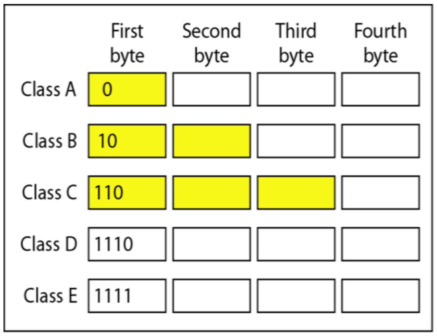
Internet Layer

The Internet layer accepts all input from the Transport layer as data. To the data, it adds a header which includes the IP address of the source and destination computer. The Internet layer is responsible for routing of the datagram from source to destination, deciding the best path among multiple paths. The path chosen to send the datagram is called the route.

At the present time, the TCP/IP protocol stack is running parallel IP protocols: IPv4 and IPv6. All new Internet devices are mandated to run a “dual protocol” stack to ensure backward compatibility with IPv4. The dual stack is called IPv4/v6; this dual approach will allow IPv4 devices to communicate with each other and IPv6 devices to communicate will IPv6 hosts. The goal of the IEFT is a smooth transition from IPv4 to IPv6. IPv6 was commenced on the Internet June 6, 2012. There has been a 500% increase in its use since then, mostly in the developing world and Asia. Why the change to IPv6 (also called IPng –next generation?). The short answer the IPv4 address space was depleted in 2015.

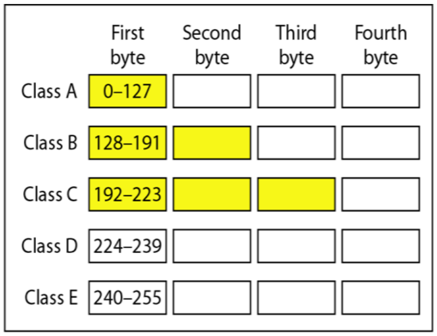
IPv4 Addressing

IPv4 is a classful addressing scheme and it has five classes as shown in the figure below. The first 3 classes A, B,C are used for one-to-one or unicast communication between source and destination. All internet communication is in general is unicast. Class D addresses are used for one to many known as multicast communication e.g. BellFibe and Rogers Ignite. The shaded yellow boxes in the figure below identifies the network prefix or the number of bits used for network ID in each class, where the unshaded part represents the suffix or host ID.



	First byte	Second byte	Third byte	Fourth byte
Class A	0			
Class B	10	🟨		
Class C	110	🟨	🟨	
Class D	1110			
Class E	1111			

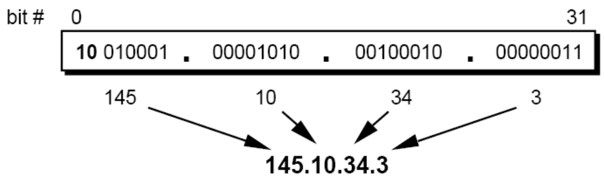
a. Binary notation

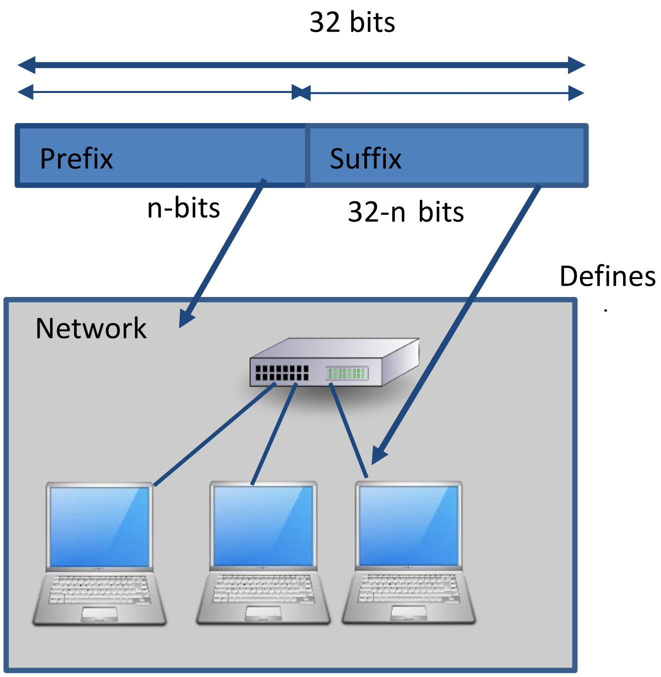


	First byte	Second byte	Third byte	Fourth byte
Class A	0–127			
Class B	128–191	🟨		
Class C	192–223	🟨	🟨	
Class D	224–239			
Class E	240–255			

