12.1 Introduction

Over the coming years, there will probably be a gradual transition of the Internet from IPv4 to IPv6. During this transition phase, it is important that existing IPv4 applications continue to work with newer IPv6 applications. For example, a vendor cannot provide a `telnet` client that works only with IPv6 `telnet` servers but must provide one that works with IPv4 servers and one that works with IPv6 servers. Better yet would be one IPv6 `telnet` client that can work with both IPv4 and IPv6 servers, along with one `telnet` server that can work with both IPv4 and IPv6 clients. We will see how this is done in this chapter.

We assume throughout this chapter that the hosts are running *dual stacks*, that is, both an IPv4 protocol stack and an IPv6 protocol stack. Our example in Figure 2.1 is a dual-stack host. Hosts and routers will probably run like this for many years into the transition to IPv6. At some point, many systems will be able to turn off their IPv4 stack, but only time will tell when (and if) that will occur.

In this chapter, we will discuss how IPv4 applications and IPv6 applications can communicate with each other. There are four combinations of clients and servers using either IPv4 or IPv6 and we show these in Figure 12.1.

Figure 12.1. Combinations of clients and servers using IPv4 or IPv6.

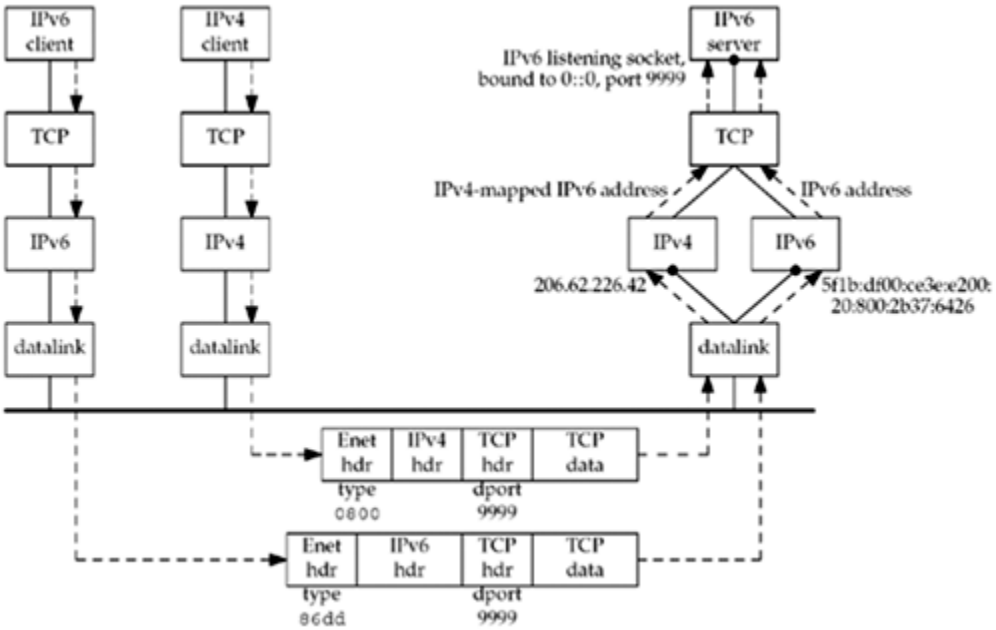
	IPv4 server	IPv6 server
IPv4 client	Almost all existing clients and servers	Discussed in Section 12.2
IPv6 client	Discussed in Section 12.3	Simple modifications to most existing clients and servers (e.g., Figure 1.5 to Figure 1.6)

We will not say much more about the two scenarios where the client and server use the same protocol. The interesting cases are when the client and server use different protocols.

12.2 IPv4 Client, IPv6 Server

A general property of a dual-stack host is that IPv6 servers can handle both IPv4 and IPv6 clients. This is done using IPv4-mapped IPv6 addresses (Figure A.10). Figure 12.2 shows an example of this.

Figure 12.2. IPv6 server on dual-stack host serving IPv4 and IPv6 clients.



We have an IPv4 client and an IPv6 client on the left. The server on the right is written using IPv6 and it is running on a dual-stack host. The server has created an IPv6 listening TCP socket that is bound to the IPv6 wildcard address and TCP port 9999.

