THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX® System Programming Handbook

MICHAEL KERRISK

no starch press

59

SOCKETS: INTERNET DOMAINS

Having looked at generic sockets concepts and the TCP/IP protocol suite in previous chapters, we are now ready in this chapter to look at programming with sockets in the IPv4 (`AF_INET`) and IPv6 (`AF_INET6`) domains.

As noted in Chapter 58, Internet domain socket addresses consist of an IP address and a port number. Although computers use binary representations of IP addresses and port numbers, humans are much better at dealing with names than with numbers. Therefore, we describe the techniques used to identify host computers and ports using names. We also examine the use of library functions to obtain the IP address(es) for a particular hostname and the port number that corresponds to a particular service name. Our discussion of hostnames includes a description of the Domain Name System (DNS), which implements a distributed database that maps hostnames to IP addresses and vice versa.

59.1 Internet Domain Sockets

Internet domain stream sockets are implemented on top of TCP. They provide a reliable, bidirectional, byte-stream communication channel.

Internet domain datagram sockets are implemented on top of UDP. UDP sockets are similar to their UNIX domain counterparts, but note the following differences:

- UNIX domain datagram sockets are reliable, but UDP sockets are not—datagrams may be lost, duplicated, or arrive in a different order from that in which they were sent.
- Sending on a UNIX domain datagram socket will block if the queue of data for the receiving socket is full. By contrast, with UDP, if the incoming datagram would overflow the receiver's queue, then the datagram is silently dropped.

59.2 Network Byte Order

IP addresses and port numbers are integer values. One problem we encounter when passing these values across a network is that different hardware architectures store the bytes of a multibyte integer in different orders. As shown in Figure 59-1, architectures that store integers with the most significant byte first (i.e., at the lowest memory address) are termed *big endian*; those that store the least significant byte first are termed *little endian*. (The terms derive from Jonathan Swift's 1726 satirical novel *Gulliver's Travels*, in which the terms refer to opposing political factions who open their boiled eggs at opposite ends.) The most notable example of a little-endian architecture is x86. (Digital's VAX architecture was another historically important example, since BSD was widely used on that machine.) Most other architectures are big endian. A few hardware architectures are switchable between the two formats. The byte ordering used on a particular machine is called the *host byte order*.

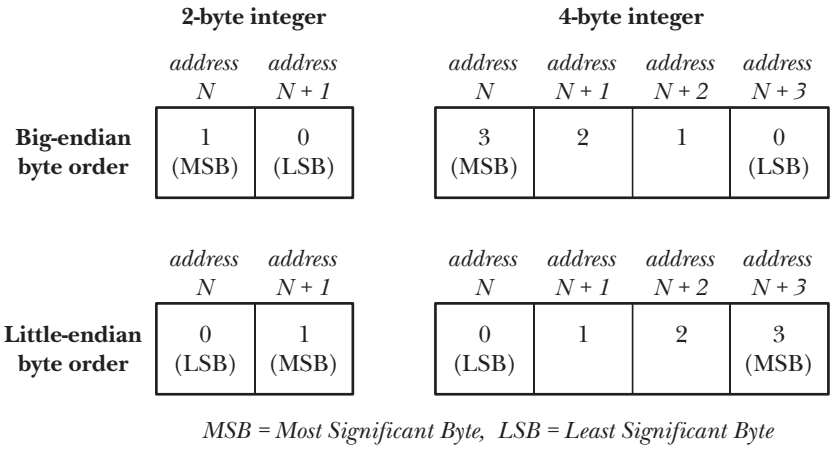

Figure 59-1: Big-endian and little-endian byte order for 2-byte and 4-byte integers

Since port numbers and IP addresses must be transmitted between, and understood by, all hosts on a network, a standard ordering must be used. This ordering is called *network byte order*, and happens to be big endian.

Later in this chapter, we look at various functions that convert hostnames (e.g., `www.kernel.org`) and service names (e.g., *http*) into the corresponding numeric forms. These functions generally return integers in network byte order, and these integers can be copied directly into the relevant fields of a socket address structure.

However, we sometimes make direct use of integer constants for IP addresses and port numbers. For example, we may choose to hard-code a port number into our program, specify a port number as a command-line argument to a program, or use constants such as `INADDR_ANY` and `INADDR_LOOPBACK` when specifying an IPv4 address. These values are represented in C according to the conventions of the host machine, so they are in host byte order. We must convert these values to network byte order before storing them in socket address structures.

The *htons()*, *htonl()*, *ntohs()*, and *ntohl()* functions are defined (typically as macros) for converting integers in either direction between host and network byte order.

```
#include <arpa/inet.h>

uint16_t htons(uint16_t host_uint16);
                                Returns host_uint16 converted to network byte order
uint32_t htonl(uint32_t host_uint32);
                                Returns host_uint32 converted to network byte order
uint16_t ntohs(uint16_t net_uint16);
                                Returns net_uint16 converted to host byte order
uint32_t ntohl(uint32_t net_uint32);
                                Returns net_uint32 converted to host byte order
```

In earlier times, these functions had prototypes such as the following:

```
unsigned long htonl(unsigned long hostlong);
```

This reveals the origin of the function names—in this case, *host to network long*. On most early systems on which sockets were implemented, short integers were 16 bits, and long integers were 32 bits. This no longer holds true on modern systems (at least for long integers), so the prototypes given above provide a more exact definition of the types dealt with by these functions, although the names remain unchanged. The *uint16_t* and *uint32_t* data types are 16-bit and 32-bit unsigned integers.

Strictly speaking, the use of these four functions is necessary only on systems where the host byte order differs from network byte order. However, these functions should always be used, so that programs are portable to different hardware architectures. On systems where the host byte order is the same as network byte order, these functions simply return their arguments unchanged.

59.3 Data Representation

When writing network programs, we need to be aware of the fact that different computer architectures use different conventions for representing various data types. We have already noted that integer types can be stored in big-endian or little-endian form. There are also other possible differences. For example, the C *long* data type may be 32 bits on some systems and 64 bits on others. When we consider structures, the issue is further complicated by the fact that different implementations

employ different rules for aligning the fields of a structure to address boundaries on the host system, leaving different numbers of padding bytes between the fields.

Because of these differences in data representation, applications that exchange data between heterogeneous systems over a network must adopt some common convention for encoding that data. The sender must encode data according to this convention, while the receiver decodes following the same convention. The process of putting data into a standard format for transmission across a network is referred to as *marshalling*. Various marshalling standards exist, such as XDR (External Data Representation, described in RFC 1014), ASN.1-BER (Abstract Syntax Notation 1, *http://www.asn1.org/*), CORBA, and XML. Typically, these standards define a fixed format for each data type (defining, for example, byte order and number of bits used). As well as being encoded in the required format, each data item is tagged with extra field(s) identifying its type (and, possibly, length).

However, a simpler approach than marshalling is often employed: encode all transmitted data in text form, with separate data items delimited by a designated character, typically a newline character. One advantage of this approach is that we can use *telnet* to debug an application. To do this, we use the following command:

```
$ telnet host port
```

We can then type lines of text to be transmitted to the application, and view the responses sent by the application. We demonstrate this technique in Section 59.11.

> The problems associated with differences in representation across heterogeneous systems apply not only to data transfer across a network, but also to any mechanism of data exchange between such systems. For example, we face the same problems when transferring files on disk or tape between heterogeneous systems. Network programming is simply the most common programming context in which we are nowadays likely to encounter this issue.

If we encode data transmitted on a stream socket as newline-delimited text, then it is convenient to define a function such as *readLine()*, shown in Listing 59-1.

```
#include "read_line.h"

ssize_t readLine(int fd, void *buffer, size_t n);
```

Returns number of bytes copied into *buffer* (excluding terminating null byte), or 0 on end-of-file, or −1 on error

The *readLine()* function reads bytes from the file referred to by the file descriptor argument *fd* until a newline is encountered. The input byte sequence is returned in the location pointed to by *buffer*, which must point to a region of at least *n* bytes of memory. The returned string is always null-terminated; thus, at most *(n − 1)* bytes of actual data will be returned. On success, *readLine()* returns the number of bytes of data placed in *buffer*; the terminating null byte is not included in this count.

Listing 59-1: Reading data a line at a time

sockets/read_line.c

```
#include <unistd.h>
#include <errno.h>
#include "read_line.h"                  /* Declaration of readLine() */

ssize_t
readLine(int fd, void *buffer, size_t n)
{
    ssize_t numRead;                    /* # of bytes fetched by last read() */
    size_t totRead;                     /* Total bytes read so far */
    char *buf;
    char ch;

    if (n <= 0 || buffer == NULL) {
        errno = EINVAL;
        return -1;
    }

    buf = buffer;                       /* No pointer arithmetic on "void *" */

    totRead = 0;
    for (;;) {
        numRead = read(fd, &ch, 1);

        if (numRead == -1) {
            if (errno == EINTR)         /* Interrupted --> restart read() */
                continue;
            else
                return -1;              /* Some other error */

        } else if (numRead == 0) {      /* EOF */
            if (totRead == 0)           /* No bytes read; return 0 */
                return 0;
            else                        /* Some bytes read; add '\0' */
                break;

        } else {                        /* 'numRead' must be 1 if we get here */
            if (totRead < n - 1) {      /* Discard > (n - 1) bytes */
                totRead++;
                *buf++ = ch;
            }

            if (ch == '\n')
                break;
        }
    }

    *buf = '\0';
    return totRead;
}
```

sockets/read_line.c

If the number of bytes read before a newline is encountered is greater than or equal to *(n − 1)*, then the *readLine()* function discards the excess bytes (including the newline). If a newline was read within the first *(n − 1)* bytes, then it is included in the returned string. (Thus, we can determine if bytes were discarded by checking if a newline precedes the terminating null byte in the returned *buffer*.) We take this approach so that application protocols that rely on handling input in units of lines don't end up processing a long line as though it were multiple lines. This would likely break the protocol, as the applications on either end would become desynchronized. An alternative approach would be to have *readLine()* read only sufficient bytes to fill the supplied buffer, leaving any remaining bytes up to the next newline for the next call to *readLine()*. In this case, the caller of *readLine()* would need to handle the possibility of a partial line being read.