b. Dotted-decimal notation

IPv4 addresses are 32-bit long, and written in dotted decimal notation. The total address space or the number of IPv4 addresses are **2^{32}=4.3** billion addresses





Hierarchy of Ipv4 Addressing

Subnet Mask

Hides, or "masks," the network part of a system's IP address and leaves only the host part as the machine identifier. It is used to identify the network segment to which packet belongs. If a machine’s IP address and subnet mask/network prefix is known, then the network can be identified by doing a logical AND operation.

Class	Binary	Dotted Decimal	Prefix/CIDR
A	**1111111.00000000.00000000.00000000**	**255.0.0.0**	**/8**
B	**11111111.11111111.00000000.00000000**	**255.255.0.0**	**/16**
C	**11111111.11111111.11111111.00000000**	**255.255.255.0**	**/24**

Default Mask of Classful Addressing

IPv4 and Changes to Preserve the Address Space

The IPv4 address space, which we are all familiar with like, 192.168.0.1, is based on 4 octets, or 32 bit address. Each octet has a range of values from 0-255; thus, the maximum size of the address space is 256 X

256 X 256 X 256 or 4.3 billion addresses. The 32 bits can be divided into different classes to allocate network and host addresses. With the development of the World Wide Web, the growth of PCs, the use of smartphones, tablets, gaming systems, and VoIP systems, far more IP addresses were necessary than the founders envisaged. The new address space IPv6 uses 128-bit addresses and is capable of 340 trillion, trillion, trillion addresses. How big is this number? If you assigned an IPv6 address to every atom on Earth, you would still have enough addresses to do another 100 Earths. It is not even remotely a possibility that we will run out of IPv6 addresses, before humans are extinct.

The depletion of IPv4 addresses was predicted in 1993 and steps where take to preserve the space as long as possible. The IETF instituted the following changes:

- Private address spaces where created to allow networks to create LAN addresses with approval of the Internet registry. However, hosts on the LAN can't communicate with the public Internet except through a proxy gateway.
- NAT, Network Address Translator, was created to act as a proxy gateway converting private host addresses to public addresses to access the Internet. Thus, many LAN hosts can share a single Internet address.
- DHCP, Dynamic Host Configuration Protocols was created to act as a server to allocate addresses from a pool of available LAN addresses. The address assigned to the host is "leased" from the server for a time and can be reallocated to another host when the lease expires. Thus, many hosts can share a pool of addresses.
- CIDR, Classless Inter-Domain Routing was a fundamental change in how IP addresses were assigned. The old class based system of allocating IPv4 addresses was discontinued for a classless system which used the 32 bit address space more efficiently, avoiding wasted IP addresses.

All of these network technologies we have heard before and are currently implemented on our home and Seneca networks. In North America, IPv4 will continue to be used for a long time. Networks won't convert to IPv6 until new hardware is required.

