This article discusses the concepts behind a simple IPv6 program -- specifically, how IPv6 solves the problems of address space and large routing tables. A programmer familiar with IPv4 will be able to recognise an IPv6 address and configure one for his machine. The article also covers tunneling, mapped addresses, and porting IPv4 to IPv6 applications, as well as the logic of enabling an IPv4 client to handle IPv6 addresses.

In today's networking world, IPv4 is the foundation of networking, but in the last 10 years, questions have come up due to:

- Fear of running out of IPv4 address space as soon as 2002
- Fear of running out of capacity in global routing tables

Network Address Translation (NAT) and Classless Inter-Domain Routing (CIDR) (see Relevant concepts) have been used as stopgap measures for these simple but serious problems. IPv6 -- also called IPng (IP new generation) -- has been viewed as the long-term solution.

The following enhancements to IPv4 have also been planned:

- Simplified header processing
- Support for extended options
- Enhancements like quality-of-service capabilities, authentication and privacy capabilities, flow control capabilities, and autoconfiguration

The key rule behind all this change is that IPv6 applications should continue to live with IPv4 applications. The bottom line is that IPv6 should support a mixed IPv6 and IPv4 environment.

This article will help you quickly understand the concepts behind IPv6 and write a simple program in it. Let's start with IPv6 addressing.
IPv6 addressing

Types of addresses in IPv6: Anycast is the new baby

In IPv6, there are three types of addresses -- unicast, multicast, and anycast. We had unicast addresses in IPv4, and many systems support multicast, as well. Anycast is a new type of address defined by IPv6.

1. **Unicast**: This is like any normal IP address on a single interface (for example, the IPv4 address 9.185.101.1 on en0).
2. **Multicast**: A packet sent to a multicast address is delivered to all interfaces identified by that address. There is no broadcast address type since the multicast type can take care of it.
3. **Anycast**: A packet sent to an anycast address is delivered to one of the interfaces identified by that address (the "nearest" one, according to the routing protocols' measure of distance). Let's consider a situation where anycast addresses can be used -- connecting to a service provider's router. Assume that the service provider can give you a set of addresses to connect to, and you choose one of these addresses. With IPv6, the service provider can give you an anycast address, which you will use to automatically connect to the "nearest" address. This is a new feature in IPv6 and there is still a lot of debate going on about its implementation.

How IPv6 addresses are written: What a change

There are three conventional forms for representing IPv6 addresses as text strings:

1. **The primary form**: The preferred form is x:x:x:x:x:x:x, where the "x"s are the hexadecimal value of the eight 16-bit pieces of the address. Two examples:

   fe80:0:0:0:207:30ee:edcb:d05d
   1080:0:0:0:1:700:200B:417C

   There are eight hex fields in the first address:

   1. fe80
   2. 0
   3. 0
   4. 0
   5. 207
   6. 30ee
   7. edcb
   8. d05d

   In IPv6, we do not write the leading zeros in a field. That is, the second field above is just written as "0" rather than "0000." Note that there are 4 hex digits in each field. Each hex digit is 4 bits (and can represent a hex value of 0-F). This means that there are 16 bits in each field (4 hex digits x 4 bits per digit). The total size of an IPv6 address is 128 bits (8 hex fields x 16 bits per field).

   2. **A different representation of the above address**: Due to some methods of allocating certain styles of IPv6 addresses, it is common for addresses to contain long strings of zero bits. In order to make it easier to write addresses containing zero bits, a special syntax is available to compress...
the zeros. The use of :: indicates multiple groups of 16-bits of zeros. The :: can only appear once in an address, and can also be used to compress the leading zeros in an address. For example:

- FF01::0:0:0:0:0:101 is a multicast address that can be written as FF01::101.
- 0::0:0:0:0:0:1 is a loopback address that can be written as ::1.

3. For dual environments: An alternative form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes is 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) -- that is, the first 96 bits are represented as 6- x 16-bit hex fields and the last 32 bits are 4- x 8-bit decimal digits. For example:

| ::9.184.201.1 |
| ::ffff:9.184.209.2 |

**IPv6 address prefix**

The IPv6 address prefix denotes the network part of an address and is represented by the notation ipv6-address/prefix-length.

Take this example:

```
fe80::206:29ff:fedc:e06e/64
```

In this instance, fe80::206:29ff:fedc:e06e is the address and 64 is the prefix length. These two together give us the address prefix. In the example, specifying 64 means that we take the first 64 bits of the above 128-bit address to identify the network part of the address.