We employ the *readLine()* function in the example programs presented in Section 59.11.

59.4 Internet Socket Addresses

There are two types of Internet domain socket addresses: IPv4 and IPv6.

IPv4 socket addresses: *struct sockaddr_in*

An IPv4 socket address is stored in a *sockaddr_in* structure, defined in `<netinet/in.h>` as follows:

```
struct in_addr {                    /* IPv4 4-byte address */
    in_addr_t s_addr;               /* Unsigned 32-bit integer */
};

struct sockaddr_in {                /* IPv4 socket address */
    sa_family_t    sin_family;      /* Address family (AF_INET) */
    in_port_t      sin_port;        /* Port number */
    struct in_addr sin_addr;        /* IPv4 address */
    unsigned char  __pad[X];        /* Pad to size of 'sockaddr'
                                       structure (16 bytes) */
};
```

In Section 56.4, we saw that the generic *sockaddr* structure commences with a field identifying the socket domain. This corresponds to the *sin_family* field in the *sockaddr_in* structure, which is always set to `AF_INET`. The *sin_port* and *sin_addr* fields are the port number and the IP address, both in network byte order. The *in_port_t* and *in_addr_t* data types are unsigned integer types, 16 and 32 bits in length, respectively.

IPv6 socket addresses: *struct sockaddr_in6*

Like an IPv4 address, an IPv6 socket address includes an IP address plus a port number. The difference is that an IPv6 address is 128 bits instead of 32 bits. An IPv6 socket address is stored in a *sockaddr_in6* structure, defined in `<netinet/in.h>` as follows:

```
struct in6_addr {                   /* IPv6 address structure */
    uint8_t s6_addr[16];            /* 16 bytes == 128 bits */
};
```

```
struct sockaddr_in6 {               /* IPv6 socket address */
    sa_family_t sin6_family;        /* Address family (AF_INET6) */
    in_port_t   sin6_port;          /* Port number */
    uint32_t    sin6_flowinfo;      /* IPv6 flow information */
    struct in6_addr sin6_addr;      /* IPv6 address */
    uint32_t    sin6_scope_id;      /* Scope ID (new in kernel 2.4) */
};
```

The *sin_family* field is set to AF_INET6. The *sin6_port* and *sin6_addr* fields are the port number and the IP address. (The *uint8_t* data type, used to type the bytes of the *in6_addr* structure, is an 8-bit unsigned integer.) The remaining fields, *sin6_flowinfo* and *sin6_scope_id*, are beyond the scope of this book; for our purposes, they are always set to 0. All of the fields in the *sockaddr_in6* structure are in network byte order.

> IPv6 addresses are described in RFC 4291. Information about IPv6 flow control (*sin6_flowinfo*) can be found in Appendix A of [Stevens et al., 2004] and in RFCs 2460 and 3697. RFCs 3493 and 4007 provide information about *sin6_scope_id*.

IPv6 has equivalents of the IPv4 wildcard and loopback addresses. However, their use is complicated by the fact that an IPv6 address is stored in an array (rather than using a scalar type). We use the IPv6 wildcard address (0::0) to illustrate this point. The constant IN6ADDR_ANY_INIT is defined for this address as follows:

```
#define IN6ADDR_ANY_INIT { { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 } }
```

> On Linux, some details in the header files differ from our description in this section. In particular, the *in6_addr* structure contains a union definition that divides the 128-bit IPv6 address into 16 bytes, eight 2-byte integers, or four 32-byte integers. Because of the presence of this definition, the *glibc* definition of the IN6ADDR_ANY_INIT constant actually includes one more set of nested braces than is shown in the main text.

We can use the IN6ADDR_ANY_INIT constant in the initializer that accompanies a variable declaration, but can't use it on the right-hand side of an assignment statement, since C syntax doesn't permit structured constants to be used in assignments. Instead, we must use a predefined variable, *in6addr_any*, which is initialized as follows by the C library:

```
const struct in6_addr in6addr_any = IN6ADDR_ANY_INIT;
```

Thus, we can initialize an IPv6 socket address structure using the wildcard address as follows:

```
struct sockaddr_in6 addr;

memset(&addr, 0, sizeof(struct sockaddr_in6));
addr.sin6_family = AF_INET6;
addr.sin6_addr = in6addr_any;
addr.sin6_port = htons(SOME_PORT_NUM);
```

The corresponding constant and variable for the IPv6 loopback address (::1) are IN6ADDR_LOOPBACK_INIT and *in6addr_loopback*.

Unlike their IPv4 counterparts, the IPv6 constant and variable initializers are in network byte order. But, as shown in the above code, we still must ensure that the port number is in network byte order.

If IPv4 and IPv6 coexist on a host, they share the same port-number space. This means that if, for example, an application binds an IPv6 socket to TCP port 2000 (using the IPv6 wildcard address), then an IPv4 TCP socket can't be bound to the same port. (The TCP/IP implementation ensures that sockets on other hosts are able to communicate with this socket, regardless of whether those hosts are running IPv4 or IPv6.)

The *sockaddr_storage* structure

With the IPv6 sockets API, the new generic *sockaddr_storage* structure was introduced. This structure is defined to be large enough to hold any type of socket address (i.e., any type of socket address structure can be cast and stored in it). In particular, this structure allows us to transparently store either an IPv4 or an IPv6 socket address, thus removing IP version dependencies from our code. The *sockaddr_storage* structure is defined on Linux as follows:

```
#define __ss_aligntype uint32_t         /* On 32-bit architectures */
struct sockaddr_storage {
    sa_family_t ss_family;
    __ss_aligntype __ss_align;          /* Force alignment */
    char __ss_padding[SS_PADSIZE];      /* Pad to 128 bytes */
};
```

59.5 Overview of Host and Service Conversion Functions

Computers represent IP addresses and port numbers in binary. However, humans find names easier to remember than numbers. Employing symbolic names also provides a useful level of indirection; users and programs can continue to use the same name even if the underlying numeric value changes.

A *hostname* is the symbolic identifier for a system that is connected to a network (possibly with multiple IP addresses). A *service name* is the symbolic representation of a port number.

The following methods are available for representing host addresses and ports:

- A host address can be represented as a binary value, as a symbolic hostname, or in presentation format (dotted-decimal for IPv4 or hex-string for IPv6).
- A port can be represented as a binary value or as a symbolic service name.

Various library functions are provided for converting between these formats. This section briefly summarizes these functions. The following sections describe the modern APIs (*inet_ntop()*, *inet_pton()*, *getaddrinfo()*, *getnameinfo()*, and so on) in detail. In Section 59.13, we briefly discuss the obsolete APIs (*inet_aton()*, *inet_ntoa()*, *gethostbyname()*, *getservbyname()*, and so on).

Converting IPv4 addresses between binary and human-readable forms

The *inet_aton()* and *inet_ntoa()* functions convert an IPv4 address in dotted-decimal notation to binary and vice versa. We describe these functions primarily because

they appear in historical code. Nowadays, they are obsolete. Modern programs that need to do such conversions should use the functions that we describe next.

Converting IPv4 and IPv6 addresses between binary and human-readable forms

The *inet_pton()* and *inet_ntop()* functions are like *inet_aton()* and *inet_ntoa()*, but differ in that they also handle IPv6 addresses. They convert binary IPv4 and IPv6 addresses to and from *presentation* format—that is, either dotted-decimal or hex-string notation.

Since humans deal better with names than with numbers, we normally use these functions only occasionally in programs. One use of *inet_ntop()* is to produce a printable representation of an IP address for logging purposes. Sometimes, it is preferable to use this function instead of converting ("resolving") an IP address to a hostname, for the following reasons:

- Resolving an IP address to a hostname involves a possibly time-consuming request to a DNS server.
- In some circumstances, there may not be a DNS (PTR) record that maps the IP address to a corresponding hostname.

We describe these functions (in Section 59.6) before *getaddrinfo()* and *getnameinfo()*, which perform conversions between binary representations and the corresponding symbolic names, principally because they present a much simpler API. This allows us to quickly show some working examples of the use of Internet domain sockets.

Converting host and service names to and from binary form (obsolete)

The *gethostbyname()* function returns the binary IP address(es) corresponding to a hostname and the *getservbyname()* function returns the port number corresponding to a service name. The reverse conversions are performed by *gethostbyaddr()* and *getservbyport()*. We describe these functions because they are widely used in existing code. However, they are now obsolete. (SUSv3 marks these functions obsolete, and SUSv4 removes their specifications.) New code should use the *getaddrinfo()* and *getnameinfo()* functions (described next) for such conversions.

Converting host and service names to and from binary form (modern)

The *getaddrinfo()* function is the modern successor to both *gethostbyname()* and *getservbyname()*. Given a hostname and a service name, *getaddrinfo()* returns a set of structures containing the corresponding binary IP address(es) and port number. Unlike *gethostbyname()*, *getaddrinfo()* transparently handles both IPv4 and IPv6 addresses. Thus, we can use it to write programs that don't contain dependencies on the IP version being employed. All new code should use *getaddrinfo()* for converting hostnames and service names to binary representation.

The *getnameinfo()* function performs the reverse translation, converting an IP address and port number into the corresponding hostname and service name.

We can also use *getaddrinfo()* and *getnameinfo()* to convert binary IP addresses to and from presentation format.

The discussion of *getaddrinfo()* and *getnameinfo()*, in Section 59.10, requires an accompanying description of DNS (Section 59.8) and the `/etc/services` file (Section 59.9). DNS allows cooperating servers to maintain a distributed database

that maps binary IP addresses to hostnames and vice versa. The existence of a system such as DNS is essential to the operation of the Internet, since centralized management of the enormous set of Internet hostnames would be impossible. The `/etc/services` file maps port numbers to symbolic service names.

