IPv6 and the Future of the Internet

A Technical White Paper

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
1 (800) 786.7638
1.512.434.1511

Please
Recycle

Adobe PostScript

Contents

CHAPTER 1

Introduction

The TCP/IP protocol suite is the fundamental basis of the Internet. Virtually all data communications among companies and individuals today are based on these Internet protocols, including Web browsers, electronic mail, file transfers, and remote logins. Over the next few years, the Internet will face an important transition that will enable it to grow beyond its current limitations. This transition will include moving to a newer, more reliable version of the Internet protocol (IP) — IPv6. The changes are to the IP portion of the TCP/IP protocol suite.

Internet Growth and Internet Protocols

By all accounts, the Internet is growing at an incredible rate. The Internet Software Consortium's annual survey of Domain Name Servers (DNS — Internet service defining all permanent host systems) indicates over 56 million hosts are identified in DNS. This is up from over 36 million a year earlier and from less than two million in July of 1993. The latest number represents an annual growth rate of over 65 percent, and includes only those systems that are publicly advertised by DNS. Private nodes are not included in this count.

This growth, with its anticipated future requirement for more addresses, is a key factor driving the need for a new version of the Internet protocol. With the Internet protocol, the total number of possible unique Internet addresses is in the four billion range. However, this is a theoretical limit and is actually constrained by several factors to a number in the range of a few hundred million addresses. Currently, there are estimated to be approximately 100 million computers connected to the Internet. While it's not absolutely clear when the Internet will run out of addresses, most estimates are somewhere between the years 2002 and 2012.

The problem of running out of IP addresses is going to become worse over time. Today, the most common device that takes up an Internet address is a PC, which can cost hundreds to thousands of dollars. In the future, the numbers and types of devices connecting to the Internet will increase. Devices like PDAs, pagers, telephones, automobiles, and many other new inventions will have computers built into them. Many of these devices will cost significantly less than a PC, and as these devices start to network, many of them will require Internet connectivity. This will dramatically fuel demand for even more IP addresses.

It has also become more and more difficult to get IPv4 network addresses. As fewer and fewer Internet addresses are available, corporations must justify their requests to their upstream ISPs for additional addresses. In some cases, the justification needs to include a detailed business plan, which corporations might be reluctant to divulge. Depending on the justification, and the number of nodes requested and available, some requests may be denied. Outside the U.S., it can be even tougher to get new addresses. Some countries are running out of IP addresses even faster than in the U.S.

The Internet is currently based on Version 4 of the Internet protocol (IPv4). This standard was authored by Jon Postel and published on September 1, 1981. At that time, there were only approximately one thousand computers on the Internet, and virtually all were large, timeshare-based systems in universities and government offices. Computers designed for single users were almost non-existent.

In 1981, the four billion addresses available under the IPv4 standard seemed like an address space so large, there was no need to be conservative with IP address allocation. As a result early address allocations were not as well distributed as later allocations. Early adopters of the Internet received the most addresses. This meant that some of the Internet address space was wasted and could not be reallocated. In addition, some of the potential address space is used to make networks more maintainable and routable using blocks of addresses, which further consumes available addresses.

To overcome the limitations of IPv4, the industry recognized the need to move from Version 4 (IPv4) to Version 6 (IPv6) of the Internet Protocol. IPv6 uses an address scheme that is based on a unique Internet address of 128 bits, as opposed to the 32 bits of IPv4.

One way to understand how many addresses 128 bits represents is to think of a book that describes every possible Internet node address with 100 node descriptions per page. If each page was .1mm thick and was printed double-sided, the book to describe all possible IPv4 addresses would be 2000 meters thick. This may sound big, but the book describing all possible IPv6 addresses would be $2*10^{16}$ light-years thick. To date, the farthest sight seen by the Hubble telescope is less than $2*10^{13}$ light-years. Certainly IPv6 is considered quite adequate for the foreseeable future.

IP-Version Independent Technologies

When the Internet Engineering Task Force (IETF) began designing IPv6, it was recognized that for an extended time, both IPv4 and IPv6 nodes would co-exist on the network. To help make the transition, the IPv6 code interface was designed to communicate with both IPv4 and IPv6 nodes. This is achieved by supporting both protocols through a single programming interface. Specifically, the new socket interface API for IPv6 will support both IPv4 and IPv6 communication transparently.

Using this new interface, applications that have been upgraded can automatically use IPv4 to talk to existing IPv4 clients and servers, as well as to IPv6 systems that have been configured with only an address for the IPv4 protocol.

Once applications have been upgraded to the new IPv6 interface, they can start using the IPv6 protocol to connect to other IPv6 nodes. This will occur as soon as a company deploys an IPv6 network. Most importantly with this transition is that the application need not worry about which protocol is used. Once the new interfaces are used and the system is configured for IPv6 (given an IPv6 address and configuration), the IPv6 socket interface will automatically choose the right protocol to use to communicate with any given node.

IPv6: The Next Generation Protocol

For more than 17 years, Sun Microsystems has been instrumental in the development of state-of-the art networked systems. Sun developed and produced the first low-cost workstation with built-in networking. And today, Sun products continue to be engineered for use in networked environments. Sun consistently delivers robust, flexible, and powerful network technology based upon industry standards, such as Ethernet, TCP/IP, and numerous other key standards. And Sun's commitment to standards continues with its support of IPv6.

Sun has been involved with IPv6 form the beginning, and is represented on almost every major working group of the IETF, including the Internet Architecture Board. Sun engineers serve as IETF Area Directors and as the IETF Working Group co-chairs for the IP Next Generation (IPNG) Working Group and the Next Generation Transition (ngtrans) Working Group. They have also co-authored numerous IETF Standards Specifications.

Sun was the first vendor to provide an IPv6 prototype for release to the Internet community. In addition, Sun engineers have been involved in creating the *6bone*, the first IPv6 backbone network, and Sun was one of the first sites on the *6bone*.

Sun is committed to using its expertise in developing the IPv6 environment to help customers make an easy and seamless upgrade from IPv4 to IPv6.

The Solaris™ Operating Environment: The Basis of the Web

Driven by a vision of seamless access to information and backed by over 17 years of consistent execution, Sun is recognized as a world leader in open network computing. And Sun continues to invest in the core technologies required to deliver network connections as reliable as the dial tone — the WebTone.