**Relevant concepts**

*Network Address Translation (NAT):* An Internet standard that enables a local-area network (LAN) to use one set of IP addresses for internal traffic and a second set of addresses for external traffic. A NAT box located where the LAN meets the Internet makes all necessary IP address translations.

NAT serves three main purposes:

- Provides a type of firewall by hiding internal IP addresses
- Enables a company to use more internal IP addresses; since they're only used internally, there's no possibility of conflict with IP addresses used by other companies and organizations
- Allows a company to combine multiple ISDN connections into a single Internet connection

(See RFC 1631, "Hide & Seek with Gateways & Translators" in Resources.)

*Classless Inter-Domain Routing (CIDR):* Classless Inter-Domain Routing. A new IP addressing scheme that replaces the older system based on classes A, B, and C. With CIDR, a single IP address can be used to designate many unique IP addresses. A CIDR IP address looks like a normal IP address except that it ends with a slash followed by a number, called the IP prefix. For example: 172.200.0.0/16

The IP prefix specifies how many addresses are covered by the CIDR address, with lower numbers covering more addresses. An IP prefix of /12, for example, can be used to address...
4,096 former Class C addresses. CIDR addresses reduce the size of routing tables and make more IP addresses available within organizations.

(See Resources for RFC 1517,1518,1519,1520.)

This raises several questions:

1. How does the above representation solve the two primary problems we are trying to address:
   - The finite amount of available address space?
   - Large global routing tables?
2. How is the network identified in an IPv4 address?
3. Why should the prefix length be allowed to be specified in an IPv6 address?
4. How is the prefix specified in an IPv4 address?
5. What are the problems caused by this?

And here are the answers:

**Address space:** Regarding the address space question, Robert M Hinden, one of the key figures in IPv6 efforts, explains:

IPv6 supports addresses that are four times the number of bits as IPv4 addresses (128 vs. 32). This is 4 billion times 4 billion times 4 billion (2^96) times the size of the IPv4 address space (2^32). This works out to be:

340,282,366,920,938,463,463,374,607,431,768,211,456

This is an extremely large address space. In a theoretical sense this is approximately 665,570,793,348,866,943,898,599 addresses per square meter of the surface of the planet Earth (assuming the earth surface is 511,263,971,197,990 square meters).

**The class enemy:** Now let’s take up the questions regarding address prefix in IPv4 and IPv6. The division of IPv4 address space into Class A, B, C, and D networks has caused some problems. In IPv4, the network part was fixed by the class of the address. Let’s illustrate our point with an example. Class A addresses can support 16 million hosts on each of their 128 networks (because in a class A address, the highest-order bit is set to 0; the next 7 bits are used for the network part; and the remaining 24 bits are used for the local address). Now, if an organisation were given a Class A address, and it didn’t have 16 million hosts, then the remaining address space would go to waste. Also note that everyone cannot be given a Class A address as there are only 127. CIDR had to be introduced to solve this problem and prolong the life of IP. This means that the network part of an address should not be fixed. There is a clear need for an organisation-specific network size. This means that the network part of an address should not be fixed. This variable prefix length is implemented in IPv6 by allowing the user to specify the network bits in the address prefix. For example, in the address fe80::206:29ff:fedc:e06e/64 -, the numeral 64 denotes the network part, and this could be changed. Here we have the option of choosing the network part. This is flexible, unlike IPv4 where it has always been fixed.

**Routing tables:** The routes in the Internet grew in time. Backbone routers were approaching their limit in 1984. If CIDR were not introduced to solve the problem of space in global backbone routers, they would have just come to a halt.
**CIDR technique:** So how does IPv6 solve this problem? The technique for stopping this problem is to allow for address prefixes that fit specific organisational needs. This technique was basically introduced in CIDR. In IPv6 the prefix or the network part is also specified by a user-specified network prefix. This helps to aggregate a large number of IP addresses and specify a single route for the organisation. If an organisation has many networks, then in the case of IPv4, many network prefixes are to be specified in the global routing table. In the case of IPv6, we can simply give one higher level route to represent the whole organisation, as we can shrink and expand the network prefix by varying it. This helps the global tables to remain small. This kind of setup did not exist in IPv4. (For more on CIDR, refer to Relevant concepts).

**Autoconfiguration in IPv6: Plug and play**

**What is autoconfiguration?** The first thing one should do is to set up a machine with an IPv6 address. There is an interesting feature in IPv6 called stateless autoconfiguration that's defined by RFC 2462 (see Resources). This RFC states that your host should be able to give you an automatic, globally unique IPv6 address.