59.6 The *inet_pton()* and *inet_ntop()* Functions

The *inet_pton()* and *inet_ntop()* functions allow conversion of both IPv4 and IPv6 addresses between binary form and dotted-decimal or hex-string notation.

```
#include <arpa/inet.h>

int inet_pton(int domain, const char *src_str, void *addrptr);
```

Returns 1 on successful conversion, 0 if *src_str* is not in presentation format, or −1 on error

```
const char *inet_ntop(int domain, const void *addrptr, char *dst_str, size_t len);
```

Returns pointer to *dst_str* on success, or `NULL` on error

The *p* in the names of these functions stands for "presentation," and the *n* stands for "network." The presentation form is a human-readable string, such as the following:

- `204.152.189.116` (IPv4 dotted-decimal address);
- `::1` (an IPv6 colon-separated hexadecimal address); or
- `::FFFF:204.152.189.116` (an IPv4-mapped IPv6 address).

The *inet_pton()* function converts the presentation string contained in *src_str* into a binary IP address in network byte order. The *domain* argument should be specified as either `AF_INET` or `AF_INET6`. The converted address is placed in the structure pointed to by *addrptr*, which should point to either an *in_addr* or an *in6_addr* structure, according to the value specified in *domain*.

The *inet_ntop()* function performs the reverse conversion. Again, *domain* should be specified as either `AF_INET` or `AF_INET6`, and *addrptr* should point to an *in_addr* or *in6_addr* structure that we wish to convert. The resulting null-terminated string is placed in the buffer pointed to by *dst_str*. The *len* argument must specify the size of this buffer. On success, *inet_ntop()* returns *dst_str*. If *len* is too small, then *inet_ntop()* returns `NULL`, with *errno* set to `ENOSPC`.

To correctly size the buffer pointed to by *dst_str*, we can employ two constants defined in `<netinet/in.h>`. These constants indicate the maximum lengths (including the terminating null byte) of the presentation strings for IPv4 and IPv6 addresses:

```
#define INET_ADDRSTRLEN  16     /* Maximum IPv4 dotted-decimal string */
#define INET6_ADDRSTRLEN 46     /* Maximum IPv6 hexadecimal string */
```

We provide examples of the use of *inet_pton()* and *inet_ntop()* in the next section.

59.7 Client-Server Example (Datagram Sockets)

In this section, we take the case-conversion server and client programs shown in Section 57.3 and modify them to use datagram sockets in the AF_INET6 domain. We present these programs with a minimum of commentary, since their structure is similar to the earlier programs. The main differences in the new programs lie in the declaration and initialization of the IPv6 socket address structure, which we described in Section 59.4.

The client and server both employ the header file shown in Listing 59-2. This header file defines the server's port number and the maximum size of messages that the client and server can exchange.

Listing 59-2: Header file used by i6d_ucase_sv.c and i6d_ucase_cl.c

sockets/i6d_ucase.h

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <ctype.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 10                     /* Maximum size of messages exchanged
                                           between client and server */

#define PORT_NUM 50002                  /* Server port number */
```

sockets/i6d_ucase.h

Listing 59-3 shows the server program. The server uses the *inet_ntop()* function to convert the host address of the client (obtained via the *recvfrom()* call) to printable form.

The client program shown in Listing 59-4 contains two notable modifications from the earlier UNIX domain version (Listing 57-7, on page 1173). The first difference is that the client interprets its initial command-line argument as the IPv6 address of the server. (The remaining command-line arguments are passed as separate datagrams to the server.) The client converts the server address to binary form using *inet_pton()*. The other difference is that the client doesn't bind its socket to an address. As noted in Section 58.6.1, if an Internet domain socket is not bound to an address, the kernel binds the socket to an ephemeral port on the host system. We can observe this in the following shell session log, where we run the server and the client on the same host:

```
$ ./i6d_ucase_sv &
[1] 31047
$ ./i6d_ucase_cl ::1 ciao                    Send to server on local host
Server received 4 bytes from (::1, 32770)
Response 1: CIAO
```

From the above output, we see that the server's *recvfrom()* call was able to obtain the address of the client's socket, including the ephemeral port number, despite the fact that the client did not do a *bind()*.

Listing 59-3: IPv6 case-conversion server using datagram sockets

sockets/i6d_ucase_sv.c

```
#include "i6d_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_in6 svaddr, claddr;
    int sfd, j;
    ssize_t numBytes;
    socklen_t len;
    char buf[BUF_SIZE];
    char claddrStr[INET6_ADDRSTRLEN];

    sfd = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sfd == -1)
        errExit("socket");

    memset(&svaddr, 0, sizeof(struct sockaddr_in6));
    svaddr.sin6_family = AF_INET6;
    svaddr.sin6_addr = in6addr_any;                    /* Wildcard address */
    svaddr.sin6_port = htons(PORT_NUM);

    if (bind(sfd, (struct sockaddr *) &svaddr,
                sizeof(struct sockaddr_in6)) == -1)
        errExit("bind");

    /* Receive messages, convert to uppercase, and return to client */

    for (;;) {
        len = sizeof(struct sockaddr_in6);
        numBytes = recvfrom(sfd, buf, BUF_SIZE, 0,
                            (struct sockaddr *) &claddr, &len);
        if (numBytes == -1)
            errExit("recvfrom");

        if (inet_ntop(AF_INET6, &claddr.sin6_addr, claddrStr,
                    INET6_ADDRSTRLEN) == NULL)
            printf("Couldn't convert client address to string\n");
        else
            printf("Server received %ld bytes from (%s, %u)\n",
                    (long) numBytes, claddrStr, ntohs(claddr.sin6_port));

        for (j = 0; j < numBytes; j++)
            buf[j] = toupper((unsigned char) buf[j]);

        if (sendto(sfd, buf, numBytes, 0, (struct sockaddr *) &claddr, len) !=
                numBytes)
            fatal("sendto");
    }
}
```

sockets/i6d_ucase_sv.c

Listing 59-4: IPv6 case-conversion client using datagram sockets

sockets/i6d_ucase_cl.c

```
#include "i6d_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_in6 svaddr;
    int sfd, j;
    size_t msgLen;
    ssize_t numBytes;
    char resp[BUF_SIZE];

    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s host-address msg...\n", argv[0]);

    sfd = socket(AF_INET6, SOCK_DGRAM, 0);      /* Create client socket */
    if (sfd == -1)
        errExit("socket");

    memset(&svaddr, 0, sizeof(struct sockaddr_in6));
    svaddr.sin6_family = AF_INET6;
    svaddr.sin6_port = htons(PORT_NUM);
    if (inet_pton(AF_INET6, argv[1], &svaddr.sin6_addr) <= 0)
        fatal("inet_pton failed for address '%s'", argv[1]);

    /* Send messages to server; echo responses on stdout */

    for (j = 2; j < argc; j++) {
        msgLen = strlen(argv[j]);
        if (sendto(sfd, argv[j], msgLen, 0, (struct sockaddr *) &svaddr,
                    sizeof(struct sockaddr_in6)) != msgLen)
            fatal("sendto");

        numBytes = recvfrom(sfd, resp, BUF_SIZE, 0, NULL, NULL);
        if (numBytes == -1)
            errExit("recvfrom");

        printf("Response %d: %.*s\n", j - 1, (int) numBytes, resp);
    }

    exit(EXIT_SUCCESS);
}
```

sockets/i6d_ucase_cl.c

59.8 Domain Name System (DNS)

In Section 59.10, we describe *getaddrinfo()*, which obtains the IP address(es) corresponding to a hostname, and *getnameinfo()*, which performs the converse task. However, before looking at these functions, we explain how DNS is used to maintain the mappings between hostnames and IP addresses.

Before the advent of DNS, mappings between hostnames and IP addresses were defined in a manually maintained local file, `/etc/hosts`, containing records of the following form:

```
# IP-address    canonical hostname      [aliases]
127.0.0.1       localhost
```

The *gethostbyname()* function (the predecessor to *getaddrinfo()*) obtained an IP address by searching this file, looking for a match on either the canonical hostname (i.e., the official or primary name of the host) or one of the (optional, space-delimited) aliases.

However, the `/etc/hosts` scheme scales poorly, and then becomes impossible, as the number of hosts in the network increases (e.g., the Internet, with millions of hosts).

DNS was devised to address this problem. The key ideas of DNS are the following:

- Hostnames are organized into a hierarchical namespace (Figure 59-2). Each *node* in the DNS hierarchy has a *label* (name), which may be up to 63 characters. At the root of the hierarchy is an unnamed node, the "anonymous root."
- A node's *domain name* consists of all of the names from that node up to the root concatenated together, with each name separated by a period (`.`). For example, `google.com` is the domain name for the node `google`.
- A *fully qualified domain name* (FQDN), such as `www.kernel.org.`, identifies a host within the hierarchy. A fully qualified domain name is distinguished by being terminated by a period, although in many contexts the period may be omitted.
- No single organization or system manages the entire hierarchy. Instead, there is a hierarchy of DNS servers, each of which manages a branch (a *zone*) of the tree. Normally, each zone has a *primary master name server*, and one or more *slave name servers* (sometimes also known as *secondary master name servers*), which provide backup in the event that the primary master name server crashes. Zones may themselves be divided into separately managed smaller zones. When a host is added within a zone, or the mapping of a hostname to an IP address is changed, the administrator responsible for the corresponding local name server updates the name database on that server. (No manual changes are required on any other name-server databases in the hierarchy.)

  > The DNS server implementation employed on Linux is the widely used Berkeley Internet Name Domain (BIND) implementation, *named(8)*, maintained by the *Internet Systems Consortium* (*http://www.isc.org/*). The operation of this daemon is controlled by the file `/etc/named.conf` (see the *named.conf(5)* manual page). The key reference on DNS and BIND is [Albitz & Liu, 2006]. Information about DNS can also be found in Chapter 14 of [Stevens, 1994], Chapter 11 of [Stevens et al., 2004], and Chapter 24 of [Comer, 2000].

- When a program calls *getaddrinfo()* to *resolve* (i.e., obtain the IP address for) a domain name, *getaddrinfo()* employs a suite of library functions (the *resolver library*) that communicate with the local DNS server. If this server can't supply the required information, then it communicates with other DNS servers within the hierarchy in order to obtain the information. Occasionally, this resolution process may take a noticeable amount of time, and DNS servers employ caching techniques to avoid unnecessary communication for frequently queried domain names.

Using the above approach allows DNS to cope with large namespaces, and does not require centralized management of names.

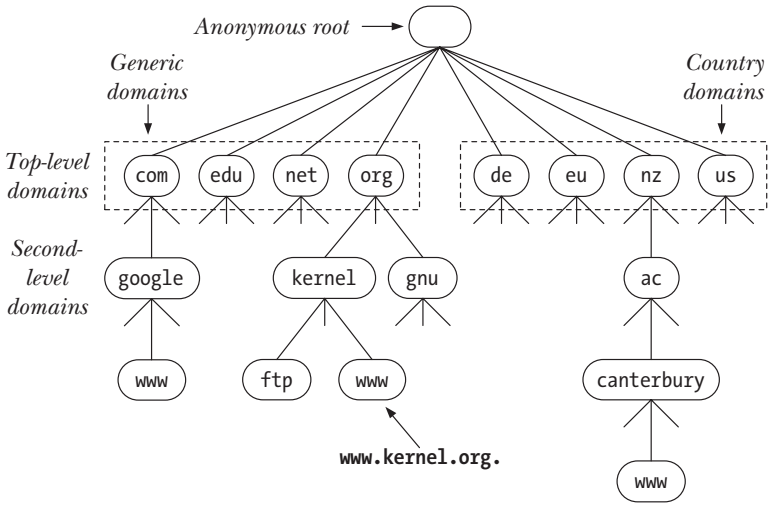

Figure 59-2: A subset of the DNS hierarchy

Recursive and iterative resolution requests

DNS resolution requests fall into two categories: *recursive* and *iterative*. In a recursive request, the requester asks the server to handle the entire task of resolution, including the task of communicating with any other DNS servers, if necessary. When an application on the local host calls *getaddrinfo()*, that function makes a recursive request to the local DNS server. If the local DNS server does not itself have the information to perform the resolution, it resolves the domain name iteratively.

We explain iterative resolution via an example. Suppose that the local DNS server is asked to resolve the name `www.otago.ac.nz`. To do this, it first communicates with one of a small set of *root name servers* that every DNS server is required to know about. (We can obtain a list of these servers using the command *dig . NS* or from the web page at *http://www.root-servers.org/*.) Given the name `www.otago.ac.nz`, the root name server refers the local DNS server to one of the `nz` DNS servers. The local DNS server then queries the `nz` server with the name `www.otago.ac.nz`, and receives a response referring it to the `ac.nz` server. The local DNS server then queries the `ac.nz` server with the name `www.otago.ac.nz`, and is referred to the `otago.ac.nz` server. Finally, the local DNS server queries the `otago.ac.nz` server with the name `www.otago.ac.nz`, and obtains the required IP address.

If we supply an incomplete domain name to *gethostbyname()*, the resolver will attempt to complete it before resolving it. The rules on how a domain name is completed are defined in `/etc/resolv.conf` (see the *resolv.conf(5)* manual page). By default, the resolver will at least try completion using the domain name of the local host. For example, if we are logged in on the machine `oghma.otago.ac.nz` and we type the command *ssh octavo*, the resulting DNS query will be for the name `octavo.otago.ac.nz`.

Top-level domains

The nodes immediately below the anonymous root form the so-called *top-level domains* (TLDs). (Below these are the *second-level domains*, and so on.) TLDs fall into two categories: *generic* and *country*.

Historically, there were seven *generic* TLDs, most of which can be considered international. We have shown four of the original generic TLDs in Figure 59-2. The other three are `int`, `mil`, and `gov`; the latter two are reserved for the United States. In more recent times, a number of new generic TLDs have been added (e.g., `info`, `name`, and `museum`).

Each nation has a corresponding *country* (or *geographical*) TLD (standardized as ISO 3166-1), with a 2-character name. In Figure 59-2, we have shown a few of these: `de` (Germany, *Deutschland*), `eu` (a supra-national geographical TLD for the European Union), `nz` (New Zealand), and `us` (United States of America). Several countries divide their TLD into a set of second-level domains in a manner similar to the generic domains. For example, New Zealand has `ac.nz` (academic institutions), `co.nz` (commercial), and `govt.nz` (government).

59.9 The `/etc/services` File

As noted in Section 58.6.1, well-known port numbers are centrally registered by IANA. Each of these ports has a corresponding *service name*. Because service numbers are centrally managed and are less volatile than IP addresses, an equivalent of the DNS server is usually not necessary. Instead, the port numbers and service names are recorded in the file `/etc/services`. The *getaddrinfo()* and *getnameinfo()* functions use the information in this file to convert service names to port numbers and vice versa.

The `/etc/services` file consists of lines containing three columns, as shown in the following examples:

```
# Service name  port/protocol  [aliases]
echo            7/tcp          Echo     # echo service
echo            7/udp          Echo
ssh             22/tcp                  # Secure Shell
ssh             22/udp
telnet          23/tcp                  # Telnet
telnet          23/udp
smtp            25/tcp                  # Simple Mail Transfer Protocol
smtp            25/udp
domain          53/tcp                  # Domain Name Server
domain          53/udp
http            80/tcp                  # Hypertext Transfer Protocol
http            80/udp
ntp             123/tcp                 # Network Time Protocol
ntp             123/udp
login           513/tcp                 # rlogin(1)
who             513/udp                 # rwho(1)
shell           514/tcp                 # rsh(1)
syslog          514/udp                 # syslog
```

The *protocol* is typically either `tcp` or `udp`. The optional (space-delimited) *aliases* specify alternative names for the service. In addition to the above, lines may include comments starting with the `#` character.

As noted previously, a given port number refers to distinct entities for UDP and TCP, but IANA policy assigns both port numbers to a service, even if that service uses only one protocol. For example, *telnet*, *ssh*, HTTP, and SMTP all use TCP, but the corresponding UDP port is also assigned to these services. Conversely, NTP uses only UDP, but the TCP port 123 is also assigned to this service. In some cases, a service uses both UDP and TCP; DNS and *echo* are examples of such services. Finally, there are a very few cases where the UDP and TCP ports with the same number are assigned to different services; for example, *rsh* uses TCP port 514, while the *syslog* daemon (Section 37.5) uses UDP port 514. This is because these port numbers were assigned before the adoption of the present IANA policy.

> The `/etc/services` file is merely a record of name-to-number mappings. It is not a reservation mechanism: the appearance of a port number in `/etc/services` doesn't guarantee that it will actually be available for binding by a particular service.

59.10 Protocol-Independent Host and Service Conversion

The *getaddrinfo()* function converts host and service names to IP addresses and port numbers. It was defined in POSIX.1g as the (reentrant) successor to the obsolete *gethostbyname()* and *getservbyname()* functions. (Replacing the use of *gethostbyname()* with *getaddrinfo()* allows us to eliminate IPv4-versus-IPv6 dependencies from our programs.)

The *getnameinfo()* function is the converse of *getaddrinfo()*. It translates a socket address structure (either IPv4 or IPv6) to strings containing the corresponding host and service name. This function is the (reentrant) equivalent of the obsolete *gethostbyaddr()* and *getservbyport()* functions.

> Chapter 11 of [Stevens et al., 2004] describes *getaddrinfo()* and *getnameinfo()* in detail, and provides implementations of these functions. These functions are also described in RFC 3493.

59.10.1 The *getaddrinfo()* Function

Given a host name and a service name, *getaddrinfo()* returns a list of socket address structures, each of which contains an IP address and port number.

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *host, const char *service,
                const struct addrinfo *hints, struct addrinfo **result);
```

Returns 0 on success, or nonzero on error

As input, *getaddrinfo()* takes the arguments *host*, *service*, and *hints*. The *host* argument contains either a hostname or a numeric address string, expressed in IPv4 dotted-decimal notation or IPv6 hex-string notation. (To be precise, *getaddrinfo()* accepts IPv4 numeric strings in the more general numbers-and-dots notation described in Section 59.13.1.) The *service* argument contains either a service name or a decimal port number. The *hints* argument points to an *addrinfo* structure that specifies further criteria for selecting the socket address structures returned via *result*. We describe the *hints* argument in more detail below.

As output, *getaddrinfo()* dynamically allocates a linked list of *addrinfo* structures and sets *result* pointing to the beginning of this list. Each of these *addrinfo* structures includes a pointer to a socket address structure corresponding to *host* and *service* (Figure 59-3). The *addrinfo* structure has the following form:

```
struct addrinfo {
    int    ai_flags;           /* Input flags (AI_* constants) */
    int    ai_family;          /* Address family */
    int    ai_socktype;        /* Type: SOCK_STREAM, SOCK_DGRAM */
    int    ai_protocol;        /* Socket protocol */
    size_t ai_addrlen;         /* Size of structure pointed to by ai_addr */
    char  *ai_canonname;       /* Canonical name of host */
    struct sockaddr *ai_addr;  /* Pointer to socket address structure */
    struct addrinfo *ai_next;  /* Next structure in linked list */
};
```

The *result* argument returns a list of structures, rather than a single structure, because there may be multiple combinations of host and service corresponding to the criteria specified in *host*, *service*, and *hints*. For example, multiple address structures could be returned for a host with more than one network interface. Furthermore, if *hints.ai_socktype* was specified as 0, then two structures could be returned—one for a `SOCK_DGRAM` socket, the other for a `SOCK_STREAM` socket—if the given *service* was available for both UDP and TCP.

