IPv6 – The Next Generation Internet

IPv6 Overview

- IPv6 is naturally multi-homed, that is, each network interface may have more than one IPv6 address.

- An IPv6 address has a scope, that is, it can be restricted to a single LAN or a private network, or have global uniqueness.

I

- Pv6 Addresses may be assigned to interfaces using one of three methods:

    - Stateful - Statically assigned = manual configuration
    - Stateful - DHCPv6 - Automatically assigned
    - Stateless - Automatically assigned

- The following table defines the types of IPv6 addresses that can be supported and contrasts them with the closest IPv4 functional equivalent.

IPv6 Name	Scope/Description	IPv4 Equivalent	Notes
Link-Local	Local LAN only. Automatically assigned based on MAC. Cannot be routed outside local LAN.	No real equivalent. Assigned IPv4 over ARP'd MAC.	Scoped address concept new to IPv6.
Site-Local	Optional. Local Site only. Cannot be routed over Internet. Assigned by user.	Private network address with multi-homed interface is closest equivalent.	Scoped address concept new to IPv6. Unlike the IPv4 private network address the IPv6 device can have, and most likely will have, Link-Local, Site-Local and a Global Unicast address. The address block used for this purpose has been marked Reserved by IANA.
Global Unicast	Globally unique.	Global IP address.	Pv6 and IPv4

	Fully routable. Assigned by IANA/delegated Aggregators.		similar but IPv6 can have other scoped addresses.
Multicast	One-to-many. Hierarchy of multicasting.	Similar to IPv4 Class D.	Significantly more powerful than IPv4 version. No broadcast in IPv6, replaced by multicast.
Anycast	One-to-nearest. Uses Global Unicast Addresses. Routers only. Discovery uses.	Unique protocols in IPv4 e.g. IGMP.	Some anycast addresses reserved for special functions.
Loopback	Local interface scope.	Same as IPv4 127.0.0.1	Same function

IPv6 Address Notation

- An IPv6 address consists of 128 bits - an IPv4 address consists of 32 bits - and is written as a series of 8 hexadecimal strings separated by colons. Examples:
- All the following refer to the same address

  2001:0000:0234:C1AB:0000:00A0:AABC:003F
- Leading zeros can be omitted

  2001:0:234:C1AB:0:A0:AABC:3F
- Not case sensitive - any mixture allowed

  2001:0:0234:C1ab:0:A0:aabc:3F
- One or more zeros entries can be omitted entirely but only once in an address.

- We can select the most efficient place to omit multiple zero entries. Examples:
- A raw ipv6 address:

  2001:0000:0234:C1AB:0000:00A0:AABC:003F
- Address with single 0 dropped:

  2001::0234:C1ab:0:A0:aabc:003F
- Alternate version - address with single 0 dropped:

  2001:0:0234:C1ab::A0:aabc:003F
- The following is not valid:

  2001::0234:C1ab::A0:aabc:003F

- Multiple zeros can be omitted entirely but only once in an address. Examples:
- Omitting multiple 0's in address:

  2001:0:0:0:0:0:0:3F
- The above can be written as:

  2001::3F
- Lots of zeros (loopback address)

  0:0:0:0:0:0:0:1
- The above can be written as:

  ::1
- All zeros (unspecified a.k.a unassigned IP)

  0:0:0:0:0:0:0:0
- The above can be written as:

  ::

- However, the following address

  2001:0:0:1:0:0:0:3F

- Cannot be reduced to:

  2001::1::3F // NOT VALID!

- It can only be reduced to

  2001::1:0:0:0:3F

- Or

  2001:0:0:1::3F

IPv6 Prefix or Slash Notation

- IPv6 uses a similar / (forward slash) notation to IPv4 CIDR (Classless Interdomain Routing), which describes the number of contiguous bits used.

- Formally this way of writing and address is called an IP prefix but more commonly called the slash format. Examples:

- Single user address:

  2001:db8::1/128

- Normal user IPv6 address allocation allows the user to control the low order 80 bits:

  2001:db8::/48

- Global routing prefix - top 3 bits only with fixed value 001 (binary)

  2::/3

IPv6 Address Types

- The type of IP address is defined by a variable number of the top bits known as the Binary Prefix (BP).
- Only as many bits as required are used to identify the address type as shown in the following table (defined in RFC 3513).