With TCP/IP and NFS™ standards-based networking built into the software right from the beginning, the Solaris™ Operating Environment offers proven interoperability and includes leading-edge features to make networking easier. And Sun continues to improve performance, and includes the protocols needed to support new applications as they become available.

With hundreds of world-class development tools, powerful infrastructure technologies like CORBA and Java™ technology development products, and advanced client/server and network management tools, the Solaris Operating Environment is ideal for network application software development, including IPv6 applications.

CHAPTER **2**

IPv4 versus IPv6: An Overview

Most of the growth of the Internet has occurred using the IPv4 specification of the Internet protocol. While this version of the protocol has done an excellent job of handling the tremendous growth of the Internet, it is rapidly approaching the end of its physical limits. Given anticipated growth of the Internet in coming years, and the necessary upgrade of millions of network nodes and their users, a new protocol is required. However, before discussing the details of IPv6, it is important to understand the need for this transition.

IPv4 Background

The Internet Engineering Task Force published the IPv4 specification (RFC 791) in the fall of 198l. When the IPv4 specification was released, the Internet was a community of approximately one thousand systems. The IPv4 specification called for every IP address to be represented by a 32-bit number made up of four groups of eight-bit numbers. This provides a total of just over four billion addresses, although only a few hundred million are actually available due to hierarchic allocation schemes.

Since the release of IPv4, the Internet population has grown to over 100 million computers, increasing far faster than anticipated. As the pool of available addresses decreases, it will become increasingly difficult to obtain IPv4 addresses. Furthermore, the pace of this growth is expected to continue for years to come.

The bottom line is this: *The Internet is running out of addresses*. And by some hard estimates, this could happen as soon as 2002.

Early IP assignments reserved addresses for some corporations and institutions in very large blocks. These “Class A” and “Class B” network assignments were issued in the early days when the current growth was not anticipated. While some early adopters may still have addresses available for internal usage, the pool of unissued addresses is becoming smaller every day. The addresses that were handed out to some of the early large corporate networks cannot now be reissued to other users.

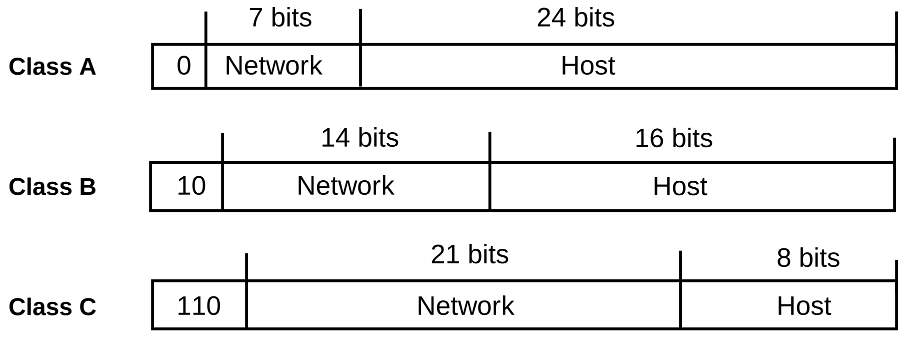

FIGURE 1 Mapping of network and host identifier in the 32-bit IPv4 protocol

NAT as a Stopgap Measure

In the short-term, Network Address Translation (NAT) is relieving the pressure for additional address space. By using some special addresses (such as 10.*.*.*, and 192.168.*.*) that are reserved for local usage, NAT allows network administrators to hide large communities of users behind firewalls and NAT boxes. The rest of the Internet sees all of the requests for users in the community as coming from one NAT box. When the response to the request comes back to the NAT box, it forwards the information to the appropriate local user. Since multiple, different corporate networks can each reuse the same local addresses, NAT reduces the need for new unique Internet addresses.

Unfortunately, NAT is a stopgap measure, not a permanent solution. It addresses the needs of large communities of client systems, but it does not help Internet servers that each require a unique permanent address. Nor does it work for peer-to-peer communication for the same reason. NAT also breaks the end-to-end model of Internet access and undermines the IPsec model of Internet security.

IPv6 is the Long-Term Solution

The IPv6 protocol solves the pressing problems of IP addressing, while simultaneously making administration easier. It uses an address scheme made up of eight groups of 16 bits, defining a 128-bit network address. This addressing scheme provides roughly $6*10^{23}$ addresses per square meter over the entire face of the earth. As a result, it is expected that the IPv6 protocol will last many years into the future.

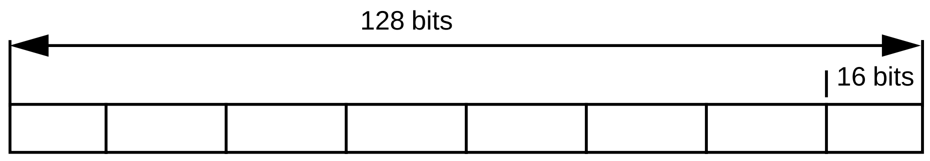

FIGURE 2 The 128-bit IPv6 address format. Each 16-bit segment is identified by 4 hexadecimal digits with colons separating the segments. A single double colon can be used in the address to represent one or more segments of all zeros.

With a pool of addresses this large, it is possible to utilize some of the addresses to make network management and routing easier. The new address scheme allows automatic address configuration and reconfiguration (servers can re-number network addresses without accessing all clients, and mobile clients can move about the network). NAT servers are no longer required because there is no need to use private addresses. Each computing device can have a its own unique address without fear of running out of addresses.

Planning for the Future Using IPv6

The IPv6 specification has several possible APIs to enable IPv6 communications, and most are IP-version independent. By using these APIs, developers can write a single segment of code that will support both IPv4 and IPv6 communications. Based on the name of the system that is the target of communication and the configuration of the current node, the API will determine the target IP address and whether it’s using IPv4 or IPv6 protocol. By using IP version independent APIs, developers can enable IPv6 communication transparently.

The most common API, particularly in the short term, is the *Basic Socket API*. Once an application has been ported to this interface, the socket code determines, based on the current configuration, whether to communicate using IPv4 or IPv6. The application automatically decides which version of the protocol to choose. This enables the use of IPv4 now and requires no new coding to utilize IPv6 later. *Code once and get both protocols*.

