 ttl=50 time=87.1 ms
64 bytes from unknown.uni.lu (158.64.50.42): icmp_seq=2 ttl=50 time=86.9 ms
64 bytes from unknown.uni.lu (158.64.50.42): icmp_seq=3 ttl=50 time=87.9 ms
64 bytes from unknown.uni.lu (158.64.50.42): icmp_seq=4 ttl=50 time=84.5 ms
64 bytes from unknown.uni.lu (158.64.50.42): icmp_seq=5 ttl=50 time=85.1 ms

--- IPv6: ---

$ ping6 www.ipv6forum.com
PING www.ipv6forum.com(2001:a18:1:20::42) 56 data bytes
64 bytes from 2001:a18:1:20::42: icmp_seq=1 ttl=55 time=102 ms
64 bytes from 2001:a18:1:20::42: icmp_seq=2 ttl=55 time=99.1 ms
64 bytes from 2001:a18:1:20::42: icmp_seq=3 ttl=55 time=99.2 ms
```

Do a name server lookup using 'nslookup' for both protocols IPv4 and IPv6:

```
$ nslookup
> set q=any
> www.ipv6forum.com
Server:     192.168.2.1
Address:    192.168.2.1#53

Non-authoritative answer:
www.ipv6forum.com   has AAAA address 2001:a18:1:20::42
Name:   www.ipv6forum.com
Address: 158.64.50.42

Authoritative answers can be found from:
> exit
```

Do a telnet session to test the TCP stack:

```
$ telnet www.ipv6forum.com 80
Trying 2001:a18:1:20::42...
Connected to www.ipv6forum.com.
Escape character is '^]'.
```

11 IPv4/IPv6 SOCKET API

In general the C programming language API for socket programming remains the same. But there are some extensions and some changes. Look at the table for the most important changes:

IPv4 SOCKET API	IPv6 SOCKET API

struct sockaddr_in sin4;
sin4.ain_family = AF_INET;
sin4.sin_port = htons(port);
sin6.sin_addr =
inet_addr("192.168.96.5");
	struct sockaddr_in6 sin6;
	sin6.sin6_family = AF_INET6;
	sin6.sin6_port = htons(port);
	inet_pton(
	AF_INET6,
	"FF02::6790:90A",
	&sin6.sin6_addr);
	sin6.sin6_scope_id = 0;
	sin6.sin6_flowinfo = 0;

gethostbyname(), gethostbyaddr()	getaddrinfo(), getnameinfo()
inet_addr(), inet_ntoa()	inet_pton(), inet_ntop()

Example code for IPv4/IPv6:

```
/*
 * Includes (same for IPv4/IPv6):
 *
 */

#include <unistd.h>
#include <time.h>
#include <limits.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <assert.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/file.h>
#include <sys/select.h>

#include <netdb.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <netinet/ip.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

/*
 * Typedefs (same for IPv4/IPv6):
 *
 */

typedef int16_t int16;
typedef int32_t int32;
typedef int64_t int64;

typedef uint16_t uint16;
typedef uint32_t uint32;
typedef uint64_t uint64;

/*
 * Function to read from an open socket (same for IPv4/IPv6):
 *
 * @sock ..........: file descriptor of the socket
 * @szBuf .........: buffer to store the data read (will be zero terminated)
 * @nLen ..........: length of szBuf
 * @wTimeout_ms ...: max. time to wait for data
 * @wMinNumBytes ..: min. number of bytes to read (0 = undefined)
 *
 */

bool recv_data(
             int sock,
             char* szBuf,
             size_t nLen,
             uint16 wTimeout_ms,
             uint16 wMinNumBytes = 0)
{
    if(!szBuf)
        return false;
    if(!nLen)
        return false;
    szBuf[0] = (char) 0x00;

    struct timeval tvTO;
    tvTO.tv_sec = wTimeout_ms/1000;
    tvTO.tv_usec = wTimeout_ms - (tvTO.tv_sec * 1000);
    tvTO.tv_usec *= 1000;
```

```
    //Number of bytes to read:
    int iStillToRead = wMinNumBytes;
    if(iStillToRead < 0)
        iStillToRead = 0;

    //Local buffer:
    char szLocalBuf[512] = "";
    int iBufSize = 512;
    if(iStillToRead)
    {
        if(iBufSize > iStillToRead)
            iBufSize = iStillToRead;
    }

    //Read the data:
    uint32 dwRetries = 3;
    size_t nBufIdx = (size_t) -1;
    fd_set fdsetRead;
    for(;;)
    {
        //Check if the socket is ready for reading:
        FD_ZERO(&fdsetRead);
        FD_SET(sock,&fdsetRead);
        int num_socks_ready = select(
                                sock + 1,
                                &fdsetRead, //read
                                (fd_set*) 0, //write
                                (fd_set*) 0, //except
                                &tvTO);
        if(num_socks_ready <= 0) //if not at least one socket (sock) is ready
        {
            if(!--(dwRetries))
                return false;
            usleep(100000); //100 ms
            continue; //retry
        }

