the second bind unless the user ID is the same as that of the process that has already bound the port.

11.10 At the end of Section 2.12, we showed two `telnet` examples: to the daytime server and to the echo server. Knowing that a client goes through the two steps `gethostbyname` and `connect`, which lines output by the client indicate which steps?

11.11 `getnameinfo` can take a long time (up to 80 seconds) to return an error if a hostname cannot be found for an IP address. Write a new function named `getnameinfo_timeo` that takes an additional integer argument specifying the maximum number of seconds to wait for a reply. If the timer expires and the `NI_NAMEREQD` flag is not specified, just call `inet_ntop` and return an address string.

Part 3: Advanced Sockets

Chapter 12. IPv4 and IPv6 Interoperability

Chapter 13. Daemon Processes and the `inetd` Superserver

Chapter 14. Advanced I/O Functions

Chapter 15. Unix Domain Protocols

Chapter 16. Nonblocking I/O

Chapter 17. `ioctl` Operations

Chapter 18. Routing Sockets

Chapter 19. Key Management Sockets

Chapter 20. Broadcasting

Chapter 21. Multicasting

Chapter 22. Advanced UDP Sockets

Chapter 23. Advanced SCTP Sockets

Chapter 24. Out-of-Band Data

Chapter 25. Signal-Driven I/O

Chapter 26. Threads

Chapter 27. IP Options

Chapter 28. Raw Sockets

Chapter 29. Datalink Access

Chapter 30. Client/Server Design Alternatives

Chapter 31. Streams

Chapter 12. IPv4 and IPv6 Interoperability

Section 12.1. Introduction

Section 12.2. IPv4 Client, IPv6 Server

Section 12.3. IPv6 Client, IPv4 Server

Section 12.4. IPv6 Address-Testing Macros

Section 12.5. Source Code Portability

Section 12.6. Summary

Exercises

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.

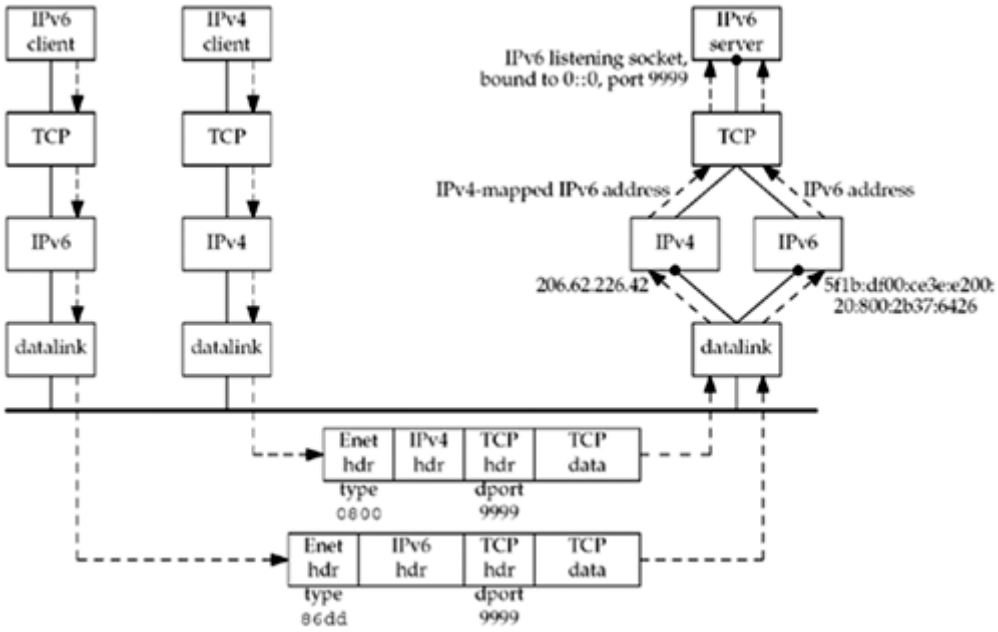
	IPv4 server	IPv6 server
IPv4 client	Almost all existing clients and servers	Discussed in Section 12.2
IPv6 client	Discussed in Section 12.3	Simple modifications to most existing clients and servers (e.g., Figure 1.5 to Figure 1.6)

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:

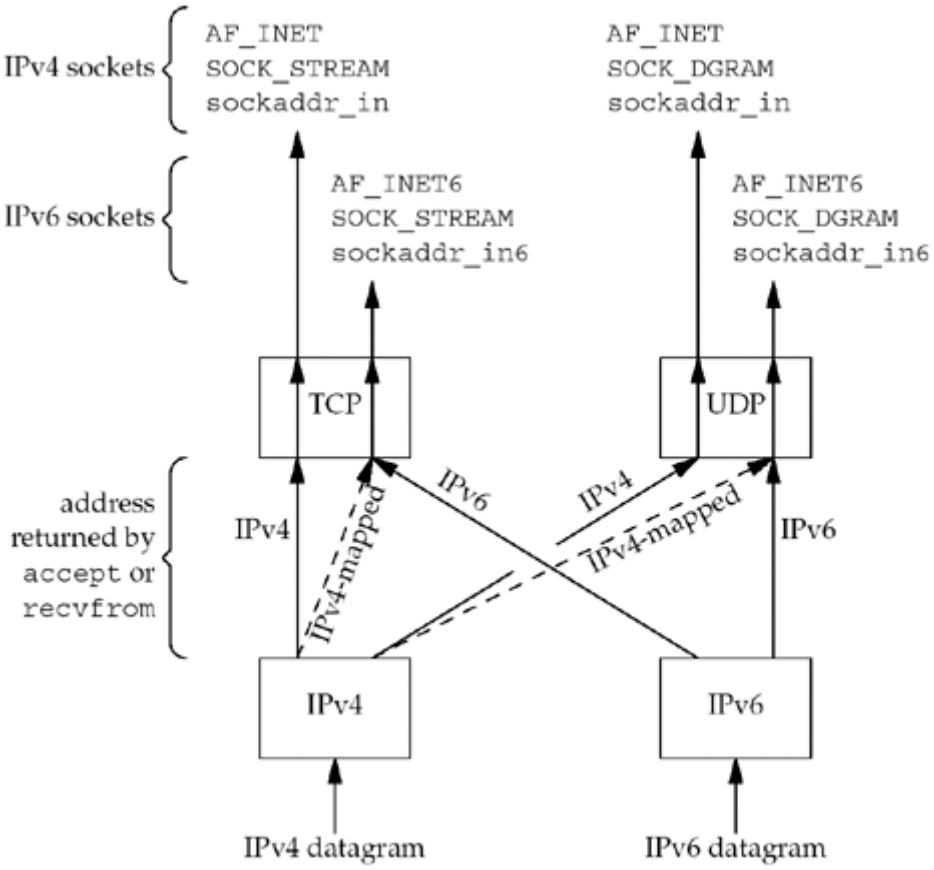
1. The IPv6 server starts, creates an IPv6 listening socket, and we assume it `binds` the wildcard address to the socket.
2. The IPv4 client calls `gethostbyname` and finds an A record for the server. The server host will have both an A record and a AAAA record since it supports both protocols, but the 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.

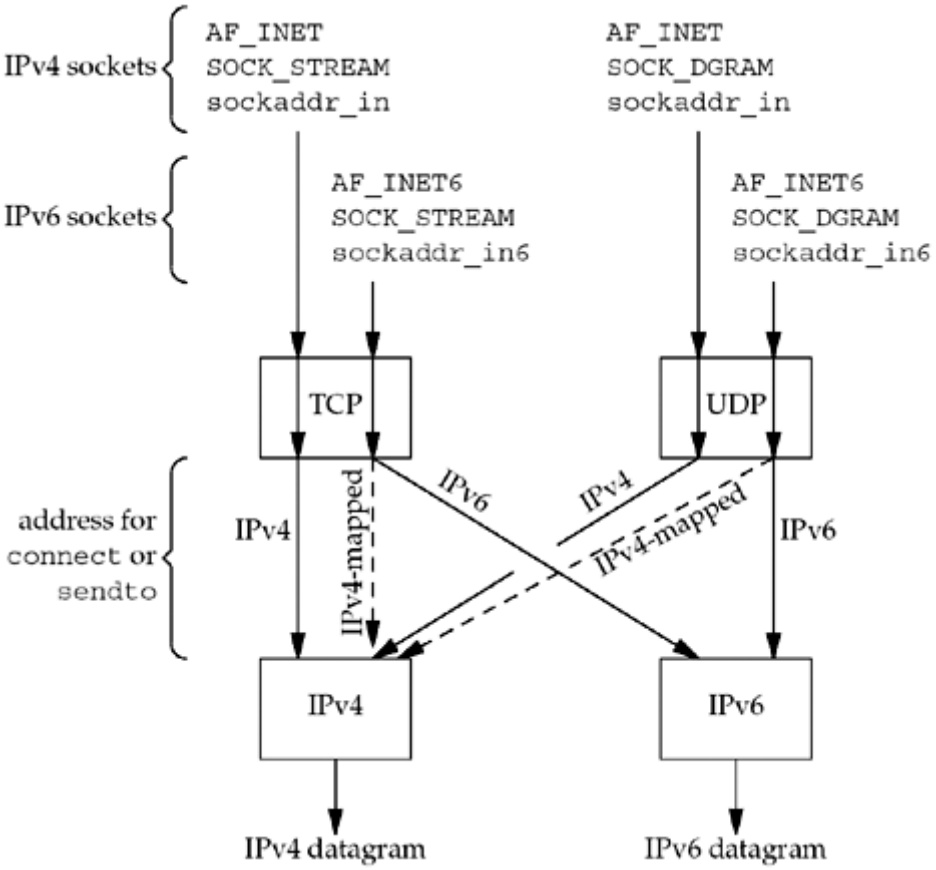
12.3 IPv6 Client, IPv4 Server

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);
```