We assume the clients and server are on the same Ethernet. They could also be connected by routers, as long as all the routers support IPv4 and IPv6, but that adds nothing to this discussion. Section B.3 discusses a different case where IPv6 clients and servers are connected by IPv4-only routers.

We assume both clients send SYN segments to establish a connection with the server. The IPv4 client host will send the SYN in an IPv4 datagram and the IPv6 client host will send the SYN in an IPv6 datagram. The TCP segment from the IPv4 client appears on the wire as an Ethernet header followed by an IPv4 header, a TCP header, and the TCP data. The Ethernet header contains a type field of `0x0800`, which identifies the frame as an IPv4 frame. The TCP header contains the destination port of 9999. (Appendix A talks more about the formats and contents of these headers.) The destination IP address in the IPv4 header, which we do not show, would be 206.62.226.42.

The TCP segment from the IPv6 client appears on the wire as an Ethernet header followed by an IPv6 header, a TCP header, and the TCP data. The Ethernet header contains a type field of `0x86dd`, which identifies the frame as an IPv6 frame. The TCP header has the same format as the TCP header in the IPv4 packet and contains the destination port of 9999. The destination IP address in the IPv6 header, which we do not show, would be `5f1b:df00:ce3e:e200:20:800:2b37:6426`.

The receiving datalink looks at the Ethernet type field and passes each frame to the appropriate IP module. The IPv4 module, probably in conjunction with the TCP module, detects that the destination socket is an IPv6 socket, and the source IPv4 address in the IPv4 header is converted into the equivalent IPv4-mapped IPv6 address. That mapped address is returned to the IPv6 socket as the client's IPv6 address when `accept` returns to the server with the IPv4 client connection. All remaining datagrams for this connection are IPv4 datagrams.

When `accept` returns to the server with the IPv6 client connection, the client's IPv6 address does not change from whatever source address appears in the IPv6 header. All remaining datagrams for this connection are IPv6 datagrams.

We can summarize the steps that allow an IPv4 TCP client to communicate with an IPv6 server as follows:

1. The IPv6 server starts, creates an IPv6 listening socket, and we assume it `binds` the wildcard address to the socket.
2. The IPv4 client calls `gethostbyname` and finds an A record for the server. The server host will have both an A record and a AAAA record since it supports both protocols, but the IPv4 client asks for only an A record.
3. The client calls `connect` and the client's host sends an IPv4 SYN to the server.
4. The server host receives the IPv4 SYN directed to the IPv6 listening socket, sets a flag indicating that this connection is using IPv4-mapped IPv6 addresses, and responds with an IPv4 SYN/ACK. When the connection is established, the address returned to the server

by `accept` is the IPv4-mapped IPv6 address.

5. When the server host sends to the IPv4-mapped IPv6 address, its IP stack generates IPv4 datagrams to the IPv4 address. Therefore, all communication between this client and server takes place using IPv4 datagrams.

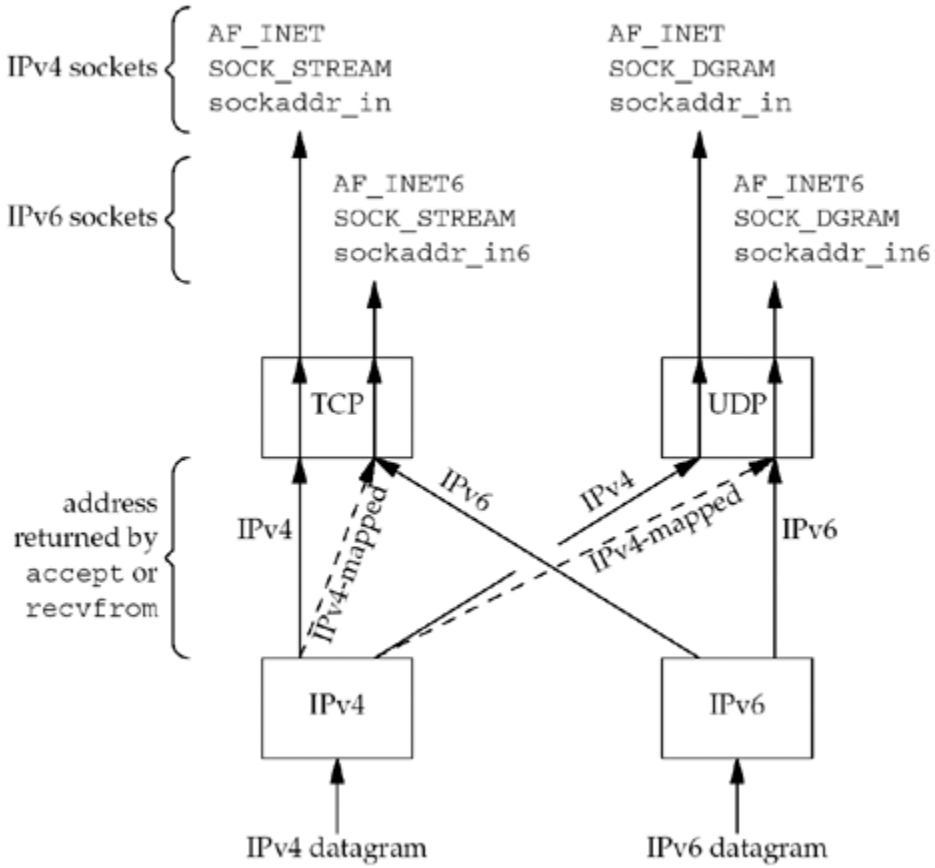
6. Unless the server explicitly checks whether this IPv6 address is an IPv4-mapped IPv6 address (using the `IN6_IS_ADDR_V4MAPPED` macro described in Section 12.4), the server never knows that it is communicating with an IPv4 client. The dual-protocol stack handles this detail. Similarly, the IPv4 client has no idea that it is communicating with an IPv6 server.