The fields of each *addrinfo* structure returned via *result* describe properties of the associated socket address structure. The *ai_family* field is set to either `AF_INET` or `AF_INET6`, informing us of the type of the socket address structure. The *ai_socktype* field is set to either `SOCK_STREAM` or `SOCK_DGRAM`, indicating whether this address structure is for a TCP or a UDP service. The *ai_protocol* field returns a protocol value appropriate for the address family and socket type. (The three fields *ai_family*, *ai_socktype*, and *ai_protocol* supply the values required for the arguments used when calling *socket()* to create a socket for this address.) The *ai_addrlen* field gives the size (in bytes) of the socket address structure pointed to by *ai_addr*. The *in_addr* field points to the socket address structure (an *in_addr* structure for IPv4 or an *in6_addr* structure for IPv6). The *ai_flags* field is unused (it is used for the *hints* argument). The *ai_canonname* field is used only in the first *addrinfo* structure, and only if the `AI_CANONNAME` flag is employed in *hints.ai_flags*, as described below.

As with *gethostbyname()*, *getaddrinfo()* may need to send a request to a DNS server, and this request may take some time to complete. The same applies for *getnameinfo()*, which we describe in Section 59.10.4.

We demonstrate the use of *getaddrinfo()* in Section 59.11.

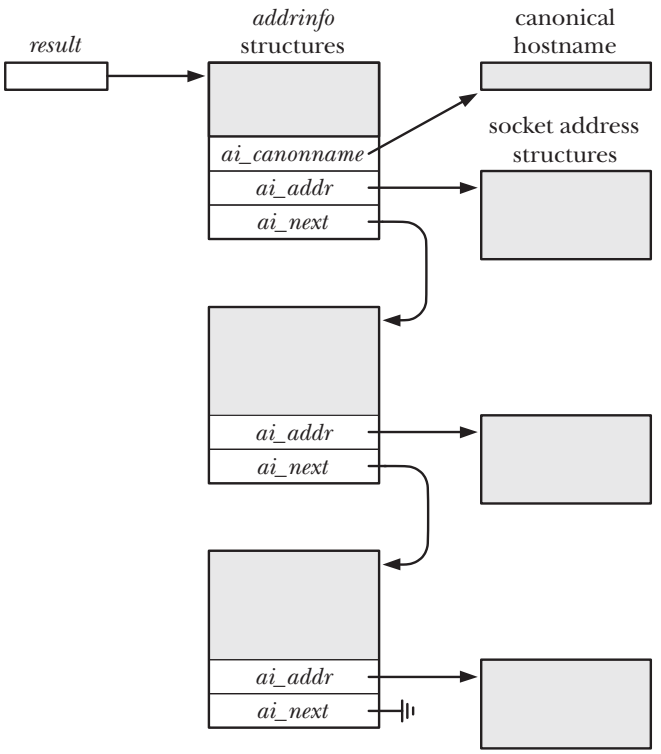

Figure 59-3: Structures allocated and returned by *getaddrinfo()*

The *hints* argument

The *hints* argument specifies further criteria for selecting the socket address structures returned by *getaddrinfo()*. When used as the *hints* argument, only the *ai_flags*, *ai_family*, *ai_socktype*, and *ai_protocol* fields of the *addrinfo* structure can be set. The other fields are unused, and should be initialized to 0 or `NULL`, as appropriate.

The *hints.ai_family* field selects the domain for the returned socket address structures. It may be specified as `AF_INET` or `AF_INET6` (or some other `AF_*` constant, if the implementation supports it). If we are interested in getting back all types of socket address structures, we can specify the value `AF_UNSPEC` for this field.

The *hints.ai_socktype* field specifies the type of socket for which the returned address structure is to be used. If we specify this field as `SOCK_DGRAM`, then a lookup is performed for the UDP service, and a corresponding socket address structure is returned via *result*. If we specify `SOCK_STREAM`, a lookup for the TCP service is performed. If *hints.ai_socktype* is specified as 0, any socket type is acceptable.

The *hints.ai_protocol* field selects the socket protocol for the returned address structures. For our purposes, this field is always specified as 0, meaning that the caller will accept any protocol.

The *hints.ai_flags* field is a bit mask that modifies the behavior of *getaddrinfo()*. This field is formed by ORing together zero or more of the following values:

AI_ADDRCONFIG
: Return IPv4 addresses only if there is at least one IPv4 address configured for the local system (other than the IPv4 loopback address), and return IPv6 addresses only if there is at least one IPv6 address configured for the local system (other than the IPv6 loopback address).

AI_ALL
: See the description of AI_V4MAPPED below.

AI_CANONNAME
: If *host* is not NULL, return a pointer to a null-terminated string containing the canonical name of the host. This pointer is returned in a buffer pointed to by the *ai_canonname* field of the first of the *addrinfo* structures returned via *result*.

AI_NUMERICHOST
: Force interpretation of *host* as a numeric address string. This is used to prevent name resolution in cases where it is unnecessary, since name resolution can be time-consuming.

AI_NUMERICSERV
: Interpret *service* as a numeric port number. This flag prevents the invocation of any name-resolution service, which is not required if *service* is a numeric string.

AI_PASSIVE
: Return socket address structures suitable for a passive open (i.e., a listening socket). In this case, *host* should be NULL, and the IP address component of the socket address structure(s) returned by *result* will contain a wildcard IP address (i.e., INADDR_ANY or IN6ADDR_ANY_INIT). If this flag is not set, then the address structure(s) returned via *result* will be suitable for use with *connect()* and *sendto()*; if *host* is NULL, then the IP address in the returned socket address structures will be set to the loopback IP address (either INADDR_LOOPBACK or IN6ADDR_LOOPBACK_INIT, according to the domain).

AI_V4MAPPED
: If AF_INET6 was specified in the *ai_family* field of *hints*, then IPv4-mapped IPv6 address structures should be returned in *result* if no matching IPv6 address could be found. If AI_ALL is specified in conjunction with AI_V4MAPPED, then both IPv6 and IPv4 address structures are returned in *result*, with IPv4 addresses being returned as IPv4-mapped IPv6 address structures.

As noted above for AI_PASSIVE, *host* can be specified as NULL. It is also possible to specify *service* as NULL, in which case the port number in the returned address structures is set to 0 (i.e., we are just interested in resolving hostnames to addresses). It is not permitted, however, to specify both *host* and *service* as NULL.

If we don't need to specify any of the above selection criteria in hints, then *hints* may be specified as NULL, in which case *ai_socktype* and *ai_protocol* are assumed

as 0, *ai_flags* is assumed as (AI_V4MAPPED | AI_ADDRCONFIG), and *ai_family* is assumed as AF_UNSPEC. (The *glibc* implementation deliberately deviates from SUSv3, which states that if *hints* is NULL, *ai_flags* is assumed as 0.)

59.10.2 Freeing *addrinfo* Lists: *freeaddrinfo()*

The *getaddrinfo()* function dynamically allocates memory for all of the structures referred to by *result* (Figure 59-3). Consequently, the caller must deallocate these structures when they are no longer needed. The *freeaddrinfo()* function is provided to conveniently perform this deallocation in a single step.

```
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo(struct addrinfo *result);
```

If we want to preserve a copy of one of the *addrinfo* structures or its associated socket address structure, then we must duplicate the structure(s) before calling *freeaddrinfo()*.

59.10.3 Diagnosing Errors: *gai_strerror()*

On error, *getaddrinfo()* returns one of the nonzero error codes shown in Table 59-1.

Table 59-1: Error returns for *getaddrinfo()* and *getnameinfo()*

Error constant	Description
EAI_ADDRFAMILY	No addresses for *host* exist in *hints.ai_family* (not in SUSv3, but defined on most implementations; *getaddrinfo()* only)
EAI_AGAIN	Temporary failure in name resolution (try again later)
EAI_BADFLAGS	An invalid flag was specified in *hints.ai_flags*
EAI_FAIL	Unrecoverable failure while accessing name server
EAI_FAMILY	Address family specified in *hints.ai_family* is not supported
EAI_MEMORY	Memory allocation failure
EAI_NODATA	No address associated with *host* (not in SUSv3, but defined on most implementations; *getaddrinfo()* only)
EAI_NONAME	Unknown *host* or *service*, or both *host* and *service* were NULL, or AI_NUMERICSERV specified and *service* didn't point to numeric string
EAI_OVERFLOW	Argument buffer overflow
EAI_SERVICE	Specified *service* not supported for *hints.ai_socktype* (*getaddrinfo()* only)
EAI_SOCKTYPE	Specified *hints.ai_socktype* is not supported (*getaddrinfo()* only)
EAI_SYSTEM	System error returned in *errno*

Given one of the error codes in Table 59-1, the *gai_strerror()* function returns a string describing the error. (This string is typically briefer than the description shown in Table 59-1.)

```
#include <netdb.h>

const char *gai_strerror(int errcode);
                                    Returns pointer to string containing error message
```

We can use the string returned by *gai_strerror()* as part of an error message displayed by an application.

59.10.4 The *getnameinfo()* Function

The *getnameinfo()* function is the converse of *getaddrinfo()*. Given a socket address structure (either IPv4 or IPv6), it returns strings containing the corresponding host and service name, or numeric equivalents if the names can't be resolved.