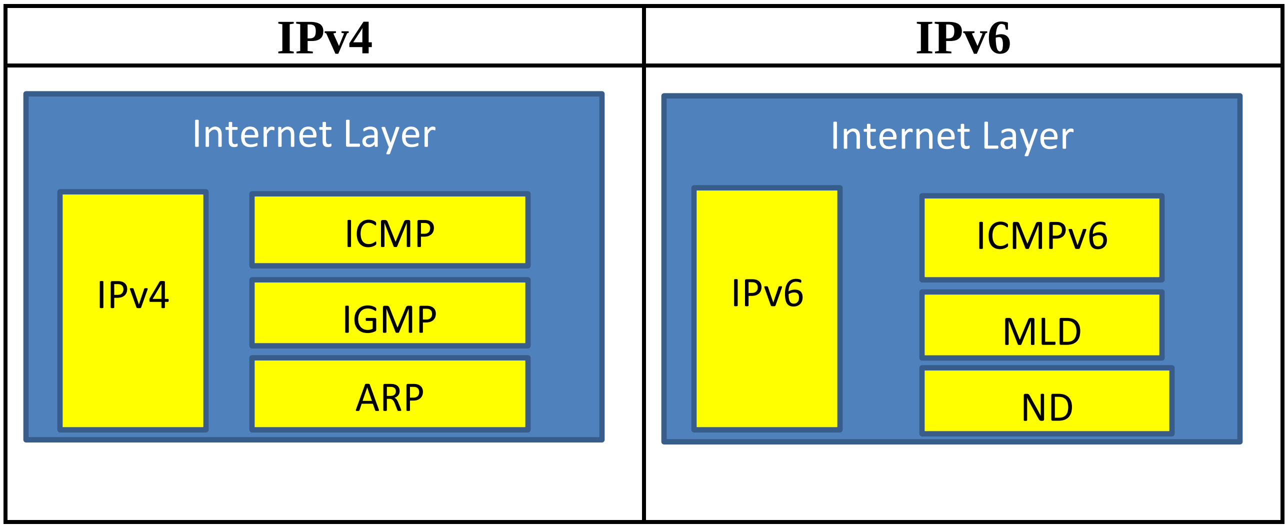
IPv6 addressing

IPv6 uses three types of addresses -- unicast, multicast, and anycast. Unicast and multicast addresses also existed in IPv4, but Anycast is a new type of address defined by IPv6.

1. **Unicast:** A unicast address is a one-to-one address. Packets send to a unicast address travel between two hosts on a single interface.
2. **Multicast:** A multicast address is a one-to-many address. Packets sent to a multicast address travel to all interfaces identified by the multicast group address (this replaces the broadcast address used in IPv4).
3. **Anycast:** A anycast address is a one-to-one address sent to the nearest host. Packets sent to an anycast address are sent to a single interface of the nearest host identified by the address.

Both IPv4 and IPv6 are connectionless protocols. This means that every data packet that is sent across the network is treated as an independent unit. IP does not maintain the details of the connection between the server and the client and does not guarantee reliable delivery. It is a "best effort" delivery system, trying to avert data loss as much as possible. If the host is located on the same network, this is called "local delivery" and if the host is on a different network, it is called "remote delivery". With remote delivery, the data will be handled by intermediate devices called routers which will forward the data based on the destination address. The sender is unaware of how the data is transmitted to the destination. This is similar to when you mail a letter. You address the letter and you are ignorant of the number of intermediate steps Canada Post carries out to forward the letter to its destination.

IP works with ICMP, (Internet Control Message Protocol) which is responsible for generating error messages when an error occurs during data transmission. The Transport layer protocols, namely TCP and UDP, decide to retransmit data based on the error message that is received. The table below is a comparison of the IPv4 and IPv6 layers and the protocol specifications of each layer.



When you ping a host. The message is an ICMP echo request packet which the receiving host returns an ICMP echo reply packet, or "host unreachable" if the host is offline. IGMP, Internet Group Management Protocol is used for grouping multicast hosts. ARP, Address Message Protocol, is used by IP to resolve MAC addresses of a host, when the IP address is known. An ARP request packet is broadcast to all hosts, saying "Whoever has this IP address, please send me your MAC address". The receiving host, who has that particular IP address, then returns a unicast ARP reply to the sending host, so that the data link layer frame can be completed. IPv6 has new protocols but serve the same function. ICMPv6 is the error message protocol. MLD, Multicast Listener Discovery is an improved version of IGMP used for multicasting and ND, Neighbor Discovery replaces ARP. With IPv6, all devices will be connected to the public network, which will make it easier for users to manage home automation, file sharing, online gaming, etc. without complex settings on their routers.

The dual stack approach to IPv6 means that IPv4 will be around for a long time. IPv4 devices will continue to work in the foreseeable future. However, there are 3 reasons why businesses should plan to change to IPv6.