An underlying assumption in this scenario is that the dual-stack server host has both an IPv4 address and an IPv6 address. This will work until all the IPv4 addresses are taken.

The scenario is similar for an IPv6 UDP server, but the address format can change for each datagram. For example, if the IPv6 server receives a datagram from an IPv4 client, the address returned by `recvfrom` will be the client's IPv4-mapped IPv6 address. The server responds to this client's request by calling `sendto` with the IPv4-mapped IPv6 address as the destination. This address format tells the kernel to send an IPv4 datagram to the client. But the next datagram received for the server could be an IPv6 datagram, and `recvfrom` will return the IPv6 address. If the server responds, the kernel will generate an IPv6 datagram.

Figure 12.3 summarizes how a received IPv4 or IPv6 datagram is processed, depending on the type of the receiving socket, for TCP and UDP, assuming a dual-stack host.

Figure 12.3. Processing of received IPv4 or IPv6 datagrams, depending on type of receiving socket.



- If an IPv4 datagram is received for an IPv4 socket, nothing special is done. These are the two arrows labeled "IPv4" in the figure: one to TCP and one to UDP. IPv4 datagrams are exchanged between the client and server.
- If an IPv6 datagram is received for an IPv6 socket, nothing special is done. These are the two arrows labeled "IPv6" in the figure: one to TCP and one to UDP. IPv6 datagrams are exchanged between the client and server.
- When an IPv4 datagram is received for an IPv6 socket, the kernel returns the corresponding IPv4-mapped IPv6 address as the address returned by `accept` (TCP) or `recvfrom` (UDP). These are the two dashed arrows in the figure. This mapping is possible because an IPv4 address can always be represented as an IPv6 address. IPv4 datagrams are exchanged between the client and server.
- The converse of the previous bullet is false: In general, an IPv6 address cannot be represented as an IPv4 address; therefore, there are no arrows from the IPv6 protocol box to the two IPv4 sockets

Most dual-stack hosts should use the following rules in dealing with listening sockets:

1. A listening IPv4 socket can accept incoming connections from only IPv4 clients.
2. If a server has a listening IPv6 socket that has bound the wildcard address and the `IPV6_V6ONLY` socket option (Section 7.8) is not set, that socket can accept incoming connections from either IPv4 clients or IPv6 clients. For a connection from an IPv4 client, the server's local address for the connection will be the corresponding IPv4-mapped IPv6 address.
3. If a server has a listening IPv6 socket that has bound an IPv6 address other than an IPv4-mapped IPv6 address, or has bound the wildcard address but has set the `IPv6_V6ONLY` socket option (Section 7.8), that socket can accept incoming connections from IPv6 clients only.