Name servers manage IPv4 and IPv6 environments and their cooperation. Once a name server has been upgraded to support IPv6, it will return the appropriate address (IPv4 or IPv6) for the remote system based on the capabilities and configuration of the environment. Using the name server to manage information about IPv4 and IPv6 makes code development and management much easier for developers. More detail on how name servers work is available in Chapter 4, "Porting Client Code to IPv6".

Sun provides a complete set of tools to help developers convert current applications from IPv4 to IPv6. In particular, Sun’s *Socket Scrubber* can be used to search large segments of code to identify IPv4 socket-dependent routines. This tool searches for references to standard IPv4 socket routines and key data structure identifiers. Using Socket Scrubber, developers can quickly identify where changes need to be made, easing the process of upgrading code to use the new interface. For more detail on this process, see Chapter 6, "Tools to Help Upgrade Applications".

All new development should use IP version independent APIs. Since these interfaces support both IPv4 and IPv6, their use assures a seamless upgrade path to the IPv6 standard. Continuing to use IPv4 interfaces at this point is to build in obsolescence.

IPv6 is the future of the Internet. The question is not whether IPv6 will be the future standard protocol for the Internet, but rather when this transition will occur. Developers should prepare for the future now by being proactive with new development and effectively planning the conversion of current IPv4 applications.

CHAPTER **3**

IPv6 Application Program Interfaces

Developers generally access new capabilities through new interfaces. Since the parameters (such as the node address) have to change between the two versions, the API itself must change. Fortunately, the engineers that developed IPv6 have made this a fairly simple upgrade.

IETF Specification

The IETF has released several RFC specifications for the transition to the IPv6 environment. The most significant specifies the basic socket communication used in 90 to 95 percent of all IPv6 applications. In addition, the IETF is currently working on an RFC — an advanced IPv6 API — that will deliver additional socket capabilities and extended features to enable greater levels of control for programmers. It is expected that less that 10 percent of IPv6 applications will use this extension.

RFC 2553 is a primary IPv6 specification that covers basic socket communication and identifies IPv6 extensions to the existing socket interface. RFC 2553 is the basic API because applications that use it will be insulated from the specific underlying protocol. Since this specification supports both IPv4 and IPv6 communication, it can be thought of as an upward extension of the current IPv4 specification. In general, a one-to-one relationship exists between the IPv4 and IPv6 calls. As a result, it is fairly simple to upgrade existing IPv4 socket-based applications to use the IPv6 interface.

Basic Socket API

There is very little doubt that, for an extended period of time, the Internet will be made up of both IPv4 and IPv6 hosts. For that reason, the IPv6 basic socket API supports both IPv4 and IPv6. This approach is called a dual-stack interface. Once an application has been upgraded to the IPv6 socket interface, no more code is required to enable communication with both IPv4 and IPv6 systems. When a call is made to the new socket interface, it will look at the data structures and determine if it is possible to communicate with this node using IPv6. If not, the socket will automatically make the connection using an IPv4 protocol connection. Since all current Internet software use IPv4, a dual-stack IPv6 application can communicate using IPv4 to all current software without any additional coding of the IPv4 applications.

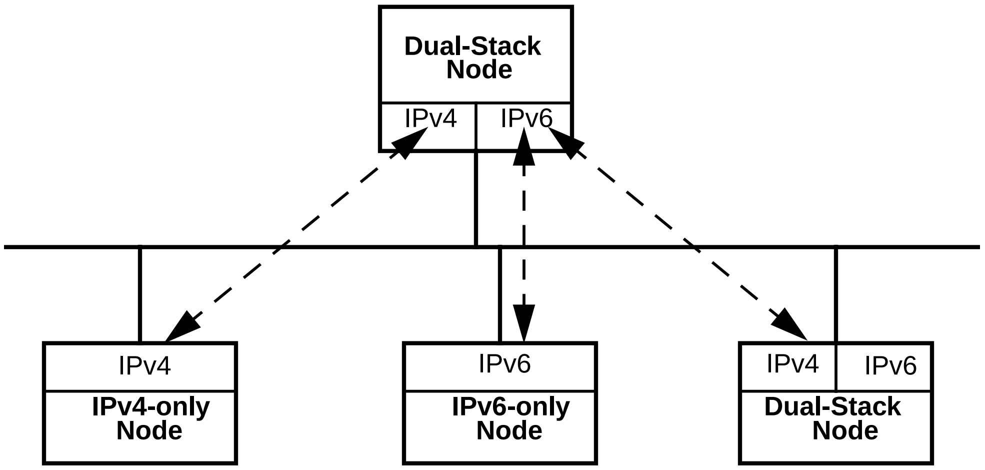

FIGURE 1 The dual-stack socket interface will default to using the IPv6 protocol if it is available. If not, it will automatically communicate using IPv4 protocol.

FIGURE 1 provides a more detailed breakdown of IPv4 and IPv6 communication based on the capabilities of the client and the server.

Type of Application (Type of Node)	IPv6-unaware Server (IPv4-only Node)	IPv6-unaware Server (IPv6-enabled Node)	IPv6-aware Server (IPv6-only Node)	IPv6-aware Server (IPv6-enabled Node)
IPv6-unaware Client (IPv4-only Node)	IPv4	IPv4	X	IPv4
IPv6-unaware Client (IPv6-enabled Node)	IPv4	IPv4	X	IPv4
IPv6-aware Client (IPv6-only Node)	X	X	IPv6	IPv6
IPv6-aware Client (IPv6-enabled Node)	IPv4	(IPv4)	IPv6	IPv6

TABLE 1 IPv4 and IPv6 interoperability

In the table above, “X” denotes that communication between the respective server and client is not possible at the IP layer. Other tools, like proxies, can be used for this communication. This table assumes that the dual-stack interface has both an IPv4 and IPv6 address in the respective name service(s). An IPv6-unaware node has the IPv4 stack only. An IPv6-enabled node has a dual stack and at least one interface IPv6 configured. An IPv6-only system implements only the IPv6 stack and only has one address in the name service database.

It is a straightforward process to convert IPv4 socket applications to use the IPv6 socket extensions. In general, there is a direct conversion to update the IPv4 socket call to the IPv6 socket call. In fact, there are some places where two or three IPv4 calls can be replaced by a single IPv6 call, although users are not required to use these shortcuts. The first argument of the socket call will change and a few of the parameters will need to be updated, but generally this is a very simple process. The format of the calls and the data structures are similar, and tools are available to help identify where changes are required.