- **Inevitability**: At some point in the future, IPv4 will be no longer supported. Moving to IPv6 when hardware needs to be changed is a good approach
- **Efficiency:** IPv6 is a better protocol than IPv4, with faster routing by removing the need to check packet integrity and fragment a packet. In IPv6, only the sending host performs fragmentation. If an

IPv6 router cannot forward a packet because it is too large, the router sends an ICMPv6 Packet Too Big message to the sending host and discards the packet. NAT will no longer be needed, because each host will have a unique IP address on the public network, unless the network continues to use private addressing.

- **Security:** IPv6 has been built from the ground up with security in mind with built-in encryption.IPv6 encrypts traffic and checks packet integrity to provide VPN-like protection for standard Internet traffic.

IPv6 Address Space

Network Bits 48 bit assigned	Prefix Bits 16 bits self-assigned	Host Bits 64 bits

IPv6 has a network address of 48 bits which is assigned by the Internet Society. Adjacent to the network bits are network prefix bits which businesses can use to divide their address space to match organizational needs. These bits use CIDR notation to determine which bits have fixed values and which represent the subnet identifier. The remaining 64 bits are host bits.

For IPv6, the 128-bit address is divided along 16-bit boundaries. Each 16-bit block is converted to a 4-digit hexadecimal number and separated by colons. The resulting representation is known as colon-hexadecimal.

The following is an IPv6 address in binary form:

00100001110110100000000011010011000000000000000000101111001110110000001010101010000000001111111111111110001010001001110001011010

The 128-bit address is divided along 16-bit boundaries:

0010000111011010 0000000011010011 0000000000000000 0010111100111011 0000001010101010 0000000011111111 1111111000101000 1001110001011010

Each 16-bit block is converted to hexadecimal and delimited with colons. The result is:

21DA:00D3:0000:2F3B:02AA:00FF:FE28:9C5A

IPv6 representation can be further simplified by removing the leading zeros within each 16-bit block. However, each block must have at least a single digit. With leading zero suppression, the address representation becomes:

21DA:D3:0:2F3B:2AA:FF:FE28:9C5A

Compressing Zeros

Some types of addresses contain long sequences of zeros. To further simplify the representation of IPv6 addresses, a contiguous sequence of 16-bit blocks set to 0 in the colon-hexadecimal format can be compressed to "::," known as double-colon.

For example, the link-local address of **FE80:0:0:0:2AA:FF:FE9A:4CA2** can be compressed to **FE80::2AA:FF:FE9A:4CA2**. The multicast address **FF02:30:0:0:0:0:0:2** can be compressed to **FF02:30::2**

Zero compression can be used to compress only a single contiguous series of 16-bit blocks expressed in colon-hexadecimal notation. You cannot use zero compression to include part of a 16-bit block. For example, you cannot express **FF02:30:0:0:0:0:0:5 as FF02:3::5.**

To determine how many 0 bits are represented by the double colon, you can count the number of blocks in the compressed address, subtract this number from 8, and then multiply the result by 16. For example, the address FF02::2 has two blocks (the "FF02" block and the "2" block.) The number of 0 bits expressed by the double colon is 96 (96 = (8 – 2)×16).

Zero compression can be used only once in a given address. Otherwise, you could not determine the number of 0 bits represented by each double colon.