[Team LiB]

PREVIOUS NEXT

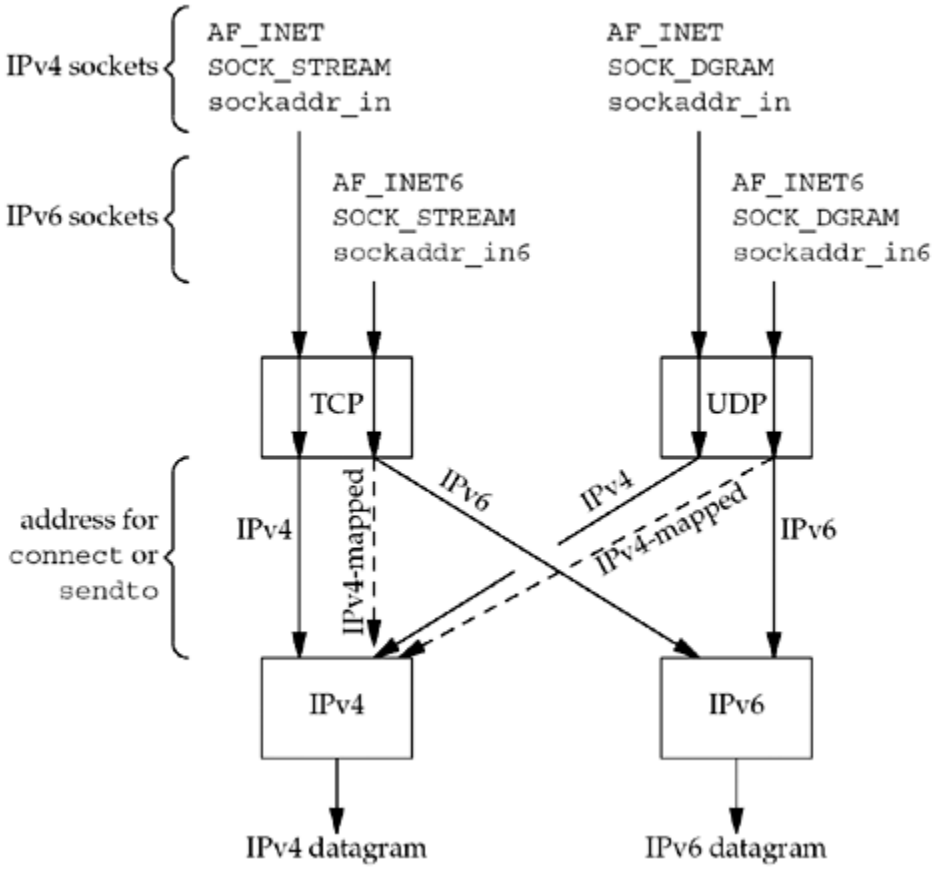
12.3 IPv6 Client, IPv4 Server

We now swap the protocols used by the client and server from the example in the previous section. First consider an IPv6 TCP client running on a dual-stack host.

1. An IPv4 server starts on an IPv4-only host and creates an IPv4 listening socket.
2. The IPv6 client starts and calls `getaddrinfo` asking for only IPv6 addresses (it requests the `AF_INET6` address family and sets the `AI_V4MAPPED` flag in its *hints* structure). Since the IPv4-only server host has only A records, we see from Figure 11.8 that an IPV4-mapped IPv6 address is returned to the client.
3. The IPv6 client calls `connect` with the IPv4-mapped IPv6 address in the IPv6 socket address structure. The kernel detects the mapped address and automatically sends an IPv4 SYN to the server.
4. The server responds with an IPv4 SYN/ACK, and the connection is established using IPv4 datagrams.

We can summarize this scenario in Figure 12.4.

Figure 12.4. Processing of client requests, depending on address type and socket type.



- If an IPv4 TCP client calls `connect` specifying an IPv4 address, or if an IPv4 UDP client calls `sendto` specifying an IPv4 address, nothing special is done. These are the two arrows labeled "IPv4" in the figure.
- If an IPv6 TCP client calls `connect` specifying an IPv6 address, or if an IPv6 UDP client calls `sendto` specifying an IPv6 address, nothing special is done. These are the two arrows labeled "IPv6" in the figure.
- If an IPv6 TCP client specifies an IPv4-mapped IPv6 address to `connect` or if an IPv6 UDP client specifies an IPv4-mapped IPv6 address to `sendto`, the kernel detects the mapped address and causes an IPv4 datagram to be sent instead of an IPv6 datagram. These are the two dashed arrows in the figure.
- An IPv4 client cannot specify an IPv6 address to either `connect` or `sendto` because a 16-byte IPv6 address does not fit in the 4-byte `in_addr` structure within the IPv4 `sockaddr_in` structure. Therefore, there are no arrows from the IPv4 sockets to the IPv6 protocol box in the figure.