Advanced Socket Capabilities

As previously mentioned, the socket specification for IPv6 has been formalized with RFC 2553 and is expected to become the primary standard for Internet communication. This specification is considered complete and can handle virtually every need for IPv6. It has been fully defined, and most system vendors have publicly stated their commitment to support it.

On the other hand, there has been a growing consensus that a few additional capabilities beyond the current specification would be desirable. RFC 2292 is designed to offer some extensions to the current specification, enabling developers to have more control of IPv6-only communications. It is estimated that these additional control requirements would only be used by approximately 5 to 10 percent of all socket applications. For example, advanced capabilities enable operations such as source routing, raw socket access, interface identification, and extension headers not defined in RFC 2553 are all being considered.

These advanced capabilities are being evaluated by the IETF, and Sun is working with other vendors to help solidify all of these advanced capabilities. As more information is available, Sun will continue to update the developer community.

Sun Remote Procedure Call

Another method for applications to communicate is through Remote Procedure Calls (RPC). The RPC facility shipped in the Solaris Operating Environment is built on a protocol-independent base. Using this interface, application developers can use these calls to communicate transparently with IPv6 clients.

Sun's RPC facility provides another way to communicate using the IPv6 protocol. By using standard calls, applications can use the existing RPC conduit to communicate to either IPv4 or IPv6 nodes. Since all of the modifications to allow the Sun RPC to communicate using IPv6 occur in the Sun libraries, there is no need for application developers to re-code their applications. By using standard dynamic linking to the newest libraries, applications will transparently use IPv6 when appropriate.

CHAPTER **4**

Porting Client Code to IPv6

One of the key technologies making the Internet possible is name services. Name server technologies, such as the Domain Name Service (DNS) and the Network Information Service (NIS and NIS+), are used to translate logical references (such as the names of computers) to physical references (addresses, names, etc.). IPv6 uses these name-to-address translation routines to help smooth the transition from IPv4- to IPv6-based protocols.

Name Services and the IPv4 to IPv6 Transition

When the IETF began planning for the transition from IPv4 to IPv6, they quickly recognized that this could be a complex undertaking. To help simplify the process, the IETF created a document to help companies make the transition more easily. The resulting specification is RFC 1933.

There are several methods that the IETF recommends for updating from pure IPv4 to an IPv6-enabled environment. Depending on the needs of the organization, the transition can start by converting the clients in the environment, the servers in the environment, or the routers between the environments. Regardless of where the transition begins, IPv6-enabled name servers are an important early requirement.

Name servers connect the two IP environments and determine which protocol stack is accessed. IPv6-enabled name servers store the name-to-address translation for both IPv4 and IPv6 addresses. When a request is made to translate a network name to an address, the name server will reply with the appropriate address based on the requesting node, target node, and call parameters. If both the requesting and destination nodes are capable of IPv6 communication (and the calling parameters allow it), the name server will respond with the IPv6 address. Using this address, the socket interface code will communicate using the IPv6 protocol. As IPv6 is enabled through the environment, the name server will automatically convert the common communication protocol from IPv4 to IPv6.

Node Name	Node Type	IPv4 Address	IPv6 Address
earth	Dual Stack	129.146.86.2	fe80::aoo:20ff:fea1:6a83
venus	IPv4 Only	129.146.86.14	
mars	IPv6 Only		fe80::a00:20ff:fe87:8dde

TABLE 1 An example of what the data structure for a name server might look like. The node type would not explicitly be stored, but could be inferred from the other columns.

To make certain that the upgrade from IPv4 to IPv6 is as simple as possible, Sun has upgraded the name servers that ship with the Solaris 8 Operating Environment to include IPv6 support. These name servers store both IPv4 and IPv6 address information in their databases, thereby supporting both the IPv4 and IPv6 environments. This makes the Solaris Operating Environment an excellent point from where to begin the upgrade to the IPv6 protocol.

IPv4 and IPv6 Interoperability

Using tools provided by Sun makes converting from an IPv4 environment to an IPv6 environment straightforward. After installing several IPv6 capable name servers, the upgrade to the IPv6 environment may begin at any place on the network. For example, nodes are transitioned from an IPv4-only system to dual-stack systems by upgrading the IP socket calls from the IPv4 socket-only interface to the dual-stack IP socket interface. An example of this dual-stack arrangement is shown in FIGURE 1 on page 15.

After an application has been upgraded to use the dual-stack IPv4/IPv6 interface, the next step is to configure the IPv6 information. To do this, a user must configure such parameters as the IPv6 address of the node and ensure that the node is connected to the IPv6 network. Once IPv6 is enabled, the node can be registered with the IPv6 name server. For more details on this process, see the *Porting Network Applications to the IPv6 APIs* white paper on the Sun Web site at *http://www.sun.com/solaris/ipv6*.