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *addr, socklen_t addrlen, char *host,
                size_t hostlen, char *service, size_t servlen, int flags);
                                    Returns 0 on success, or nonzero on error
```

The *addr* argument is a pointer to the socket address structure that is to be converted. The length of that structure is given in *addrlen*. Typically, the values for *addr* and *addrlen* are obtained from a call to *accept()*, *recvfrom()*, *getsockname()*, or *getpeername()*.

The resulting host and service names are returned as null-terminated strings in the buffers pointed to by *host* and *service*. These buffers must be allocated by the caller, and their sizes must be passed in *hostlen* and *servlen*. The <netdb.h> header file defines two constants to assist in sizing these buffers. NI_MAXHOST indicates the maximum size, in bytes, for a returned hostname string. It is defined as 1025. NI_MAXSERV indicates the maximum size, in bytes, for a returned service name string. It is defined as 32. These two constants are not specified in SUSv3, but they are defined on all UNIX implementations that provide *getnameinfo()*. (Since *glibc* 2.8, we must define one of the feature text macros _BSD_SOURCE, _SVID_SOURCE, or _GNU_SOURCE to obtain the definitions of NI_MAXHOST and NI_MAXSERV.)

If we are not interested in obtaining the hostname, we can specify *host* as NULL and *hostlen* as 0. Similarly, if we don't need the service name, we can specify *service* as NULL and *servlen* as 0. However, at least one of *host* and *service* must be non-NULL (and the corresponding length argument must be nonzero).

The final argument, *flags*, is a bit mask that controls the behavior of *getnameinfo()*. The following constants may be ORed together to form this bit mask:

NI_DGRAM

> By default, *getnameinfo()* returns the name corresponding to a *stream* socket (i.e., TCP) service. Normally, this doesn't matter, because, as noted in Section 59.9, the service names are usually the same for corresponding

TCP and UDP ports. However, in the few instances where the names differ, the NI_DGRAM flag forces the name of the datagram socket (i.e., UDP) service to be returned.

NI_NAMEREQD
: By default, if the hostname can't be resolved, a numeric address string is returned in *host*. If the NI_NAMEREQD flag is specified, an error (EAI_NONAME) is returned instead.

NI_NOFQDN
: By default, the fully qualified domain name for the host is returned. Specifying the NI_NOFQDN flag causes just the first (i.e., the hostname) part of the name to be returned, if this is a host on the local network.

NI_NUMERICHOST
: Force a numeric address string to be returned in *host*. This is useful if we want to avoid a possibly time-consuming call to the DNS server.

NI_NUMERICSERV
: Force a decimal port number string to be returned in *service*. This is useful in cases where we know that the port number doesn't correspond to a service name—for example, if it is an ephemeral port number assigned to the socket by the kernel—and we want to avoid the inefficiency of unnecessarily searching /etc/services.

On success, *getnameinfo()* returns 0. On error, it returns one of the nonzero error codes shown in Table 59-1.

59.11 Client-Server Example (Stream Sockets)

We now have enough information to look at a simple client-server application using TCP sockets. The task performed by this application is the same as that performed by the FIFO client-server application presented in Section 44.8: allocating unique sequence numbers (or ranges of sequence numbers) to clients.

In order to handle the possibility that integers may be represented in different formats on the server and client hosts, we encode all transmitted integers as strings terminated by a newline, and use our *readLine()* function (Listing 59-1) to read these strings.

Common header file

Both the server and the client include the header file shown in Listing 59-5. This file includes various other header files, and defines the TCP port number to be used by the application.

Server program

The server program shown in Listing 59-6 performs the following steps:

- Initialize the server's sequence number either to 1 or to the value supplied in the optional command-line argument ①.

- Ignore the SIGPIPE signal ②. This prevents the server from receiving the SIGPIPE signal if it tries to write to a socket whose peer has been closed; instead, the *write()* fails with the error EPIPE.
- Call *getaddrinfo()* ④ to obtain a set of socket address structures for a TCP socket that uses the port number PORT_NUM. (Instead of using a hard-coded port number, we would more typically use a service name.) We specify the AI_PASSIVE flag ③ so that the resulting socket will be bound to the wildcard address (Section 58.5). As a result, if the server is run on a multihomed host, it can accept connection requests sent to any of the host's network addresses.
- Enter a loop that iterates through the socket address structures returned by the previous step ⑤. The loop terminates when the program finds an address structure that can be used to successfully create and bind a socket ⑦.
- Set the SO_REUSEADDR option for the socket created in the previous step ⑥. We defer discussion of this option until Section 61.10, where we note that a TCP server should usually set this option on its listening socket.
- Mark the socket as a listening socket ⑧.
- Commence an infinite for loop ⑨ that services clients iteratively (Chapter 60). Each client's request is serviced before the next client's request is accepted. For each client, the server performs the following steps:
  - Accept a new connection ⑩. The server passes non-NULL pointers for the second and third arguments to *accept()*, in order to obtain the address of the client. The server displays the client's address (IP address plus port number) on standard output ⑪.
  - Read the client's message ⑫, which consists of a newline-terminated string specifying how many sequence numbers the client wants. The server converts this string to an integer and stores it in the variable *reqLen* ⑬.
  - Send the current value of the sequence number (*seqNum*) back to the client, encoding it as a newline-terminated string ⑭. The client can assume that it has been allocated all of the sequence numbers in the range *seqNum* to *(seqNum + reqLen − 1)*.
  - Update the value of the server's sequence number by adding *reqLen* to *seqNum* ⑮.

Listing 59-5: Header file used by is_seqnum_sv.c and is_seqnum_cl.c

sockets/is_seqnum.h

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <signal.h>
#include "read_line.h"          /* Declaration of readLine() */
#include "tlpi_hdr.h"

#define PORT_NUM "50000"        /* Port number for server */

#define INT_LEN 30              /* Size of string able to hold largest
                                   integer (including terminating '\n') */
```

sockets/is_seqnum.h

Listing 59-6: An iterative server that uses a stream socket to communicate with clients

sockets/is_seqnum_sv.c

```
#define _BSD_SOURCE             /* To get definitions of NI_MAXHOST and
                                   NI_MAXSERV from <netdb.h> */
#include <netdb.h>
#include "is_seqnum.h"

#define BACKLOG 50

int
main(int argc, char *argv[])
{
    uint32_t seqNum;
    char reqLenStr[INT_LEN];            /* Length of requested sequence */
    char seqNumStr[INT_LEN];            /* Start of granted sequence */
    struct sockaddr_storage claddr;
    int lfd, cfd, optval, reqLen;
    socklen_t addrlen;
    struct addrinfo hints;
    struct addrinfo *result, *rp;
#define ADDRSTRLEN (NI_MAXHOST + NI_MAXSERV + 10)
    char addrStr[ADDRSTRLEN];
    char host[NI_MAXHOST];
    char service[NI_MAXSERV];

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [init-seq-num]\n", argv[0]);

①   seqNum = (argc > 1) ? getInt(argv[1], 0, "init-seq-num") : 0;

②   if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
        errExit("signal");

    /* Call getaddrinfo() to obtain a list of addresses that
       we can try binding to */

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_family = AF_UNSPEC;        /* Allows IPv4 or IPv6 */
③   hints.ai_flags = AI_PASSIVE | AI_NUMERICSERV;
                        /* Wildcard IP address; service name is numeric */
④   if (getaddrinfo(NULL, PORT_NUM, &hints, &result) != 0)
        errExit("getaddrinfo");

    /* Walk through returned list until we find an address structure
       that can be used to successfully create and bind a socket */

    optval = 1;
⑤   for (rp = result; rp != NULL; rp = rp->ai_next) {
        lfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (lfd == -1)
            continue;                   /* On error, try next address */
```

```
⑥          if (setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval))
                   == -1)
               errExit("setsockopt");

⑦          if (bind(lfd, rp->ai_addr, rp->ai_addrlen) == 0)
               break;                       /* Success */

           /* bind() failed: close this socket and try next address */

           close(lfd);
       }

       if (rp == NULL)
           fatal("Could not bind socket to any address");

⑧      if (listen(lfd, BACKLOG) == -1)
           errExit("listen");

       freeaddrinfo(result);

⑨      for (;;) {                  /* Handle clients iteratively */

           /* Accept a client connection, obtaining client's address */

           addrlen = sizeof(struct sockaddr_storage);
⑩          cfd = accept(lfd, (struct sockaddr *) &claddr, &addrlen);
           if (cfd == -1) {
               errMsg("accept");
               continue;
           }

⑪          if (getnameinfo((struct sockaddr *) &claddr, addrlen,
                       host, NI_MAXHOST, service, NI_MAXSERV, 0) == 0)
               snprintf(addrStr, ADDRSTRLEN, "(%s, %s)", host, service);
           else
               snprintf(addrStr, ADDRSTRLEN, "(?UNKNOWN?)");
           printf("Connection from %s\n", addrStr);

           /* Read client request, send sequence number back */

⑫          if (readLine(cfd, reqLenStr, INT_LEN) <= 0) {
               close(cfd);
               continue;                    /* Failed read; skip request */
           }

⑬          reqLen = atoi(reqLenStr);
           if (reqLen <= 0) {               /* Watch for misbehaving clients */
               close(cfd);
               continue;                    /* Bad request; skip it */
           }

⑭          snprintf(seqNumStr, INT_LEN, "%d\n", seqNum);
           if (write(cfd, &seqNumStr, strlen(seqNumStr)) != strlen(seqNumStr))
               fprintf(stderr, "Error on write");
```

```
⑮          seqNum += reqLen;               /* Update sequence number */

           if (close(cfd) == -1)           /* Close connection */
               errMsg("close");
       }
   }
```

sockets/is_seqnum_sv.c

Client program

The client program is shown in Listing 59-7. This program accepts two arguments. The first argument, which is the name of the host on which the server is running, is mandatory. The optional second argument is the length of the sequence desired by the client. The default length is 1. The client performs the following steps:

- Call *getaddrinfo()* to obtain a set of socket address structures suitable for connecting to a TCP server bound to the specified host ①. For the port number, the client specifies PORT_NUM.
- Enter a loop ② that iterates through the socket address structures returned by the previous step, until the client finds one that can be used to successfully create ③ and connect ④ a socket to the server. Since the client has not bound its socket, the *connect()* call causes the kernel to assign an ephemeral port to the socket.
- Send an integer specifying the length of the client's desired sequence ⑤. This integer is sent as a newline-terminated string.
- Read the sequence number sent back by the server (which is likewise a newline-terminated string) ⑥ and print it on standard output ⑦.