For example, In AIX, you simply boot up your machine and type `autoconf6 -v` from the # prompt, and you will see your machine automatically detecting the subnet and assigning you a valid IPng address.

I ran `ifconfig` to see the IPv6 address. Here is a partial output of `ifconfig -a` on my AIX machine:

```
inet 9.184.209.3 netmask 0xffffff00 broadcast 9.184.209.255
inet6 fe80::207:30ee:edcb:d05d/64
```

I got the inet6 address when I ran `autoconf -v6` (inet6 is defined on en0). This machine now has both an IPv6 and IPv4 on the same physical ethernet interface.

**How is this done?** In very simple terms, the link-layer address is used as a base to get the IPv6 address and the host and router to communicate, so that the host can get an idea about the subnet. (Refer to the RFC for a more detailed discussion.)

**How about other operating systems?** The other UNIX implementations have similar IPv6 autconfiguration commands like AIX. There is also a variety of free-soft implementations of IPv6 (see Resources).

**Can I manually configure?** Yes. You can also configure an IPv6 address using `ifconfig`. It's important to plan your network to assign the network prefix.

**Tunneling and mapped IPng addresses: The transition should be smooth**

**Example of a transition problem**

Consider this situation. We have an existing IPv4 environment with IPv4-only hosts and routers. Now let's say we add a few IPv6 routers and hosts to our network. Some of these hosts have the capability to handle both IPv6 and IPv4 addresses, and some of them are pure IPv6 or pure IPv4. If we have to write an application that runs in this environment, then the application's client and
A server should be able to handle all possible client-server pairs. That is, a client or server can be purely IPv4, purely IPv6, or both IPv6- and IPv4-enabled. (For a detailed explanation, read RFC 2893: "Transition mechanisms for hosts and routers" -- see Resources.)

**What is the tunneling technique?** Again, let's take an example situation. We need to carry an IPv6 packet over an IPv4 network. How do we proceed? Simple -- we just encapsulate the IPv6 packet in an IPv4 packet and send it across the IPv4 network. This is called tunneling.

**Configured tunneling:** We need to configure the host that is at the entry point of the IPv4 network so that it can convert the IPv6 packet into an IPv4 packet. Also, the node that is the exit point of the IPv4 network needs to be configured so that it can convert the packet back to an IPv6 packet. This is called configured tunneling.

**Automatic tunneling:** If a host has the capability to do this conversion dynamically then it's called automatic tunneling.

**Support for Automatic tunneling in the protocol:** The nodes that utilize this technique are assigned special IPv6 unicast addresses. These addresses carry an IPv4 address in the low-order 32-bits. This type of address is termed an IPv4-compatible IPv6 address and has the following format:

| 80 bits | 16 | 32 bits |
|---------+----+---------|
| 0000...0000 | 0000 | IPV4 ADDRESS |

A second type of IPv6 address that holds an embedded IPv4 address is also defined. This address is used to represent the addresses of IPv4-only nodes (those that do not support IPv6) as IPv6 addresses. This type of address is termed an "IPv4-mapped IPv6 address" and has the format:

| 80 bits | 16 | 32 bits |
|---------+----+---------|
| 0000...0000 | FFFF | IPV4 ADDRESS |

**Usage of mapped addresses**

If you are writing an IPv6-enabled client, you're faced with this question: Do you send out an IPv6 packet or do you send out an IPv4 packet? You are given no guarantee about the underlying network. The next machine you contact to get this connection can be an IPv6 machine, an IPv4 machine, or a dual host.

Let's assume that the applications responsible for routing the connections are capable of knowing whether the next machine is an IPv6 machine or an IPv4 machine. In this case, it would be really helpful if we could have IPv6 addresses that can contain IPv4 addresses inside them. It would be good to have a mechanism (the ffff in mapped v4 addresses) to tell us if the address is referring to a pure IPv4 node; this would help us make appropriate decisions as to which type of packet is to be sent. Our discussion in the final section should make this clearer.
Porting IPv4 applications to IPv6

Here are some things to consider when porting an IPv4 application to IPv6:

- The `sockaddr_in6` structure and the `in6_addr` structure, which can hold 128 bit addresses, have been defined. Check if you are using the relevant IPv6 structure.
- `INADDR_ANY` and `INADDR_LOOPBACK` must be modified to `in6addr_any` or `in6addr_loopback` for assignments. The `IN6ADDR_ANY_INIT` or `IN6ADDR_LOOPBACK_INIT` macros can be helpful.
- Use `AF_INET6` instead of `AF_INET`.
- Note there are structures and programs that will work for IPv6 and IPv4. One of the links points to porting examples and this link can be referred to (see "Moving to IPv6" in Resources).
- Note that no change in the syntax is necessary when using certain functions for IPv6. The only difference when using these functions is that you must cast `sockaddr_in6` to struct `sockaddr*`.