Source: https://technet.microsoft.com/en-us/library/cc784831%28v=ws.10%29.aspx

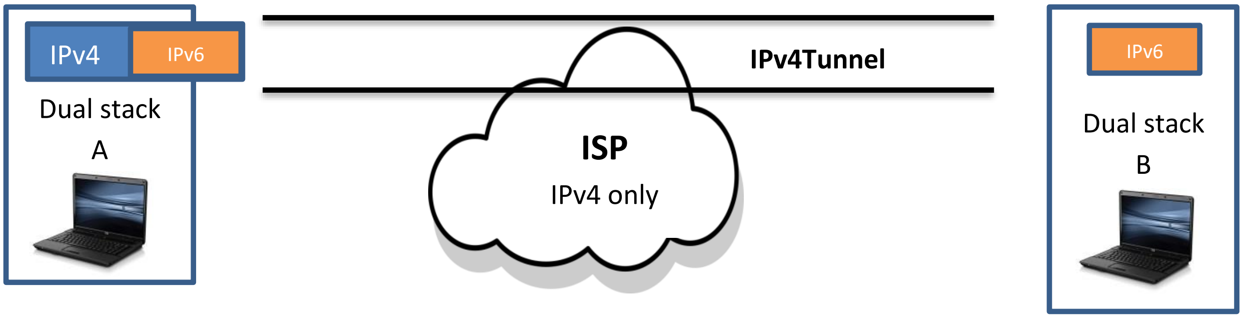
IPv6 Prefix and Dual environments

The 128 bits of the IPv6 address must denote the network address and the host address for packets to be correctly delivered. IPv6 uses CIDR notation to denote the number of bits used for the network. Within a network all network hosts will have the same network prefix. The prefix is indicated as a "/" followed by the number of bits. For example, **2001:cdba:9abc:5678::/64** denotes the network address **2001:cdba:9abc:5678**. Thus the host addresses range from **2001:cdba:9abc:5678:0000:0000:0000:0000 up to 2001:cdba:9abc:5678:ffff:ffff:ffff:ffff**

For dual environments a special notation is permitted, using the syntax **x:x:x:x:x:x:d.d.d.d** where the "x"s are the hexadecimal values of the six high-order 16-bit pieces of the address, and the "d"s are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation) with a prefix length of 96. For example, the IPv4 address 126.101.64.1 can be represented as **::126.101.64.1/96** The exception to this rule is the 127.0.0.1 loop back address to test if TCP/IP is properly installed on the local interface. This address has an IPv6 representation of **::1/128**.

Tunnelling

The dual stack approach is an important feature to ensure compatibility between IPv4 and IPv6 hosts. However, it is possible that packets will arrive at routers that are not IPv6 enabled. To solve this problem a process can **IPv6 over IPv4 tunneling** has been defined. Suppose you are at work and have a dual stack host and you want to send a file to your home, which also has a dual stack. However, the upstream ISP is not IPv6 enabled. How can you send this file? The Internet layer of PC A will encapsulate the IPv6 address inside an IPv4 address, so the packet can travel over the ISP network and then PC B's Internet layer will de-encapsulate the address.



How would a router or host know which protocol to use? This would be determined by the DNS records. Users communicate on the Internet using fully qualified domain names (FQDN), not IP addresses. DNS records map the IP address to the domain name. Thus, the transition from IPv4 to IPv6, from a user's point of view will be invisible. For instance, assume that we have a dual stack host, and we want to access the URL https://www.senecacollege.ca. A dual stack node will behave as follows:

1. If www.senecacollege.ca resolves to an IPv4 address, connect to the IPv4 address. In such a case, the DNS database record for www.senecacollege.ca will be as follows:

   www.senecacollege.ca. IN A 142.204.0.0

2. If www.senecacollege.ca resolves to an IPv6 address, connect to the IPv6 address.

   www.senecacollege.ca. IN AAAA FE80:DC28:ffff::1234

3. If www.senecacollege.ca resolves to multiple IPv4/v6 addresses, IPv6 addresses will be tried first, and then IPv4 addresses will be tried. For example, with the following DNS records, we will try connecting to **FE80:DC28:ffff::1234**, then

   www.senecacollege.ca IN AAAA FE80:DC28:ffff::1234
   www.senecacollege.ca IN AAAA FE80:DC28:ffff::5678
   www.senecacollege.ca IN A 142.204.0.0

Since we assume that IPv6 hosts will be able to use IPv4 as well, the Internet will be filled with IPv4/v6 dual stack devices for the near future, until the use of IPv6 becomes dominant

IPv6: A Programmer's Perspective

From a programmer's perspective IPv6 is very different that IPv4 programming and will present some new challenges.

1. **User Interface Design:** Unlike the IPv4 address space, the IPv6 address space is much larger and must be displayed in hexadecimal notation. An advantage of dotted decimal notation was that it was very predictable when displaying addresses. IPv6 on the other hand, with its truncated notation

of using double colons to represent a series of zeros is less predictable. Lastly, GUI application Users will normally use DNS names to identify hosts not IP addresses. Thus, GUI applications which supply text boxes, such as TCP/IP properties, would be unnecessary and should be avoided given the complexity of IPv6 format over IPv4. However, applications used by administrators would need such text boxes.