        //Read after socket is ready for reading:
        int iBytesRead = recv(sock,szLocalBuf,iBufSize,0);
        if(iBytesRead <= 0) //-> connection lost (0 -> gracefully lost)
        {
            if(!--(dwRetries))
                return false;
            usleep(100000); //100 ms
            continue; //retry
        }

        //Get the bytes into the main buffer:
        for(int i = 0;i < iBytesRead;++i)
        {
            szBuf[++nBufIdx] = szLocalBuf[i];
            if(nBufIdx == (nLen - 2))
                break;
        }
        szBuf[nBufIdx + 1] = (char) 0x00; //end of string

        //Check if there is still a minimum of bytes to read:
        if(iStillToRead)
        {
            if(iStillToRead > iBytesRead)
                iStillToRead -= iBytesRead;
            else
                iStillToRead = 0;
            if(iBufSize > iStillToRead)
                iBufSize = iStillToRead;
        }
        if(!iStillToRead)
            break; //ok
    }

    return true;
}
```

```
/*
 * Cleanup a socket and its file descriptor (same for IPv4/IPv6):
 *
 */

void cleanup(int& sock)
{
    shutdown(sock,SHUT_RDWR); //stop sending and receiving
    close(sock); //close the socket
    sock = -1;
}
```

In the following you find two implementations of main(), one for IPv4 and one for IPv6. They don't differ much and the code could be intermixed, but to show the difference more clear I decided to keep it separated.

```
/*
 * IPv4: Open a TCP connection and wait for a welcome message:
 *
 */

int main(void)
{
    char szIP4Addr[] = "193.99.144.85";
    uint16 wPort = 80;

    printf("IPv4 - setting up TCP connection to '%s:%u'...",szIP4Addr,wPort);
    fflush(stdout);

    protoent* pPE = getprotobyname("tcp");
    if(!pPE)
    {
        printf("failed.\n");
        perror("getprotobyname() failed");
        return -1;
    }

    int sock = socket(AF_INET,SOCK_STREAM,pPE->p_proto);
    if(sock < 0)
    {
        printf("failed.\n");
        fflush(stdout);
        perror("socket() failed");
        return -2;
    }

    sockaddr_in IPv4Server;
    IPv4Server.sin_family = AF_INET;
    IPv4Server.sin_addr.s_addr = inet_addr(szIP4Addr);
    IPv4Server.sin_port = htons(wPort);
    int iRC = connect(sock,(sockaddr*) &IPv4Server,sizeof(IPv4Server));
    if(iRC < 0)
    {
        printf("failed.\n");
        fflush(stdout);
        cleanup(sock);
        return -3;
    }
    uint32 dwNonBlocking = 1; //nonzero: non-blocking mode, 0: blocking mode
    if(ioctl(sock,FIONBIO,&dwNonBlocking) < 0)
    {
        printf("failed.\n");
        fflush(stdout);
        perror("ioctl() failed");
        cleanup(sock);
        return -4;
    }

    printf("ok.\n");

    printf("Reading welcome message:\n");

    char szMsg[1024 + 1] = "";
    recv_data(sock,szMsg,1024,1000);
    printf("[%s]\n",szMsg);

    printf("Disconnecting...");
    fflush(stdout);
    cleanup(sock);
    printf("ok.\n");

    return 0;
}
```

```
/*
 * IPv6: Open a TCP connection and wait for a welcome message:
 *
 */

int main(int argc,char* argv[])
{
    char szIP6Addr[] = "2001:a18:1:20::42";
    uint16 wPort = 80;

    printf("IPv6 - setting up TCP connection to '%s:%u'...",szIP6Addr,wPort);
    fflush(stdout);

    protoent* pPE = getprotobyname("tcp");
    if(!pPE)
    {
        printf("failed.\n");
        perror("getprotobyname() failed");
        return -1;
    }

    int sock = socket(AF_INET6,SOCK_STREAM,pPE->p_proto);
    if(sock < 0)
    {
        printf("failed.\n");
        fflush(stdout);
        perror("socket() failed");
        return -2;
    }

    struct sockaddr_in6 IPv6Server;
    IPv6Server.sin6_family = AF_INET6;
    IPv6Server.sin6_port = htons(wPort);
    inet_pton(AF_INET6,szIP6Addr,&IPv6Server.sin6_addr);
    IPv6Server.sin6_scope_id = 0;
    IPv6Server.sin6_flowinfo = 0;
    int iRC = connect(sock,(sockaddr*) &IPv6Server,sizeof(IPv6Server));
    if(iRC < 0)
    {
        printf("failed.\n");
        fflush(stdout);
        cleanup(sock);
        return -3;
    }
    uint32 dwNonBlocking = 1; //nonzero: non-blocking mode, 0: blocking mode
    if(ioctl(sock,FIONBIO,&dwNonBlocking) < 0)
    {
        printf("failed.\n");
        fflush(stdout);
        perror("ioctl() failed");
        cleanup(sock);
        return -4;
    }

    printf("ok.\n");

    printf("Reading welcome message:\n");

    char szMsg[1024 + 1] = "";
    recv_data(sock,szMsg,1024,1000);
    printf("[%s]\n",szMsg);

    printf("Disconnecting...");
    fflush(stdout);
    cleanup(sock);
    printf("ok.\n");

    return 0;
}
```