The following macros and functions are used to write IPv6-enabled applications:

- The `IN6_IS_ADDR_V4MAPPED` can be used to determine whether an IPv6 address is an IPv4-mapped address.
- `gethostbyname` retrieves a network host entry via its name and address family.
- `getaddrinfo` returns address information related to a specified service location.
- `getnameinfo` returns the text strings associated with the supplied IP address and port number.
- `inet_pton` converts the specified address in text form to its binary equivalent.
- `inet_ntop` converts the specified binary address into a text equivalent that's suitable for presentation.
- `getaddrinfo` and `getnameinfo` can both be used to retrieve information related to IPv4 and IPv6 addresses. `inet_pton` and `inet_ntop` can both convert IPv4 and IPv6 addresses. This means that in "IPv6-ready" applications, you do not need to use either `inet_addr` or `inet_ntoa`.
- The following functions do not require a change in syntax when used for IPv6: `bind`, `connect`, `sendmsg`, `sendto`, `accept`, `recvfrom`, `recvmsg`, `getpeername`, and `getsockname`, although the code for these functions has been modified.

Writing a simple IPv6 client

Let's now take a look at the logic behind writing an IPv6-enabled client. I believe we are equipped with the basics. We know about IPv6 addresses. We will be able to recognise them if we see them in different representations. We will be able to autoconfigure an IPv6 address on our machine using autoconf. We also know about the mapped address transition mechanism and have an idea of the functions to use. Consider the following IPv4 client:

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
...
```
main(argc, argv) /* client side */
int argc;
char *argv[];
{
    struct sockaddr_in server;
    struct servent *sp;
    struct hostent *hp;
    int s;
    ...
    sp = getservbyname("login", "tcp");
    if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
    }
    hp = gethostbyname(argv[1]);
    if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
    }
    memset((char *)&server, 0, sizeof(server));
    memcpy((char *)&server.sin_addr, hp->h_addr, hp->h_length);
    server.sin_len = sizeof(server);
    server.sin_family = hp->h_addrtype;
    server.sin_port = sp->s_port;
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("rlogin: socket");
        exit(3);
    }
    ...
    /* Connect does the bind for us */
    if (connect(s, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("rlogin: connect");
        exit(5);
    }
}

We will examine the logic behind converting this to an IPv6-enabled client.

How to make the above client IPv6-enabled

**The sockaddr_in6 structure:** We are using `struct sockaddr_in` structure. We cannot use the same structure as the member `sin_addr` can only hold 32 bits. When porting the client to an IPv6 client we need to use `sockaddr_in6`, which can hold a 128-bit address.

```
struct sockaddr_in6 {
    u_char sin6_len;
    u_char sin6_family;
    u_int16_t sin6_port;
    u_int32_t sin6_flowinfo;
    struct in6_addr sin6_addr;
};
```

**The family:** `sin6_family` will have `AF_INET6` instead of `AF_INET` in our program.

**sin6_flowinfo field:** An application may specify the flow label and priority by setting the `sin6_flowinfo` field of the destination address `sockaddr_in6` structure. We can set it to 0 for now.

**What type of addresses will the client handle?** We have three situations. A user may pass:

1. A colon-separated IPv6 address
2. A dot-separated IPv4 address
3. Just a host name

**IPv6 address:** If it's a colon-separated IPv6 address, then we can just copy it into the structure.

**IPv4 address:** If it's an IPv4 address, we need to copy it into the last 32 bits and mark the 16 bits before those 32 bits with 0xffff.

**Host name:** If it's a host name, then we use gethostbyname to pick up the address. gethostbyname picks up an IPv4 address by default.

If we call gethostbyname after setting _res.options (resolv.h) in AIX, we can force it to do an IPv6 lookup:

```c
_res.options |= ~RES_USE_INET6
```

Note that if there is no IPv6 address present for the host name (in the /etc/hosts in UNIX or the DNS), then an IPv6 lookup by gethostbyname will return an IPv4 address, but we still need to do the mapping (filling bits 81-96 with 0xffff). Also, some implementations have another gethostbyname2 call for IPv6 lookups.

**Why is mapping done?** We do this mapping in order to use the sockaddr_in6 structure in the connect call regardless of whether we are trying to send to an IPv6 address or an IPv4 address. If not, we will require two connect calls -- one that takes an IPv6 address and one that takes an IPv4 address. The other technique is to use a union of the sockaddr_in and sockaddr_in6 structure. Programmers can also devise their own techniques and it's not compulsory to map.