Use	Binary Prefix	Slash	Description/Notes
Unspecified	00...0	::/128	IPv6 address = 0:0:0:0:0:0:0:0 (or ::) Used before an address allocated by DHCP (equivalent of IPv4 0.0.0.0)
Loopback	00...1	::1/128	IPv6 address = 0:0:0:0:0:0:0:1 (or ::1) Local PC Loopback (equivalent of IPv4 127.0.0.1)
Multicast	1111 1111	FF::/8	(See format below)
Link-Local unicast	1111 1110 10	FE8::/10	Local LAN scope. Lower bits created from MAC address.
Site-Local unicast	1111 1110 11	FEC::/10	Local Site scope. Lower bits assigned by user. This binary prefix has been marked Reserved by IANA to reflect the currently unsupported state of Site-Local addressing.
Global Unicast	All other values	2::/3	A note in RFC 3513 suggests that IANA should continue to allocate only from the binary prefix 001 (as in RFC

Use	Binary Prefix	Slash	Description/Notes
			2373 version) for the time being.

IPv6 Global Unicast Address Format

- The IPv6 Global Unicast 128 bits are divided into a 48 bit global routing prefix (a.k.a site prefix) which is assigned by various authorities and 80 bits which define the subnet ID and interface ID as follows:

Site Prefix of 48 bits - assigned by IANA/Aggregator.		
Name	Size	Description/Notes
global routing prefix	48 bits	Variable format depending on Binary Prefix e.g. if 001 - Global Unicast Address (assigned by IANA) uses this format.
subnet ID	16 bits	Used for subnet routing.
interface ID	64 bits	The unique interface identifier (host address equivalent in IPv4).

- The current IPv6 address allocation policy adopted by the various Internet Registries is based on the IETF/IAB recommendation (in RFC 3177) and allows for:

Name	Allocation	Description/Notes
Normal User	/48	The user controls the full 80 bits addresses comprising the subnet ID and interface ID
Single subnet	/64	Where it is known that only a single subnet can be used the user is assigned control of the interface ID part only
Single Device	/128	Where it is known that only one device can be used the user is assigned a single interface ID

IPv6 End-User Address Format

- End-User addresses are assigned from Global Unicast pool and currently only with the binary prefix 001 (or 2::/3).
- The IETF 6bone currently uses a special range of 3FFe::/16 but the 6bone is being disbanded (reflecting the production ready state of IPv6) and the address range was planned to be returned to the IANA Reserved pool by June 2006.
- The 128 bits breakdown as follows:

global routing prefix of 48 bits - assigned by IANA/Aggregator.		
Name	Size	Description/Notes
reserved	3 bits	001 - Global Unicast Address (assigned by IANA)
TLA ID	13 bits	0 0000 0000 00001 (address block 2001::/16) Top Level Aggregator (TLA). Assigned by IANA for use by the Regional Internet Registries (RIRs)
Sub-TLA	13 bits	Assigned by IANA to the RIRs. The RIRs assign blocks from this range to the National or Local Internet Registries.
NLA	19 bits	Assigned by RIR to Next Level Aggregator (NLA) (either a National or Local Internet Registry or in some cases an ISP). The NLA assigns blocks from its allocated range to end-users

80 bits - typically assigned by the user		
Name	Size	Description/Notes
subnet ID	16 bits	Used for subnet routing
interface ID	64 bits	Equivalent to IPv4 host address - since this field alone is bigger that the whole IPv4 address space it is fairly generous!

IPv6 Link-Local Address Format

- Link-Local addresses are automatically assigned by the end user equipment and require no external configuration.
- The address format uses a unique binary prefix (FE8::/10) and the remaining bits (118) are built from the local interface identifier.
- In the case of Ethernet the MAC (48 bits) is used to create the EUI-64 value as shown below. If an interface identifier has more than 118 bits the link-local address cannot be generated and the unit must be manually configured.
- Link-local addresses are not routable outside the local LAN. The 128 bits of a link-local address for an Ethernet interface breakdown as follows:

10 bits - Binary Prefix		
Name	Size	Description/Notes
Binary Prefix	10 bits	1111 1110 10 or FE8::/10 Link-Local Prefix
118 bits - constructed from interface MAC (EUI-64)		
Name	Size	Description/Notes
-	54 bits	all zeros - padding from 64 to 118 bits
MAC	24 bits	top 24 bits of the 48 bit interface MAC. Vendor ID
ID	16 bits	Fixed value of FFFE inserted
MAC	24 bits	low 24 bits of the 48 bit interface MAC. Serial number.