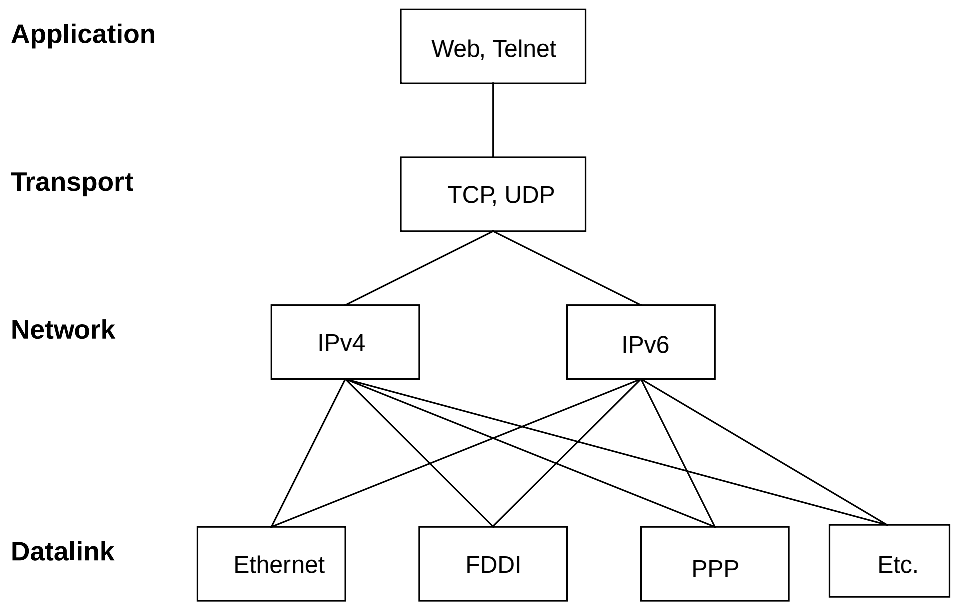

FIGURE 1 Dual-stack protocols

Once a node has been registered with the name server, that name server returns the IPv6 address for the destination, and packets are routed using IPv6 protocols. Any dual-stack node will have both an IPv4 and an IPv6 address in the name server database. The name server will automatically return the IPv4 address if it is the only address stored. If an IPv6 address, or if both IPv4 and IPv6 names are stored, the name server will return the IPv6 address. Based on the address returned, the application automatically selects the appropriate protocol and connects to the node.

IPv4 versus IPv6 Naming API Differences

The process of converting from an IPv4 naming routine to an IPv6 based routine is fairly simple. The following example code segment demonstrates this process.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

int
myconnect(char *hostname, int port)
```

```
{
    struct sockaddr_insin;
    struct hostent*hp;
    int         sock;

    /*
     * Open socket.
     */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("socket");
        return (-1);
    }

    /*
     * Get host address.
     */
    hp = gethostbyname(hostname);
    if (hp == NULL || hp->h_addrtype != AF_INET || hp->h_length != 4) {
        (void) fprintf(stderr, "Unknown host \"%s\"\n", hostname);
        (void) close(sock);
        return (-1);
    }

    sin.sin_family = AF_INET;
    sin.sin_port = htons(port);
    (void) memcpy((void *)&sin.sin_addr, (void *)hp->h_addr,
        hp->h_length);
    /*
     * Connect to the host.
     */
    if (connect(sock, (struct sockaddr *)&sin, sizeof (sin)) == -1) {
        perror("connect");
        (void) close(sock);
        return (-1);
    }
    return (sock);
}

main(int argc, char *argv[])
{
    int     sock;
    char    buf[BUFSIZ];
    int     cc;
    switch (argc) {
    case 2:
        sock = myconnect(argv[1], IPPORT_ECHO);
        break;
    case 3:
        sock = myconnect(argv[1], atoi(argv[2]));
        break;
    default:
        (void) fprintf(stderr,
            "Usage: %s <hostname> <port>\n", argv[0]);
        exit(1);
    }
    if (sock == -1)
        exit(1);
    if (write(sock, "hello world", strlen("hello world") + 1) == -1) {
        perror("write");
        exit(1);
    }
    cc = read(sock, buf, sizeof (buf));
    if (cc == -1) {
        perror("read");
        exit(1);
    }
    (void) printf("Read <%s>\n", buf);
    return (0);
}
```

The preceding example uses the IPv4 standard `gethostbyname()` routine to translate the IPv4 node name to an IPv4 address and opens the connection. Compare this to the same call to translate an IPv6 name into an IPv6 name using the IPv6 `getipnodebyname()` equivalent in the following example.

```
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netdb.h>
    #include <string.h>
    #include <stdio.h>
    #include <unistd.h>

    int
    myconnect2(char *hostname, int port)
    {
        struct sockaddr_in6sin;
        struct hostent*hp;
        int         sock, errnum;

        /*
         * Open socket.
         */
        sock = socket(AF_INET6, SOCK_STREAM, 0);
        if (sock == -1) {
            perror("socket");
            return (-1);
        }

        /*
         * Get host address.  An IPv4-mapped IPv6 address is okay.
         */
        hp = getipnodebyname(hostname, AF_INET6, AI_DEFAULT, &errnum);
        if (hp == NULL) {
            (void) fprintf(stderr, "Unknown host \"%s\"\n", hostname);
            (void) close(sock);
            freehostent(hp);
            return (-1);
        }

        /* Make sure all sockaddr_in6 fields are zero */
        bzero(&sin, sizeof (sin));
        sin.sin6_family = hp->h_addrtype;
        sin.sin6_flowinfo = 0;
        sin.sin6_port = htons(port);
        (void) memcpy((void *)&sin.sin6_addr, (void *)hp->h_addr,
            hp->h_length);
        freehostent(hp);

        /*
         * Connect to the host.
         */
        if (connect(sock, (struct sockaddr *)&sin, sizeof (sin)) == -1) {
            perror("connect");
            (void) close(sock);
    main(int argc, char *argv[])
    {
        int     sock;
        char    buf[BUFSIZ];
        int     cc;
        switch (argc) {
        case 2:
            sock = myconnect2(argv[1], IPPORT_ECHO);
            break;
        case 3:
            sock = myconnect2(argv[1], atoi(argv[2]));
            break;
        default:
            (void) fprintf(stderr,
                "Usage: %s <hostname> <port>\n", argv[0]);
            exit(1);
        }
        if (sock == -1)
            exit(1);
        if (write(sock, "hello world", strlen("hello world") + 1) == -1) {
            perror("write");
            exit(1);
        }
        cc = read(sock, buf, sizeof (buf));
        if (cc == -1) {
            perror("read");
            exit(1);
        }
        (void) printf("Read <%s>\n", buf);
        return (0);
    }
```

This example is a minimal change from `myconnect()` when porting to handle both IPv4 and IPv6. It uses IPv4-mapped addresses stored in `sockaddr_in6` data structures when communicating with IPv4 nodes.

A different way to port is to use the new API functions `getaddrinfo(3N)` and `getnameinfo(3N)` as shown in the following example. This code has no knowledge about address families. As an additional bonus, the example is much more robust since it tries all addresses returned by the name server until it finds one when the `connect()` succeeds.