In the previous section (an IPv4 datagram arriving for an IPv6 server socket), the conversion of the received address to the IPv4-mapped IPv6 address is done by the kernel and returned transparently to the application by `accept` or `recvfrom`. In this section (an IPv4 datagram needing to be sent on an IPv6 socket), the conversion of the IPv4 address to the IPv4-mapped IPv6 address is done by the resolver according to the rules in Figure 11.8, and the mapped address is then passed transparently by the application to `connect` or `sendto`.

Summary of Interoperability

Figure 12.5 summarizes this section and the previous section, plus the combinations of clients and servers.

Figure 12.5. Summary of interoperability between IPv4 and IPv6 clients and servers.

	IPv4 server IPv4-only host (A only)	IPv6 server IPv6-only host (AAAA only)	IPv4 server dual-stack host (A and AAAA)	IPv6 server dual-stack host (A and AAAA)
IPv4 client, IPv4-only host	IPv4	(no)	IPv4	IPv4
IPv6 client, IPv6-only host	(no)	IPv6	(no)	IPv6
IPv4 client, dual-stack host	IPv4	(no)	IPv4	IPv4
IPv6 client, dual-stack host	IPv4	IPv6	(no*)	IPv6

Each box contains "IPv4" or "IPv6" if the combination is okay, indicating which protocol is used, or "(no)" if the combination is invalid. The third column on the final row is marked with an asterisk because interoperability depends on the address chosen by the client. Choosing the AAAA record and sending an IPv6 datagram will not work. But choosing the A record, which is returned to the client as an IPv4-mapped IPv6 address, causes an IPv4 datagram to be sent, which will work. By looping through all adresses that `getaddrinfo` returns, as shown in Figure 11.4, we can ensure that we will (perhaps after some timeouts) try the IPv4-mapped IPv6 address.

Although it appears that five entries in the table will not interoperate, in the real world for the foreseeable future, most implementations of IPv6 will be on dual-stack hosts and will not be IPv6-only implementations. If we therefore remove the second row and the second column, all of the "(no)" entries disappear and the only problem is the entry with the asterisk.

12.4 IPv6 Address-Testing Macros

There is a small class of IPv6 applications that must know whether they are talking to an IPv4 peer. These applications need to know if the peer's address is an IPv4-mapped IPv6 address. The following 12 macros are defined to test an IPv6 address for certain properties.

```
#include <netinet/in.h>

int IN6_IS_ADDR_UNSPECIFIED(const struct in6_addr *aptr);

int IN6_IS_ADDR_LOOPBACK(const struct in6_addr *aptr);

int IN6_IS_ADDR_MULTICAST(const struct in6_addr *aptr);

int IN6_IS_ADDR_LINKLOCAL(const struct in6_addr *aptr);

int IN6_IS_ADDR_SITELOCAL(const struct in6_addr *aptr);

int IN6_IS_ADDR_V4MAPPED(const struct in6_addr *aptr);

int IN6_IS_ADDR_V4COMPAT(const struct in6_addr *aptr);

int IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr *aptr);

int IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr *aptr);

int IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr *aptr);

int IN6_IS_ADDR_MC_ORGLOCAL(const struct in6_addr *aptr);

int IN6_IS_ADDR_MC_GLOBAL(const struct in6_addr *aptr);
```

All return: nonzero if IPv6 address is of specified type, zero otherwise

The first seven macros test the basic type of IPv6 address. We show these various address types in Section A.5. The final five macros test the scope of an IPv6 multicast address (Section 21.2).

> IPv4-compatible addresses are used by a transition mechanism that has fallen out of favor. You're not likely to actually see this type of address or need to test for it.

An IPv6 client could call the `IN6_IS_ADDR_V4MAPPED` macro to test the IPv6 address returned by the resolver. An IPv6 server could call this macro to test the IPv6 address returned by `accept` or `recvfrom`.