- Example

  # Interface MAC
  00-40-63-ca-9a-20
  # IPv6 Interface ID (EUI-64)
  ::0040:63FF:FECA:9A20
  # or
  ::40:63FF:FECA:9A20
  # link local
  FE80::40:63FF:FECA:9A20

I**Pv6 Multicast Address Format**

- The Multicast format (which also replaces broadcast in IPv4) is defined by RFC 3513 Section 2.7. The format of a Multicast address is defined below:

Name	Bits	Size	Value	Description/Notes
Binary Prefix	0 - 7	8	1111 1111	Fixed value a.k.a the routing prefix
flags	8-11	4	000T	Where T may be either: 0 = "well-known" or permanently (IANA) assigned group 1 = "transient" group which has no IANA assignment
scope	12-15	4	-	May take one of the following assigned values: 0 - reserved 1 - interface-local scope 2 - link-local scope 3 - reserved 4 - admin-local scope 5 - site-local scope 6 - (unassigned) 7 - (unassigned) 8 - organization-local scope 9 - (unassigned) A - (unassigned) B - (unassigned) C - (unassigned) D - (unassigned) E - global scope F - reserved
Group ID	16 - 127	112	-	Uniquely assigned by IANA if "well-known" bit = 0 set in flags above. IANA IPv6 Multicast assignments

- The following lists some of the more common Multicast groups:

Address	Description/Notes
FF01::1	Interface local - all nodes
FF02::1	Link local - all nodes
FF01::2	Interface local - all routers
FF02::2	Link local - all routers

Writing IPv6 Applications

- The new IPv6 functions are useful only if the new protocol is widely supported by installed networks. IPv6 is now reaching maturity and most popular systems provide it as default in their standard distributions

- The Berkeley Socket API has been updated to implement several features that adapt the socket API for IPv6 support:

    - A new socket address structure to accommodate IPv6 addresses
    - New address conversion functions and several new socket options that are developed in RFC-3493.

- These extensions are designed to provide access to the basic IPv6 features required by TCP and UDP applications, including multicasting, while maintaining complete compatibility with existing IPv4 applications.

- Access to the more advanced features (raw sockets, header configuration, etc.) is addressed in RFC-3542.

- Within the transport module the use of the new API with extensions for IPv6 is mandatory, but for future compatibility it is important to design and implement protocol independent modules.

- For example, use the new ***getaddrinfo*** and ***getnameinfo*** system calls instead of the older ***gethostbyname*** and ***gethostbyaddr*** functions.

- Besides the application transport module, there are other IP dependencies that must be considered within the application design:

- **Presentation format for IP addresses**

    - IPv4: uses dot as separator a.b.c.d
    - IPv6: uses colon as separator x:x:x:x:x:x:x:x
    - When using Uniform Resource Locators (URLs), IP addresses should be enclosed in brackets, see RFC-2732.
    - The recommendation is using FQDN instead of presentation format of IP addresses.

- **IP address selection**

    - Choosing IP source/destination addresses pairs must observe the preferential rules outlined in RFC-3484.

- **Application Framing**

    - Some applications may prefer implementing their own fragmentation mechanisms instead of using IP layer fragmentation in order to achieve better performance.

    - The application fragment size is directly related to the PMTU and applications can implement their own PMTU-D in order to determine this value.

- **Storage of IP addresses**

    - Given the mew IPv6 addressing scheme and mechanism, IP addresses can change over the course of application execution time (typically after a renumbering process).

    - The same node can reach a destination using different source IP addresses, all of them configured in the source network interface.

- Dual stack mechanisms do not, by themselves, solve the IPv4 and IPv6 interworking problems.

- Protocol translation is required several times, for instance when an IPv6 application has to communicate with an IPv4 application to exchange IPv6 packets.

- Translation refers to the direct translation of protocols, including headers and sometimes the protocol payload.

- Protocol translation often results in loss of available (new) features. For instance, translation of IPv6 header into an IPv4 header will lead to the loss of the IPv6 flow label.

- In other cases, tunneling can be used, which is used to bridge compatible networks across incompatible ones.

Socket Address Structures

- IPv6 sockets use the new **sockaddr_in6** structure, in conjunction with a new address family, **AF_INET6**:

```
struct in6_addr {
    union {
        uint8_t u6_addr8[16];
        uint16_t u6_addr16[8];
        uint32_t u6_addr32[4];
 } in6_u;

    #define s6_addr              in6_u.u6_addr8
    #define s6_addr16            in6_u.u6_addr16
    #define s6_addr32            in6_u.u6_addr32
};

struct sockaddr_in6 {
    sa_family_t sin6_family;    /* AF_INET6 */
    in_port_t sin6_port;        /* Transport layer port # */
    uint32_t sin6_flowinfo;     /* IPv6 flow information */
    struct in6_addr sin6_addr;  /* IPv6 address */
    uint32_t sin6_scope_id;     /* IPv6 scope-id */
};
```

- The **sockaddr_in** or **sockaddr_in6** structures are utilized when using IPv4 or IPv6 respectively.