```c
// use the isinet_addr call to find out whether its a valid
// dotted ipv4 address
if (isinet_addr(hostname )) {
    ......

    //now you might wonder what s6_addr16[5] is - this is basically a union member normally
    //defined in in.h which will point to bits 81-96
    ip6.sin6_addr.s6_addr16[5] = 0xffff;

    //now we are copying the ipv4 address in the last 32 bits
    bcopy(address, &ip6.sin6_addr.s6_addr16[6], sizeof(struct in_addr));

    ip6.sin6_len = sizeof(struct in6_addr);
    ip6.sin6_family = AF_INET6;
    ......
}

//check if its is an IPv6 : separated address - inet_pton is used for this
else if (inet_pton(AF_INET6, hostname, &ip6.sin6_addr) > 0) {
    //note inet_pton will take care of setting the address
    ......
    ip6.sin6_family = AF_INET6;
    ip6.sin6_len = sizeof(struct sockaddr_in6);
    ......
```
else {
// now its not a v6 address or a v4 address so it should be host name
// do a v6 lookup , note that a v6 lookup will look for a v6 address if not
// present it can pick up a v4 address

// res_init is defined in resolv.h
res_init();
_res.options |= RES_USE_INET6;
hptr = gethostbyname(name);
......
// check hptr->h_addrtype if its AF_INET6 you can copy the address directly
// if not you need to map it.
......
......

if (connect(sd, &ip6, sizeof (ip6 < 0)
{
// connect failure
......
}
else
{
// continue with the program.
}

**Summary of the above logic**

To summarize the logic, we check to see if we got a dotted IPv4 address to handle. If so, we go ahead and map it and fill in an IPv6 structure, to be used by the connect call later. If it's an IPv6 address, we copy it directly to the IPv6 structure. If it's a hostname, we try and do an IPv6 lookup. We can get an IPv4 or an IPv6 address. We know this from the family field. Accordingly, we either map it or copy it, then do a single connect call regardless of whether it's an IPv4 or an IPv6 address, and proceed with our program.

**Conclusion**

We have looked only at the concepts we need to write the above program. There are many more interesting concepts that will soon become part of everyday life. There are controversies and constructive debates about things like DNS for IPv6 and stateful autoconfiguration for IPv6(DHCP). These topics, along with others, such as implementation of other layers, how routing will be done, and how autoconfiguration will be implemented, will make for interesting discussion. I hope to see you soon in a more exciting IPv6 world!
Resources

• **IP Next Generation Overview**, by Robert M Hinden, gives a brief overview of all the IPng concepts.
• **Kame project** is a joint effort of seven companies in Japan that provide a free IPv6 and IPsec (for both IPv4 and IPv6) stack for BSD variants.
• **6bone** is an IPv6 testbed to assist in the evolution and deployment of IPv6.
• IPv6 drafts and RFCs can be found in **IPng Current Specifications**. They give all the required details in a clear and logical order.
• Stateless autoconfiguration is defined in RFC 2462: **IPv6 Stateless Address Autoconfiguration**.
• RFC 2893, **Transition Mechanisms for IPv6 Hosts and Routers**, specifies IPv4 compatibility mechanisms that can be implemented by IPv6 hosts and routers.
• Additional information on NAT can be found in RFC 1631, **The IP Network Address Translator (NAT)**.
• For more in-depth overviews of CIDR, read RFC 1517, RFC 1518, RFC 1519, and RFC 1520.
• Find out how **IPv6 for OS/390** provides an implementation of IPv4 and IPv6 for OS/390.
• **Planning for growth**, from IBM's High Volume Web Site Team, can show you how your Web site can satisfy future demands and evaluate potential workload and infrastructure changes.
• **IBM's Patterns for e-business site** features a number of reusable assets that can help speed the process of developing applications.
• Browse **more Web architecture resources** on developerWorks.
About the author

Senthil Sundaram

Senthil Sundaram works with the Networking Software Group at IBM Software Labs in Bangalore, India, where he provides Level 3 support for AIX TCP/IP stack. He received his Master's degree in computer applications from Pondicherry University.

© Copyright IBM Corporation 2001
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)