2. **IP Family Independent:** With a dual stack environment it is important not to write code that is family specific, such as IPv4 or IPv6. Family specific code can't be handled correctly in a dual stack environment. The best approach is to use data structures and functions that are family independent.
3. **Determine IP Family Before Creating Socket:** When creating a socket, the common procedure in IPv4 was to create the socket, and bind the address family information to the socket. This procedure will not work in IPv6. The address family must be determined first, then create the socket and bind the family to it. In IPv4, a node normally has a single IPv4 address associated with it. In IPv6, it is normal to have multiple IP addresses onto a single node. More specifically, IPv6 addresses are assigned to interfaces, not to nodes. An interface can have multiple IPv6 addresses.

1. User Interface Design

IPv4 addresses provided a rigid format: **10.0.10.1**. IPv6 addresses are less rigid because of the convention of using a double colon to provide a truncated address, such **1004:0:0:0:0:0:0:10**, can be written as **1004::1**. This requires programmers to take this capacity into account when creating user interfaces which display the IPv6 address. Also, the text box must be capable of supporting the IPv6 address with the embedded IPv4 address, such as **0000:0000:0000:0000:0000:0000:ffff:10.0.10.1.** And if the scope identifier is added to the address the length could be increased by another eleven characters.

Addressing in IPv6, due to many factors such as length, complexity, and the significance of sections within the IPv6 address space, is not conducive to modification or specification by users. Therefore, the need to provide users with the capability of specifying their own address is reduced. Additionally, due to the complexity associated with IPv6 addressing, providing administrators with the capability of specifying IPv6 address information is not likely to occur on a per-node basis.

Displaying an IPv6 address in the UI is not inconceivable, and therefore developers should consider the variability in the size of an IPv6 address when modifying an application to support IPv6.

The rest of this section discusses the difference between IPv4 address length predictability and IPv6 address length considerations. This section presumes IPv6 addresses are being displayed in their hexadecimal representation.

IPv4 addresses are predictable in size, because they rigidly follow dotted decimal notation, as the following address example illustrates:

10.10.256.1

With the complexities of the IPv6 format and the difficulty in associating it with a host, it is more likely that the user will rely on name based notation rather than number based notation. It is predicted that with IPv6 DNS will play a greater role in dynamically allocating addresses. In which case, a text box in the user interface may not be necessary (unless the application is an administrative program). Here are the following topics programmers need to consider:

- Should number based or named base notation be used?
- Should the truncated addresses be used in the interface? The double colon is an optional method of notation to simplify the address, not a specification.
- Does the user need specific parts of the address, such as the subnet prefix, scope identifier or other subfields?

Lastly, if the user is to enter an IPv6 address as part of the URL, the address must be enclosed in square brackets to avoid ambiguity with the port number which is also separated by a colon. For example: **http://[F380:DC28:ffff::1]:80/64**..

2. IP Family Independent

A review of the code below appears to address the needs of making socket connections in a dual stack environment.

```
struct sockaddr *sa;
/*
* you cannot support other address families with this code
*/
switch (sa->sa_family) {
case AF_INET:
port = ntohs(((struct sockaddr_in *)sa)->sin_port);
break;
case AF_INET6:
port = ntohs(((struct sockaddr_in6 *)sa)->sin6_port);
break;
default:
fprintf(stderr, “unsupported address family\n”);
exit(1);
/*NOTREACHED*/
}
```

We use a simple switch statement to determine if the address family and handle it in dotted decimal notation or IPv6 and handle it in hexadecimal notation. Unfortunately, the answer is not as simple. Code like the above must be avoided for the following reasons:

1. Hardcoding the address family must be avoided because the application will not function unless the operating system supports the address family. For the near future, there will be devices that are IPv4 specific, IPv6 specific, or IPv6/v4 dual stack. Hardcoding the address will make interoperability difficult.
2. If a new protocol is developed applications which are family independent will not need to be written. On the other hand, a family dependent application will need to be changed.

3. A device that is IPv4 specific will not know how to tunnel a IPv6 address inside of an IPv4 packet. Interoperability will be increased if family independent structures and functions are used.