When we run the server and the client on the same host, we see the following:

```
$ ./is_seqnum_sv &
[1] 4075
$ ./is_seqnum_cl localhost              Client 1: requests 1 sequence number
Connection from (localhost, 33273)      Server displays client address + port
Sequence number: 0                      Client displays returned sequence number
$ ./is_seqnum_cl localhost 10           Client 2: requests 10 sequence numbers
Connection from (localhost, 33274)
Sequence number: 1
$ ./is_seqnum_cl localhost              Client 3: requests 1 sequence number
Connection from (localhost, 33275)
Sequence number: 11
```

Next, we demonstrate the use of *telnet* for debugging this application:

```
$ telnet localhost 50000                Our server uses this port number
                                        Empty line printed by telnet
Trying 127.0..0.1...
Connection from (localhost, 33276)
Connected to localhost.
Escape character is '^]'.
1                                       Enter length of requested sequence
12                                      telnet displays sequence number and
Connection closed by foreign host.      detects that server closed connection
```

In the shell session log, we see that the kernel cycles sequentially through the ephemeral port numbers. (Other implementations exhibit similar behavior.) On Linux, this behavior is the result of an optimization to minimize hash lookups in the kernel's table of local socket bindings. When the upper limit for these numbers is reached, the kernel recommences allocating an available number starting at the low end of the range (defined by the Linux-specific `/proc/sys/net/ipv4/ip_local_port_range` file).

Listing 59-7: A client that uses stream sockets

sockets/is_seqnum_cl.c

```
#include <netdb.h>
#include "is_seqnum.h"

int
main(int argc, char *argv[])
{
    char *reqLenStr;                    /* Requested length of sequence */
    char seqNumStr[INT_LEN];            /* Start of granted sequence */
    int cfd;
    ssize_t numRead;
    struct addrinfo hints;
    struct addrinfo *result, *rp;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s server-host [sequence-len]\n", argv[0]);

    /* Call getaddrinfo() to obtain a list of addresses that
       we can try connecting to */

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_family = AF_UNSPEC;                /* Allows IPv4 or IPv6 */
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_NUMERICSERV;

①   if (getaddrinfo(argv[1], PORT_NUM, &hints, &result) != 0)
        errExit("getaddrinfo");

    /* Walk through returned list until we find an address structure
       that can be used to successfully connect a socket */

②   for (rp = result; rp != NULL; rp = rp->ai_next) {
③       cfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (cfd == -1)
            continue;                           /* On error, try next address */

④       if (connect(cfd, rp->ai_addr, rp->ai_addrlen) != -1)
            break;                              /* Success */
```

```
            /* Connect failed: close this socket and try next address */

            close(cfd);
        }

        if (rp == NULL)
            fatal("Could not connect socket to any address");

        freeaddrinfo(result);

        /* Send requested sequence length, with terminating newline */

⑤       reqLenStr = (argc > 2) ? argv[2] : "1";
        if (write(cfd, reqLenStr, strlen(reqLenStr)) !=  strlen(reqLenStr))
            fatal("Partial/failed write (reqLenStr)");
        if (write(cfd, "\n", 1) != 1)
            fatal("Partial/failed write (newline)");

        /* Read and display sequence number returned by server */

⑥       numRead = readLine(cfd, seqNumStr, INT_LEN);
        if (numRead == -1)
            errExit("readLine");
        if (numRead == 0)
            fatal("Unexpected EOF from server");

⑦       printf("Sequence number: %s", seqNumStr);          /* Includes '\n' */

        exit(EXIT_SUCCESS);                                /* Closes 'cfd' */
    }
```

sockets/is_seqnum_cl.c

59.12 An Internet Domain Sockets Library

In this section, we use the functions presented in Section 59.10 to implement a library of functions to perform tasks commonly required for Internet domain sockets. (This library abstracts many of the steps shown in the example programs presented in Section 59.11.) Since these functions employ the protocol-independent *getaddrinfo()* and *getnameinfo()* functions, they can be used with both IPv4 and IPv6. Listing 59-8 shows the header file that declares these functions.

Many of the functions in this library have similar arguments:

- The *host* argument is a string containing either a hostname or a numeric address (in IPv4 dotted-decimal, or IPv6 hex-string notation). Alternatively, *host* can be specified as a `NULL` pointer to indicate that the loopback IP address is to be used.
- The *service* argument is either a service name or a port number specified as a decimal string.
- The *type* argument is a socket type, specified as either `SOCK_STREAM` or `SOCK_DGRAM`.

Listing 59-8: Header file for `inet_sockets.c`

sockets/inet_sockets.h

```
#ifndef INET_SOCKETS_H
#define INET_SOCKETS_H          /* Prevent accidental double inclusion */

#include <sys/socket.h>
#include <netdb.h>

int inetConnect(const char *host, const char *service, int type);

int inetListen(const char *service, int backlog, socklen_t *addrlen);

int inetBind(const char *service, int type, socklen_t *addrlen);

char *inetAddressStr(const struct sockaddr *addr, socklen_t addrlen,
                char *addrStr, int addrStrLen);

#define IS_ADDR_STR_LEN 4096
                        /* Suggested length for string buffer that caller
                           should pass to inetAddressStr(). Must be greater
                           than (NI_MAXHOST + NI_MAXSERV + 4) */
#endif
```

sockets/inet_sockets.h

The *inetConnect()* function creates a socket with the given socket *type*, and connects it to the address specified by *host* and *service*. This function is designed for TCP or UDP clients that need to connect their socket to a server socket.

```
#include "inet_sockets.h"

int inetConnect(const char *host, const char *service, int type);
```

Returns a file descriptor on success, or –1 on error

The file descriptor for the new socket is returned as the function result.

The *inetListen()* function creates a listening stream (`SOCK_STREAM`) socket bound to the wildcard IP address on the TCP port specified by *service*. This function is designed for use by TCP servers.

```
#include "inet_sockets.h"

int inetListen(const char *service, int backlog, socklen_t *addrlen);
```

Returns a file descriptor on success, or –1 on error

The file descriptor for the new socket is returned as the function result.

The *backlog* argument specifies the permitted backlog of pending connections (as for *listen()*).

If *addrlen* is specified as a non-NULL pointer, then the location it points to is used to return the size of the socket address structure corresponding to the returned file descriptor. This value allows us to allocate a socket address buffer of the appropriate size to be passed to a later *accept()* call if we want to obtain the address of a connecting client.

The *inetBind()* function creates a socket of the given *type*, bound to the wildcard IP address on the port specified by *service* and *type*. (The socket *type* indicates whether this is a TCP or UDP service.) This function is designed (primarily) for UDP servers and clients to create a socket bound to a specific address.

```
#include "inet_sockets.h"

int inetBind(const char *service, int type, socklen_t *addrlen);
                                Returns a file descriptor on success, or –1 on error
```

The file descriptor for the new socket is returned as the function result.

As with *inetListen()*, *inetBind()* returns the length of the associated socket address structure for this socket in the location pointed to by *addrlen*. This is useful if we want to allocate a buffer to pass to *recvfrom()* in order to obtain the address of the socket sending a datagram. (Many of the steps required for *inetListen()* and *inetBind()* are the same, and these steps are implemented within the library by a single function, *inetPassiveSocket()*.)

The *inetAddressStr()* function converts an Internet socket address to printable form.

```
#include "inet_sockets.h"

char *inetAddressStr(const struct sockaddr *addr, socklen_t addrlen,
                     char *addrStr, int addrStrLen);
            Returns pointer to addrStr, a string containing host and service name
```

Given a socket address structure in *addr*, whose length is specified in *addrlen*, *inetAddressStr()* returns a null-terminated string containing the corresponding hostname and port number in the following form:

```
(hostname, port-number)
```

The string is returned in the buffer pointed to by *addrStr*. The caller must specify the size of this buffer in *addrStrLen*. If the returned string would exceed *(addrStrLen – 1)* bytes, it is truncated. The constant IS_ADDR_STR_LEN defines a suggested size for the *addrStr* buffer that should be large enough to handle all possible return strings. As its function result, *inetAddressStr()* returns *addrStr*.

The implementation of the functions described in this section is shown in Listing 59-9.

Listing 59-9: An Internet domain sockets library

sockets/inet_sockets.c

```
#define _BSD_SOURCE             /* To get NI_MAXHOST and NI_MAXSERV
                                   definitions from <netdb.h> */
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include "inet_sockets.h"       /* Declares functions defined here */
#include "tlpi_hdr.h"

int
inetConnect(const char *host, const char *service, int type)
{
    struct addrinfo hints;
    struct addrinfo *result, *rp;
    int sfd, s;

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_family = AF_UNSPEC;        /* Allows IPv4 or IPv6 */
    hints.ai_socktype = type;

    s = getaddrinfo(host, service, &hints, &result);
    if (s != 0) {
        errno = ENOSYS;
        return -1;
    }

    /* Walk through returned list until we find an address structure
       that can be used to successfully connect a socket */

    for (rp = result; rp != NULL; rp = rp->ai_next) {
        sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (sfd == -1)
            continue;                   /* On error, try next address */

        if (connect(sfd, rp->ai_addr, rp->ai_addrlen) != -1)
            break;                      /* Success */

        /* Connect failed: close this socket and try next address */

        close(sfd);
    }

    freeaddrinfo(result);

    return (rp == NULL) ? -1 : sfd;
}
```

```
static int              /* Public interfaces: inetBind() and inetListen() */
inetPassiveSocket(const char *service, int type, socklen_t *addrlen,
                  Boolean doListen, int backlog)
{
    struct addrinfo hints;
    struct addrinfo *result, *rp;
    int sfd, optval, s;

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_socktype = type;
    hints.ai_family = AF_UNSPEC;        /* Allows IPv4 or IPv6 */
    hints.ai_flags = AI_PASSIVE;        /* Use wildcard IP address */

    s = getaddrinfo(NULL, service, &hints, &result);
    if (s != 0)
        return -1;

    /* Walk through returned list until we find an address structure
       that can be used to successfully create and bind a socket */

    optval = 1;
    for (rp = result; rp != NULL; rp = rp->ai_next) {
        sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (sfd == -1)
            continue;                   /* On error, try next address */

        if (doListen) {
            if (setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &optval,
                    sizeof(optval)) == -1) {
                close(sfd);
                freeaddrinfo(result);
                return -1;
            }
        }

        if (bind(sfd, rp->ai_addr, rp->ai_addrlen) == 0)
            break;                      /* Success */

        /* bind() failed: close this socket and try next address */

        close(sfd);
    }

    if (rp != NULL && doListen) {
        if (listen(sfd, backlog) == -1) {
            freeaddrinfo(result);
            return -1;
        }
    }

    if (rp != NULL && addrlen != NULL)
        *addrlen =  rp->ai_addrlen;     /* Return address structure size */
```

```
    freeaddrinfo(result);

    return (rp == NULL) ? -1 : sfd;
}

int
inetListen(const char *service, int backlog, socklen_t *addrlen)
{
    return inetPassiveSocket(service, SOCK_STREAM, addrlen, TRUE, backlog);
}

int
inetBind(const char *service, int type, socklen_t *addrlen)
{
    return inetPassiveSocket(service, type, addrlen, FALSE, 0);
}

char *
inetAddressStr(const struct sockaddr *addr, socklen_t addrlen,
               char *addrStr, int addrStrLen)
{
    char host[NI_MAXHOST], service[NI_MAXSERV];

    if (getnameinfo(addr, addrlen, host, NI_MAXHOST,
                    service, NI_MAXSERV, NI_NUMERICSERV) == 0)
        snprintf(addrStr, addrStrLen, "(%s, %s)", host, service);
    else
        snprintf(addrStr, addrStrLen, "(?UNKNOWN?)");

    addrStr[addrStrLen - 1] = '\0';     /* Ensure result is null-terminated */
    return addrStr;
}
```