```
    int
    myconnect3(char *hostname, char *servicename)
    {
        struct addrinfo*res, *aip;
        struct addrinfohints;
        int         sock = -1;
        int         error;

        /*
         * Get host address.  Any type of address will do.
         */
        bzero(&hints, sizeof (hints));
        hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
        hints.ai_socktype = SOCK_STREAM;

        error = getaddrinfo(hostname, servicename, &hints, &res);
        if (error != 0) {
            (void) fprintf(stderr,
                "getaddrinfo: %s for host %s service %s\n",
                gai_strerror(error), hostname, servicename);
            return (-1);
        }
        for (aip = res; aip != NULL; aip = aip->ai_next) {
            /*
             * Open socket.  The address type depends on what
             * getaddrinfo() gave us.
             */
            sock = socket(aip->ai_family, aip->ai_socktype,
                aip->ai_protocol);
            if (sock == -1) {
                perror("socket");
                freeaddrinfo(res);
                return (-1);
            }
            /*
             * Connect to the host.
             */
            if (connect(sock, aip->ai_addr, aip->ai_addrlen) == -1) {
                perror("connect");
                (void) close(sock);
                sock = -1;
                continue;
            }
            break;
        }
        freeaddrinfo(res);
        return (sock);
}
main(int argc, char *argv[])
{
        int     sock;
        char    buf[BUFSIZ];
        int     cc;

        switch (argc) {
        case 1:
            sock = myconnect3(NULL, NULL);
            break;
        case 2:
            sock = myconnect3(argv[1], "echo");
            break;
        case 3:
            sock = myconnect3(argv[1], argv[2]);
            break;
        default:
            (void) fprintf(stderr,
                "Usage: %s <hostname> <port>\n", argv[0]);
            exit(1);
        }
        if (sock == -1)
            exit(1);

        if (write(sock, "hello world", strlen("hello world") + 1) == -1) {
            perror("write");
            exit(1);
        }
        cc = read(sock, buf, sizeof (buf));
        if (cc == -1) {
            perror("read");
            exit(1);
        }
        (void) printf("Read <%s>\n", buf);
        return (0);
    }
```

As can readily be seen, the differences between the code segments are fairly minor, and porting from an IPv4 name server environment to an IPv6 name server is rather straightforward.

In a more generalized sense, these code examples demonstrate the type of changes required to port an application from an IPv4-only environment to a dual-stack, IPv4-/IPv6-capable environment.

CHAPTER **5**

Porting Server Code to IPv6

The last code segment in the previous section illustrated how the client-side code changes between IPv4 and IPv6. The changes required to transition a server application to communicate with both IPv4 and IPv6 client applications are similar in their level of simplicity.

First, consider the following IPv4 server application:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

#defineBACKLOG1024      /* max # pending connections */

void do_work(int sock);

int
myserver(int port)
{
    struct sockaddr_inladdr, faddr;
    int         sock, new_sock, sock_opt;
    socklen_tfaddrlen;
```

Now, compare this to the IPv4/IPv6 dual stack:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <errno.h>
```

```
#include <unistd.h>
void do_work(int sock);

int
myserver2(int port)
{
    struct sockaddr_in6laddr, faddr;
    int         sock, new_sock, sock_opt;
    socklen_tfaddrlen;
    char        addrbuf[INET6_ADDRSTRLEN];

    /*
     * Set up a socket to listen on for connections.
     */
    /* Make sure all sockaddr_in6 fields are zero */
    bzero(&laddr, sizeof (laddr));
    laddr.sin6_family = AF_INET6;
    laddr.sin6_flowinfo = 0;
    laddr.sin6_port = htons(port);
    laddr.sin6_addr = in6addr_any;/* structure assignment */

    sock = socket(AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("socket");
        return (-1);
    }

    /*
     * Tell the system to allow local addresses to be reused.
     */
    sock_opt = 1;

    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (void *)&sock_opt,
        sizeof (sock_opt)) == -1) {
        perror("setsockopt(SO_REUSEADDR)");
        (void) close(sock);
        return (-1);
    }

    if (bind(sock, (struct sockaddr *)&laddr, sizeof (laddr)) == -1) {
        perror("bind");
        (void) close(sock);
        return (-1);
    }

        if (listen(sock, BACKLOG) == -1) {
        perror("listen");
        (void) close(sock);
        return (-1);
    }
    /*
     * Wait for a connection request.
     */
    for (;;) {

        faddrlen = sizeof (faddr);
        new_sock = accept(sock, (struct sockaddr *)&faddr, &faddrlen);
        if (new_sock == -1) {
            if (errno != EINTR && errno != ECONNABORTED) {
                perror("accept");
            }
            continue;
        }

        if (IN6_IS_ADDR_V4MAPPED(&faddr.sin6_addr)) {
            struct in_addr ina;

            IN6_V4MAPPED_TO_INADDR(&faddr.sin6_addr, &ina);
            (void) printf("Connection from %s/%d\n",
                inet_ntop(AF_INET, (void *)&ina,
                addrbuf, sizeof (addrbuf)),
                ntohs(faddr.sin6_port));
        } else {
            (void) printf("Connection from %s/%d\n",
                inet_ntop(AF_INET6, (void *)&faddr.sin6_addr,
                addrbuf, sizeof (addrbuf)),
                ntohs(faddr.sin6_port));
        }
        do_work(new_sock);/* do some work */
    }

    /*NOTREACHED*/
}

void
do_work(int sock)
{
    char    buf[BUFSIZ];
    int     cc;

    while (1) {
        cc = read(sock, buf, sizeof (buf));
        if (cc == -1) {
            perror("read");
            exit(1);
        }
        if (cc == 0) {
            /* EOF */
```

```
                (void) close(sock);
                (void) printf("Connection closed\n");
                return;
            }

            buf[cc + 1] = '\0';
            (void) printf("Read <%s>\n", buf);

            if (write(sock, buf, cc) == -1) {
                perror("write");
                exit(1);
            }
        }
    }

    main(int argc, char *argv[])
    {
        if (argc != 2) {
            (void) fprintf(stderr,
                "Usage: %s <port>\n", argv[0]);
            exit(1);
        }
        (void) myserver2(htons(atoi(argv[1])));
        return (0);
    }
```

In this last example, the application uses a single socket to communicate with both IPv4 and IPv6 clients. This is possible by binding the `AF_INET6` socket to `in6addr_any`.

Note that this server prints the addresses of the IPv4 node as a mapped address (i.e., with the leading "::ffff:" string). This behavior can be modified by using the `IN6_IS_ADDR_V4MAPPED` macro and calling `inet_ntop` differently for the mapped addresses.

A final example will show how this same server can be created using the new `getaddrinfo()` and `getnameinfo()` calls which remove address specific manipulations from the code. Also, the example code prints the peer's address in both numeric form as well as a host and service name.