To ensure IPv6 interoperability, review the IPv4 code base looking for inappropriate controls. If checking UNIX applications use grep to check for any family specific functions listed below. Windows has provided a tool called Checkv4.exe available in the Microsoft SDK or a web download. When you run your code through the utility it alerts you of any IPv4 specific data structures and function calls. Alternatively, you can search your code base for instances of the **sockaddr** and **sockaddr_in** structures and change all such usage (and other associated code, as appropriate) to the **SOCKADDR_STORAGE** structure.

Family Specific APIs to be Avoided

The following APIs that take struct in_addr or struct in6_addr, should be avoided.

inet_addr,
inet_aton,
inet_lnaof,
inet_makeaddr,
inet_netof,
inet_network,
inet_ntoa,
inet_ntop,
inet_pton,
gethostbyname,
gethostbyname2,
gethostbyaddr,
getservbyname,
getservbyport

There are also new agnostic function calls for Windows Sockets; these include the following:

- **WSAConnectByName** -- to establish a connection to an endpoint given a host name and port. The WSAConnectByName function is available on Windows Vista and later.
- **WSAConnectByList --** to establish a connection to one out of a collection of possible endpoints represented by a set of destination addresses (host names and ports). The WSAConnectByList function is available on Windows Vista and later.
- **getaddrinfo** family of functions (**getaddrinfo**, **GetAddrInfoEx**, **GetAddrInfoW**, **freeaddrinfo**, **FreeAddrInfoEx**, **FreeAddrInfoW**, and **SetAddrInfoEx**) to replace gethostbyname function calls which are IP family specific
- **getnameinfo** family of functions (**getnameinfo** and **GetNameInfoW**)[i]

3. Determine IP Family Before Creating Socket

To create a socket, the socket family, type of socket and protocol type need to be specified. This could be written as follows:

```
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

AF_INET identifies a IPv4 address space, SOCK_STREAM indicates a reliable protocol and IPPROTO_TCP specifies the TCP protocol (SOCK_STREAM and IPPROTO_TCP are used together).
To create a socket in IPv6 is very similar:

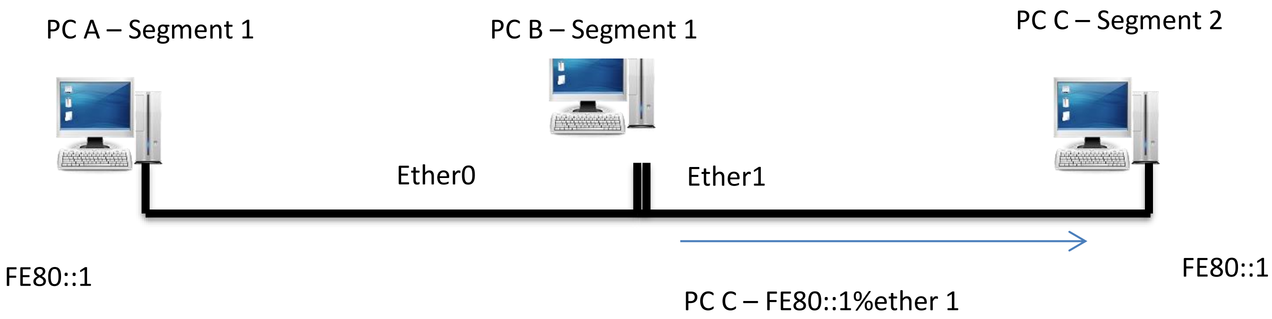
```
s = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
```

IPv6 uses a socket API with a C structure called sockaddr_in6.   You can see that the structures are very similar except sock_addr_in6 will store a 128 bit address.

To support IPv4/v6 dual stack, we need to handle both IPv4 and IPv6 addresses. This can be done using the sockaddrs data structure. With sockaddrs, the data contains the identification of address family, so we can pass around the address data using pointers and let the called function handle it.