As an example of an application that needs this macro, consider FTP and its `PORT` command. If we start an FTP client, log in to an FTP server, and issue an FTP `dir` command, the FTP client sends a `PORT` command to the FTP server across the control connection. This tells the server the client's IP address and port, to which the server then creates a data connection. (Chapter 27 of TCPv1 contains all the details of the FTP application protocol.) But, an IPv6 FTP client must know whether the server is an IPv4 server or an IPv6 server, because the former requires a command of the form `PORT a1,a2,a3,a4,p1,p2` where the first four numbers (each between 0 and 255) form the 4-byte IPv4 address and the last two numbers form the 2-byte port number. An IPv6 server, however, requires an `EPRT` command (RFC 2428 [Allman, Ostermann, and Metz 1998]), containing an address family, text format address, and text format port. Exercise 12.1 gives an example of IPv4 and IPv6 FTP protocol behavior.

12.5 Source Code Portability

Most existing network applications are written assuming IPv4. `sockaddr_in` structures are allocated and filled in and the calls to `socket` specify `AF_INET` as the first argument. We saw in the conversion from Figure 1.5 to Figure 1.6 that these IPv4 applications could be converted to use IPv6 without too much effort. Many of the changes that we showed could be done automatically using some editing scripts. Programs that are more dependent on IPv4, using features such as multicasting, IP options, or raw sockets, will take more work to convert.

If we convert an application to use IPv6 and distribute it in source code, we now have to worry about whether or not the recipient's system supports IPv6. The typical way to handle this is with `#ifdefs` throughout the code, using IPv6 when possible (since we have seen in this chapter that an IPv6 client can still communicate with IPv4 servers, and vice versa). The problem with this approach is that the code becomes littered with `#ifdefs` very quickly, and is harder to follow and maintain.

A better approach is to consider the move to IPv6 as a chance to make the program protocol-independent. The first step is to remove calls to `gethostbyname` and `gethostbyaddr` and use the `getaddrinfo` and `getnameinfo` functions that we described in the previous chapter. This lets us deal with socket address structures as opaque objects, referenced by a pointer and size, which is exactly what the basic socket functions do: `bind`, `connect`, `recvfrom`, and so on. Our `sock_`*XXX* functions from Section 3.8 can help manipulate these, independent of IPv4 or IPv6. Obviously these functions contain `#ifdefs` to handle IPv4 and IPv6, but hiding all of this protocol dependency in a few library functions makes our code simpler. We will develop a set of `mcast_`*XXX* functions in Section 21.7 that can make multicast applications independent of IPv4 or IPv6.

Another point to consider is what happens if we compile our source code on a system that supports both IPv4 and IPv6, distribute either executable code or object files (but not the source code), and someone runs our application on a system that does not support IPv6? There is a chance that the local name server supports AAAA records and returns both AAAA records and A records for some peer with which our application tries to connect. If our application, which is IPv6-capable, calls `socket` to create an IPv6 socket, it will fail if the host does not support IPv6. We handle this in the helper functions described in the previous chapter by ignoring the error from `socket` and trying the next address on the list returned by the name server. Assuming the peer has an A record, and that the name server returns the A record in addition to any AAAA records, the creation of an IPv4 socket will succeed. This is the type of functionality that belongs in a library function, and not in the source code of every application.

To enable passing socket descriptors to programs that were IPv4-only or IPv6-only, RFC 2133 [Gilligan et al. 1997] introduced the `IPV6_ADDRFORM` socket option, which could return or potentially change the address family associated with a socket. However, the semantics were never completely described, and it was only useful in very specific cases, so it was removed in the next revision of the API.

12.6 Summary

An IPv6 server on a dual-stack host can service both IPv4 clients and IPv6 clients. An IPv4 client still sends IPv4 datagrams to the server, but the server's protocol stack converts the client's address into an IPv4-mapped IPv6 address since the IPv6 server is dealing with IPv6 socket address structures.

Similarly, an IPv6 client on a dual-stack host can communicate with an IPv4 server. The client's resolver will return IPv4-mapped IPv6 addresses for all the server's A records, and calling `connect` for one of these addresses results in the dual stack sending an IPv4 SYN segment. Only a few special clients and servers need to know the protocol being used by the peer (e.g., FTP) and the `IN6_IS_ADDR_V4MAPPED` macro can be used to see if the peer is using IPv4.

[Team LiB]

PREVIOUS NEXT