```
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    #include <netdb.h>
    #include <stdio.h>
    #include <errno.h>
    #include <unistd.h>
```

```
#defineBACKLOG1024      /* max # pending connections */

void do_work(int sock);

int
myserver3(char *servicename)
{
    struct addrinfo*aip;
    struct addrinfohints;
    struct sockaddr_storagefaddr;
    int         sock, new_sock, sock_opt;
    socklen_tfaddrlen;
    int         error;
    char        hname[NI_MAXHOST];
    char        sname[NI_MAXSERV];
    /*
     * Set up a socket to listen on for connections.
     */
    bzero(&hints, sizeof (hints));
    hints.ai_flags = AI_ALL|AI_ADDRCONFIG|AI_PASSIVE;
    hints.ai_socktype = SOCK_STREAM;

    error = getaddrinfo(NULL, servicename, &hints, &aip);
    if (error != 0) {
        (void) fprintf(stderr, "getaddrinfo: %s for service %s\n",
            gai_strerror(error), servicename);
        return (-1);
    }
    sock = socket(aip->ai_family, aip->ai_socktype, aip->ai_protocol);
    if (sock == -1) {
        perror("socket");
        return (-1);
    }

    /*
     * Tell the system to allow local addresses to be reused.
     */
    sock_opt = 1;

    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (void *)&sock_opt,
        sizeof (sock_opt)) == -1) {
        perror("setsockopt(SO_REUSEADDR)");
        (void) close(sock);
        return (-1);
    }

void
do_work(int sock)
```

```
{
    char    buf[BUFSIZ];
    int     cc;

    while (1) {
        cc = read(sock, buf, sizeof (buf));
        if (cc == -1) {
            perror("read");
            exit(1);
        }
        if (cc == 0) {
            /* EOF */
            (void) close(sock);
            (void) printf("Connection closed\n");
            return;
        }
        buf[cc + 1] = '\0';
        (void) printf("Read <%s>\n", buf);

        if (write(sock, buf, cc) == -1) {
            perror("write");
            exit(1);
        }
    }
}

main(int argc, char *argv[])
{
    if (argc != 2) {
        (void) fprintf(stderr,
            "Usage: %s <servicename>\n", argv[0]);
        exit(1);
    }
    (void) myserver3(argv[1]);
    return (0);
}
```

The preceding example demonstrates two protocol specific sockets, however, a programmer would not normally use this approach. Typically, a programmer would use a single socket call and let the dual-stack software manage the use of both IPv4 and IPv6.

CHAPTER 6

Tools to Help Upgrade Applications

To make the upgrade from IPv4 to IPv6 as simple as possible, Sun has developed conversion tools, the most important of which is Sun's *Socket Scrubber*. It can be run on a series of existing source files to help identify the most likely places that would require code changes to upgrade from IPv4 to IPv6 calls. Using this tool, programmers can scan existing applications and identify changes required to communicate using the IPv6 protocol.

Socket Scrubber for C/C++

Virtually all applications today comprise a series of modular subroutine and library files combined into a program or series of programs. Most of these modules have nothing to do with communicating over TCP/IP. To easily upgrade the application to communicate over IPv6, developers need to locate those modules that actually call the IP routines. Sun's Socket Scrubber can help identify those modules.

A command procedure can be created to scan all of the modules of a major application. This procedure runs in background or batch mode to scan all source files at one time. At the completion of the batch run, the resulting log file identifies all modules, and even lines within modules, that require modification.

While every application is different, and the upgrade effort will vary accordingly, the Socket Scrubber typically identifies 10 or so likely lines of code to change. These are lines that specify parameters for the API as well as the routine calls themselves.

Socket Scrubber Example

The Socket Scrubber searches for a predefined list of keyword variables within the code module. These keywords include all of the key IPv4 parameters and routine calls. When a keyword is located, the relevant code section is displayed to the developer. Once all of these keywords are located, the developer has a list of likely locations to edit.

Below is example output from the Socket Scrubber. In this sample, the program has been run against the IPv4-based sample program `myconnect.c` from the *IPv4 versus IPv6 Naming API Differences* section of Chapter 4, "Porting Client Code to IPv6".