- Existing applications are written assuming IPv4, using the sockaddr_in structure. These can be easily ported by changing this structure to sockaddr_in6.

- However, in the interests of writing portable code, it is preferable to eliminate protocol version dependencies in the code.

- The API now provides a new data structure, **sockaddr_storage**, which is large enough to store all supported protocol-specific address structures.

```
/* Structure large enough to hold any socket address (with the
historical exception of AF_UNIX). 128 bytes reserved.  */

#if ULONG_MAX > 0xffffffff
# define __ss_aligntype __uint64_t
#else
# define __ss_aligntype __uint32_t
#endif
#define _SS_SIZE        128
#define _SS_PADSIZE     (_SS_SIZE - (2 * sizeof (__ss_aligntype)))

struct sockaddr_storage
{
   sa_family_t ss_family;      /* Address family */
   __ss_aligntype __ss_align;  /* Force desired alignment.  */
   char __ss_padding[_SS_PADSIZE];
};
```

- In essence, this new structure hides the specific socket address structure that a particular application is using.

Socket functions

- When porting IPv4 code to IPv6, there are three main areas where the API calls have been modified:
    - Creating a socket.
    - Passing the socket address structure from application to kernel.
    - Passing the socket address structure from kernel to application.

Creating a socket

- The difference between creating an IPv4 and an IPv6 socket is the value of the family argument in the socket call:

```
socket(PF_INET6, SOCK_STREAM, 0); /* TCP socket */
socket(PF_INET6, SOCK_DGRAM,  0); /* UDP socket */
```

Passing the socket address structure from application to kernel

- The socket address structure is filled before calling the socket function:

```
struct sockaddr_in6 addr;
socklen_t           addrlen = sizeof(addr);

/*
   fill addr structure using an IPv6 address before calling
   socket function
*/

bind(sockfd,(struct sockaddr *)&addr, addrlen);
```

- For portability and version independence we can do the following:

```
struct sockaddr_storage addr;
socklen_t               addrlen;

/*
   fill addr structure using an IPv4/IPv6 address and
   fill addrlen before calling socket function
*/

bind(sockfd,(struct sockaddr *)&addr, addrlen);
```

Passing the socket address structure from kernel to application

- When using this class of socket functions, the socket address structure is filled in with the address of the source entity:

```
struct sockaddr_in6 addr;
socklen_t           addrlen = sizeof(addr);

accept(sockfd,(struct sockaddr *)&addr, &addrlen);

/*
   addr structure contains an IPv6 address
*/
```

- For portability and version independence we can do the following:

```
struct sockaddr_storage addr;
socklen_t               addrlen = sizeof(addr);

accept(sockfd,(struct sockaddr *)&addr, &addrlen);

/*
   addr structure contains an IPv4/IPv6 address
   addrlen contains the size of the addr structure returned
 */
```

Address conversion functions

- The address conversion functions convert between the internal binary and text string representations.

- Binary representation is the network byte ordered binary value which is stored in the socket address structure. The text string representation is an ASCII string.

- The new address conversion functions that work with both IPv4 and IPv6 addresses are as follow:

```
struct sockaddr_in6 addr;
char straddr[INET6_ADDRSTRLEN];

memset(&addr, 0, sizeof(addr));
addr.sin6_family = AF_INET6;        // family
addr.sin6_port = htons(MYPORT);    // port, networt byte order

/*
   from ASCII to binary representation
*/
inet_pton(AF_INET6, "2001:720:1500:1::a100",
        &(addr.sin6_addr));

/*
   from binary representation to ASCII
*/
inet_ntop(AF_INET6, &addr.sin6_addr, straddr,
        sizeof(straddr));
```

- However, the use of these functions should be avoided, in favor of the newer calls instead: ***getnameinfo*** and ***getaddrinfo***.

- The code examples provided illustrate the use of the above API calls.

- The inet_* functions work with IPv4 and IPv6, however they are protocol dependent, which means you need to specify which kind of resolution should be made, v4 or v6.

- Also note that these inet_* functions do not work with scoped addresses, whereas getaddrinfo and getnameinfo do.