```
extern int foo(struct sockaddr *);
int
main(argc, argv)
int argc;
char **argv;
{
struct sockaddr_in sin;
/* setup sin */
foo((struct sockaddr *)&sin);
}
int
foo(sa)
struct sockaddr *sa;
{
switch (sa->sa_family) {
case AF_INET:
case AF_INET6:
/* do something */
return 0;
default:
return -1; /*not supported*/
}
}[ii]
```

Another important reason for using sockaddr, is due to the scoped IPv6 addresses. Unlike IPv4 where the address was tied to a host, the IPv6 address is tied to an interface and a host can have several interfaces. Thus the 128 bits does not uniquely identify the peer. In diagram below, from PC B, we can see two hosts with fe80::1: one on Ethernet segment 1, another on Ethernet segment 2. To communicate with host A or C, PC B has to disambiguate between them with a link-local address—specifying the outgoing interface to use, called the link-local address --a 128-bit address is not enough. Sockaddr_in6 has a member named sin6_scope_id to disambiguate destinations between multiple scope zones. String representation of a scoped IPv6 address is augmented with a scope identifier after the % sign, such as fe80::1%ether1.



Determine IP Family Before Creating Socket:

In IPv4, socket programming the getaddrinfo and the gethostbyname are commonly used after the socket is created using the AF_INET address family. On dual stack hosts, however, this approach will not work because the address family of the remote host name is not known. So address resolution using the getaddrinfo function must be completed first to determine the IP address and the address family of the remote host. Only then, can the socket function be called to open a socket of the address family returned by getaddrinfo. This is a fundamental change in how socket applications are written, since many IPv4 applications tend to use a different order of function calls. Also, if the name resolution returns both IPv4 and IPv6 addresses, then separate IPv4 and IPv6 sockets must be used to connect to these destination address.

If writing a Windows socket application these complexities can be avoided using the new WSAConnectByName function (available on Windows Vista and later)

The following code example shows the proper sequence for performing name resolution first (performed in the fourth line in the following source code example), then opening a socket (performed in the 19th line in the following code example).

C++

```
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

SOCKET OpenAndConnect(char *Server, char *PortName, int Family, int SocketType)
{

    int iResult = 0;
    SOCKET ConnSocket = INVALID_SOCKET;

    ADDRINFO *AddrInfo = NULL;
    ADDRINFO *AI = NULL;
    ADDRINFO Hints;

    char *AddrName = NULL;

    memset(&Hints, 0, sizeof (Hints));
    Hints.ai_family = Family;
    Hints.ai_socktype = SocketType;
```

```
  iResult = getaddrinfo(Server, PortName, &Hints, &AddrInfo);
  if (iResult != 0) {
    printf("Cannot resolve address [%s] and port [%s], error %d: %s\n",
        Server, PortName, WSAGetLastError(), gai_strerror(iResult));
    return INVALID_SOCKET;
  }
  //
  // Try each address getaddrinfo returned, until we find one to which
  // we can successfully connect.
  //
  for (AI = AddrInfo; AI != NULL; AI = AI->ai_next) {

    // Open a socket with the correct address family for this address.
    ConnSocket = socket(AI->ai_family, AI->ai_socktype, AI->ai_protocol);
    if (ConnSocket == INVALID_SOCKET) {
      printf("Error Opening socket, error %d\n", WSAGetLastError());
      continue;
    }
    //
    // Notice that nothing in this code is specific to whether we
    // are using UDP or TCP.
    //
    // When connect() is called on a datagram socket, it does not
    // actually establish the connection as a stream (TCP) socket
    // would. Instead, TCP/IP establishes the remote half of the
    // (LocalIPAddress, LocalPort, RemoteIP, RemotePort) mapping.
    // This enables us to use send() and recv() on datagram sockets,
    // instead of recvfrom() and sendto().
    //

    printf("Attempting to connect to: %s\n", Server ? Server : "localhost");
    if (connect(ConnSocket, AI->ai_addr, (int) AI->ai_addrlen) != SOCKET_ERROR)
      break;

    if (getnameinfo(AI->ai_addr, (socklen_t) AI->ai_addrlen, AddrName,
            sizeof (AddrName), NULL, 0, NI_NUMERICHOST) != 0) {
      strcpy_s(AddrName, sizeof (AddrName), "<unknown>");
      printf("connect() to %s failed with error %d\n", AddrName, WSAGetLastError());
      closesocket(ConnSocket);
      ConnSocket = INVALID_SOCKET;
    }
  }
  return ConnSocket;
}
```

[i] "IPv6 Guide for Windows Socket Programming", at https://msdn.microsoft.com/en-us/library/windows/desktop/ms738649(v=vs.85).aspx

[ii] Ibid,pp17-18