```
****************************
myconnect.c
****************************
11: struct sockaddr_insin;
18: sock = socket(AF_INET, SOCK_STREAM, 0);
27: hp = gethostbyname(hostname);
28: if (hp == NULL || hp->h_addrtype != AF_INET || hp->h_length != 4) {
34: sin.sin_family = AF_INET;
35: sin.sin_port = htons(port);
36: (void) memcpy((void *)&sin.sin_addr, (void *)hp->h_addr,
```

Because the Socket Scrubber uses a keyword search, it is clear that the best results occur in applications that are programmed using reasonable coding standards. If application developers use such tricks as hard-coding variable contents for parameters rather than using the defined constants (such as using “`2`” instead of “`AF_INET`”), the Socket Scrubber will be less successful in identifying areas for change. Modular coding guidelines should isolate network-specific code in isolated network specific modules rather than mixed into general application code. These routines help make any coding modification much more predictable.

CHAPTER 7

Conclusion

There's no doubt that IPv4 is going to be replaced by IPv6. The only question is when. With the rapid growth of the Internet, the pool of available addresses is quickly shrinking. And while tools like NAT perform a valuable stopgap role, the real solution is a new IP address standard that will increase the number of available addresses and allow easier configuration.

Software developers need to begin focusing on protocol-independent, IPv6 development to ensure that their applications are viable for the future. All new applications should be developed using the new interfaces to support IPv6 — even if current networks are still IPv4 based. Using these new interfaces will also provide IPv4 communication support in the interim.

The key advantage is that once customers begin to fully implement IPv6 networks, resulting applications will automatically switch over.

Existing applications may be upgraded to use the new IPv6 interface as time and upgrade requirements warrant. By using Sun's Socket Scrubber, application developers can identify potential areas that need upgrading. This conversion from IPv4 to IPv6 is made easier because there is a direct relationship from IPv4 calls to corresponding IPv6 calls.

Sun has the tools and the expertise to help customers prepare and transition to IPv6. Sun's Solaris Operating Environment is the ideal platform for this transition. With the largest installed base in both technical and commercial UNIX®, the Solaris 8 Operating Environment includes a dual-stack name service, as well as libraries to port code immediately. With over 15 years of network experience, and an equally successful history of innovation, Sun provides the developer community with the tools and technologies to make the conversion to IPv6 easy and straightforward.

CHAPTER 8

Appendix

Socket Changes from IPv4 to IPv6

The socket API changes, as well as new features, are documented in two IETF documents:

- RFC 2553 – Basic socket interface extensions for IPv6
- Work in progress – Advanced sockets API for IPv6 (RFC 2292)[1]

Some of extensions to the socket API are independent of IPv6 because they merely provide a better method for handling different address families and lengths of addresses. Other extensions give access to IPv6-specific features such a source routing.

An additional key document is RFC 1933, which helps customers decide how to deploy the IPv6 environment.

Most of the changes in the socket API, when combined with the use of IPv4-mapped addresses, allow a straightforward change to make an application IPv6-aware.

1. Advanced sockets API for IPv6 RFC 2292 is currently being reworked within the IETF. 90% of socket-based applications will only need the IPv6 extensions specified in RFC 2553 (basic socket interface extensions for IPv6).

Feature	Currently in IPv4	Dual IPv4/IPv6	Comments
Protocol family First argument in socket(3N)	AF_INET/PF_INET	AF_INET6/PF_INET6	Address and protocol families are used interchangeably
Address family	AF_INET	AF_INET6	In sockaddr family field
Inet address structure	struct sockaddr_in	struct sockaddr_in6	Note: sockadd r_in6 is larger than a sockaddr
Generic address structure	struct sockaddr	struct sockaddr_storage	Only when used to allocate storage
IP address structure	struct in_addr	struct in6_addr	
Loopback address	INADDR_LOOPBACK	in6addr_loopback IN6ADDR_LOOPBACK_IN IT	Constant can only be used for structure initialization
Wildcard address for binding listeners and receivers	INADDR_ANY	in6addr_any IN6ADDR_ANY_INIT	Constant can only be used for structure initialization
Name to address	gethostbyname(3N)	getipnodebyname(3N)	Retrieve either IPv6 or IPv4 address
Address to name	gethostbyaddr(3N)	getipnodebyaddr(3X)	Function already has a family parameter
Free data structure returned by getipnodeby*	—	freehostent(3N)	getipnodeby* are multithread safe
Address to name	—	getaddrinfo(3N)	Isolate application from address formats
Free data structures returned by gettaddrinfo	—	freeaddrinfo(3N)	getaddrinfo is multithread safe
Name to address	—	getnameinfo(3N)	Isolate applications from address formats
Report errors	—	gai_strerror(3N)	Errors from getaddrinfo and getnameinfo
String to address	inet_addr(3N)	inet_pton(3N)	Added family parameter
Address to string	inet_ntoa(3N) — —	inet_ntop(3N) INET_ADDSTRLEN INET6_ADDSTRLEN	Added family parameter. Constants for maximum string buffer size
Socket option for TTL	IP_TTL	IPV6_UNICAST_HOPS	Set ttl/hop limit
Get reserved port	rresvport(3N)	rresvport_af(3N)	Used by rcmd(3N)
Execute a command on a remote host	rcmd(3N)	rcmd_af(3N)	Takes a address family parameter

TABLE 1 Corresponding IPv6/dual structure and constants

Differences Between IPv4 and IPv6

Some features of IPv4 are not present in IPv6. For example the features that map directly to the IPv4 packet format and the IPv4 protocol options are not present:

- The IPv4 `Type of Service` and precedence fields have been replaced by the `Traffic Class` field in IPv6.
- There is no record route option in IPv6.

Other features are still present, but have a very different interface or have already been removed in the IPv4 architecture, for example, the macros and functions which handle network numbers in IPv4 (see TABLE 2). With the gradual introduction of first subnets [RFC 950] and then classless inter-domain routing (CIDR), the IPv4 Internet is currently no longer relying on the concept of network numbers and different classes of IP addresses. Instead, all routing is done with arbitrary prefixes of IP addresses. IPv6 builds on the class-less routing developed in IPv4, so there is no notion of class or network numbers in IPv6.

Feature	**IPv4 interface**	**Comments**

Classfull addresses	IN_CLASS_A,B,C
	inet_makeaddr(3N)
	inet_network(3N)
	inet_lnaof(3N)
	inet_netof(3N)
		No notion of address classes in IPv6 (except unicast vs. multicast distinction)

Access to IP protocol fields	IP_OPTIONS socket option IP_RECVDSTADDR IP_RECVOPTS	Replaced by source routing and options functions
Access to IP protocol fields	IP_HDRINCL socket option	No longer needed. The IPV6_ socket options allow the application to set and retrieve every IPv6 option and extension header.
Access to IP protocol fields	IP_TOS socket option	The Type Of Service field has been replaced by the Traffic Class field in IPv6 which can be set using the sin6flowinfo field.

TABLE 2 IPv4 socket features with no IPv6 counterparts

CHAPTER **9**

References

RFC 1886 — DNS Extensions to Support IP Version 6, S. Thomson, Bellcore, C. Huitema, INRIA, December 1995

RFC 1933 — Transition Mechanisms for IPv6 Hosts and Routers, R. Gilligan, E. Nordmark, Sun Microsystems, April 1996

*R*RFC 2373 — IP Version 6 Addressing Architecture, R. Hinden, Nokia, S. Deering, Cisco Systems, July 1998

RFC 2460 — Internet Protocol, Version 6 (IPv6) Specification, S. Deering, Cisco, R. Hinden, Nokia, December 1998

RFC 2461 — Neighbor Discovery for IP Version 6 (IPv6), T. Narten, IBM, E. Nordmark, Sun Microsystems, W. Simpson, Daydreamer, December 1998

RFC 2462 — IPv6 Stateless Address Autoconfiguration, S. Thomson, Bellcore and T. Narten, IBM, December 1998

RFC 2463 — Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification, A. Conta, Lucent, S. Deering, Cisco Systems, December 1998

RFC 2553 — Basic Socket Interface Extensions for IPv6, R. Gilligan, FreeGate, S. Thomson, IBM, J. Bound, Compaq, W. Stevens, Consultant, March 1999

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303

1 (800) 786.7638
1.512.434.1511

http://www.sun.com/solaris/

November 1999