Resolving Names

- The ***getaddrinfo*** function returns a linked list of **addrinfo** structures that contain the information requested for a specific set of hostname, service and other additional information, in an **addrinfo** structure.

```
struct addrinfo {
   int     ai_flags;          /* AI_PASSIVE, AI_CANONNAME */
   int     ai_family;         /* AF_UNSPEC, AF_INET, AF_INET6 */
   int     ai_socktype;       /* SOCK_STREAM, SOCK_DGRAM ... */
   int     ai_protocol;       /* IPPROTO_IP, IPPROTO_IPV6 */
   size_t  ai_addrlen;        /* length of ai_addr */
   struct sockaddr ai_addr;   /* socket address structure */
   char   ai_canonname;       /* cannonical name */
   struct addrinfo ai_next;   /* next addrinfo structure */
};


/* function to get socket address structures */

int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);

   /* Note: Check the error using !=0 instead of <0    */
   /*       On FreeBSD, a failing resolve because the  */
   /*       local nameserver is not reachable or times */
   /*       out, it will result >0 which must be also  */
   /*       considered as an error.                    */
```

- The **getaddrinfo** function allocates a set of resources for the returned linked list. The **freeaddrinfo** function frees these resources.

  // function to free the resources allocated by getaddrinfo
  void freeaddrinfo(struct addrinfo *res);

- The following example illustrates the use of both the getaddrinfo and freeaddrinfo functions:

```
error = getaddrinfo(hostname, service, &hints, &res);

/*
   Try open socket with each address getaddrinfo returned,
   until getting a valid socket.
*/

if (error !=0 ) {
   /* Note: Check the error using !=0 instead of <0    */
   /*       On FreeBSD, a failing resolve because the  */
   /*       local nameserver is not reachable or times */
   /*       out, it will result >0 which must be also  */
   /*       considered as an error.                    */

   /* handle getaddrinfo error and return  */
}

resave = res;

for (; res; res = res->ai_next ) {
   sockfd = socket(res->ai_family,
                   res->ai_socktype,
                   res->ai_protocol);

   if ((sockfd < 0)) {
       switch errno {
           case EAFNOSUPPORT:
           case EPROTONOSUPPORT:
              /* last address family is not supported,
                 continue with the next address returned
                 by getaddrinfo */
              continue;

           default:
              /* handle other socket errors */
              ;
       }
   } else {
     /* socket succesfully created */

     /* check now if bind()/connect(). If they return errors,
        continue the while loop with the next address returned
        by getaddrinfo. If they don't return errors, break */

     /* ... */
   }
}

freeaddrinfo(ressave);
```

- The ***getnameinfo*** function provides from a socket address structure the address and service as character strings.

```
char clienthost   [NI_MAXHOST];
char clientservice[NI_MAXSERV];

/* ... */

/* listenfd is a server socket descriptor waiting connections
   from clients
*/

connfd = accept(listenfd,
                (struct sockaddr *)&clientaddr,
                &addrlen);

if (connfd < 0) {
   /* handle connect error and return */
}

error = getnameinfo((struct sockaddr *)&clientaddr, addrlen,
                    clienthost, sizeof(clienthost),
                    clientservice, sizeof(clientservice),
                    NI_NUMERICHOST);

if (error != 0) {
   /* handle getnameinfo error and return */
}

printf("Received request from host=[%s] port=[%s]\n",
       clienthost, clientservice);
```

- Typically applications are not aware of the version of IP they are using. Hence, applications should attempt to establish connections using each address returned by resolver until the attempt is successful.

- However, applications may exhibit different behavior depending on the address version used: IPv4, IPv6, IPv4-compatible-IPv6 or IPv4-mapped, etc. addresses.

- The following macros can be used to help applications test the type of addresses they are using:

int IN6_IS_ADDR_UNSPECIFIED (const struct in6_addr *);

int IN6_IS_ADDR_LOOPBACK (const struct in6_addr *);

int IN6_IS_ADDR_MULTICAST (const struct in6_addr *);

int IN6_IS_ADDR_LINKLOCAL (const struct in6_addr *);

int IN6_IS_ADDR_SITELOCAL (const struct in6_addr *);

int IN6_IS_ADDR_V4MAPPED (const struct in6_addr *);

int IN6_IS_ADDR_V4COMPAT (const struct in6_addr *);

int IN6_IS_ADDR_MC_NODELOCAL (const struct in6_addr *);

int IN6_IS_ADDR_MC_LINKLOCAL (const struct in6_addr *);

int IN6_IS_ADDR_MC_SITELOCAL (const struct in6_addr *);

int IN6_IS_ADDR_MC_ORGLOCAL (const struct in6_addr *);

int IN6_IS_ADDR_MC_GLOBAL (const struct in6_addr *);