sockets/inet_sockets.c

59.13 Obsolete APIs for Host and Service Conversions

In the following sections, we describe the older, now obsolete functions for converting host names and service names to and from binary and presentation formats. Although new programs should perform these conversions using the modern functions described earlier in this chapter, a knowledge of the obsolete functions is useful because we may encounter them in older code.

59.13.1 The *inet_aton()* and *inet_ntoa()* Functions

The *inet_aton()* and *inet_ntoa()* functions convert IPv4 addresses between dotted-decimal notation and binary form (in network byte order). These functions are nowadays made obsolete by *inet_pton()* and *inet_ntop()*.

The *inet_aton()* ("ASCII to network") function converts the dotted-decimal string pointed to by *str* into an IPv4 address in network byte order, which is returned in the *in_addr* structure pointed to by *addr*.

```
#include <arpa/inet.h>

int inet_aton(const char *str, struct in_addr *addr);
```

Returns 1 (true) if *str* is a valid dotted-decimal address, or 0 (false) on error

The *inet_aton()* function returns 1 if the conversion was successful, or 0 if *str* was invalid.

The numeric components of the string given to *inet_aton()* need not be decimal. They can be octal (specified by a leading 0) or hexadecimal (specified by a leading 0x or 0X). Furthermore, *inet_aton()* supports shorthand forms that allow an address to be specified using fewer than four numeric components. (See the *inet(3)* manual page for details.) The term *numbers-and-dots notation* is used for the more general address strings that employ these features.

SUSv3 doesn't specify *inet_aton()*. Nevertheless, this function is available on most implementations. On Linux, we must define one of the feature test macros `_BSD_SOURCE`, `_SVID_SOURCE`, or `_GNU_SOURCE` in order to obtain the declaration of *inet_aton()* from `<arpa/inet.h>`.

The *inet_ntoa()* ("network to ASCII") function performs the converse of *inet_aton()*.

```
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr addr);
```

Returns pointer to (statically allocated) dotted-decimal string version of *addr*

Given an *in_addr* structure (a 32-bit IPv4 address in network byte order), *inet_ntoa()* returns a pointer to a (statically allocated) string containing the address in dotted-decimal notation.

Because the string returned by *inet_ntoa()* is statically allocated, it is overwritten by successive calls.

59.13.2 The *gethostbyname()* and *gethostbyaddr()* Functions

The *gethostbyname()* and *gethostbyaddr()* functions allow conversion between hostnames and IP addresses. These functions are nowadays made obsolete by *getaddrinfo()* and *getnameinfo()*.

```
#include <netdb.h>

extern int h_errno;

struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, socklen_t len, int type);
```

Both return pointer to (statically allocated) *hostent* structure on success, or `NULL` on error

The *gethostbyname()* function resolves the hostname given in *name*, returning a pointer to a statically allocated *hostent* structure containing information about that hostname. This structure has the following form:

```
struct hostent {
    char  *h_name;              /* Official (canonical) name of host */
    char **h_aliases;           /* NULL-terminated array of pointers
                                   to alias strings */
    int    h_addrtype;          /* Address type (AF_INET or AF_INET6) */
    int    h_length;            /* Length (in bytes) of addresses pointed
                                   to by h_addr_list (4 bytes for AF_INET,
                                   16 bytes for AF_INET6) */
    char **h_addr_list;         /* NULL-terminated array of pointers to
                                   host IP addresses (in_addr or in6_addr
                                   structures) in network byte order */
};

#define h_addr  h_addr_list[0]
```

The *h_name* field returns the official name of the host, as a null-terminated string. The *h_aliases* fields points to an array of pointers to null-terminated strings containing aliases (alternative names) for this hostname.

The *h_addr_list* field is an array of pointers to IP address structures for this host. (A multihomed host has more than one address.) This list consists of either *in_addr* or *in6_addr* structures. We can determine the type of these structures from the *h_addrtype* field, which contains either `AF_INET` or `AF_INET6`, and their length from the *h_length* field. The *h_addr* definition is provided for backward compatibility with earlier implementations (e.g., 4.2BSD) that returned just one address in the *hostent* structure. Some existing code relies on this name (and thus is not multihomed-host aware).

With modern versions of *gethostbyname()*, *name* can also be specified as a numeric IP address string; that is, numbers-and-dots notation for IPv4 or hex-string notation for IPv6. In this case, no lookup is performed; instead, *name* is copied into the *h_name* field of the *hostent* structure, and *h_addr_list* is set to the binary equivalent of *name*.

The *gethostbyaddr()* function performs the converse of *gethostbyname()*. Given a binary IP address, it returns a *hostent* structure containing information about the host with that address.

On error (e.g., a name could not be resolved), both *gethostbyname()* and *gethostbyaddr()* return a `NULL` pointer and set the global variable *h_errno*. As the name suggests, this variable is analogous to *errno* (possible values placed in this variable are described in the *gethostbyname(3)* manual page), and the *herror()* and *hstrerror()* functions are analogous to *perror()* and *strerror()*.

The *herror()* function displays (on standard error) the string given in *str*, followed by a colon (`:`), and then a message for the current error in *h_errno*. Alternatively, we can use *hstrerror()* to obtain a pointer to a string corresponding to the error value specified in *err*.

```
#define _BSD_SOURCE           /* Or _SVID_SOURCE or _GNU_SOURCE */
#include <netdb.h>

void herror(const char *str);

const char *hstrerror(int err);
```

Returns pointer to *h_errno* error string corresponding to *err*

Listing 59-10 demonstrates the use of *gethostbyname()*. This program displays *hostent* information for each of the hosts named on its command line. The following shell session demonstrates the use of this program:

```
$ ./t_gethostbyname www.jambit.com
Canonical name: jamjam1.jambit.com
        alias(es):      www.jambit.com
        address type:   AF_INET
        address(es):    62.245.207.90
```

Listing 59-10: Using *gethostbyname()* to retrieve host information

sockets/t_gethostbyname.c

```
#define _BSD_SOURCE     /* To get hstrerror() declaration from <netdb.h> */
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct hostent *h;
    char **pp;
    char str[INET6_ADDRSTRLEN];

    for (argv++; *argv != NULL; argv++) {
        h = gethostbyname(*argv);
        if (h == NULL) {
            fprintf(stderr, "gethostbyname() failed for '%s': %s\n",
                    *argv, hstrerror(h_errno));
            continue;
        }

        printf("Canonical name: %s\n", h->h_name);

        printf("        alias(es):     ");
        for (pp = h->h_aliases; *pp != NULL; pp++)
            printf(" %s", *pp);
        printf("\n");
```

```
        printf("        address type:   %s\n",
                (h->h_addrtype == AF_INET) ? "AF_INET" :
                (h->h_addrtype == AF_INET6) ? "AF_INET6" : "???");

        if (h->h_addrtype == AF_INET || h->h_addrtype == AF_INET6) {
            printf("        address(es):   ");
            for (pp = h->h_addr_list; *pp != NULL; pp++)
                printf(" %s", inet_ntop(h->h_addrtype, *pp,
                                        str, INET6_ADDRSTRLEN));
            printf("\n");
        }
    }

    exit(EXIT_SUCCESS);
}
```

sockets/t_gethostbyname.c

59.13.3 The *getservbyname()* and *getservbyport()* Functions

The *getservbyname()* and *getservbyport()* functions retrieve records from the /etc/services file (Section 59.9). These functions are nowadays made obsolete by *getaddrinfo()* and *getnameinfo()*.

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);
```

Both return pointer to a (statically allocated) *servent* structure on success, or NULL on not found or error

The *getservbyname()* function looks up the record whose service name (or one of its aliases) matches *name* and whose protocol matches *proto*. The *proto* argument is a string such as *tcp* or *udp*, or it can be NULL. If *proto* is specified as NULL, any record whose service name matches *name* is returned. (This is usually sufficient since, where both UDP and TCP records with the same name exist in the /etc/services file, they normally have the same port number.) If a matching record is found, then *getservbyname()* returns a pointer to a statically allocated structure of the following type:

```
struct servent {
    char  *s_name;          /* Official service name */
    char **s_aliases;       /* Pointers to aliases (NULL-terminated) */
    int    s_port;          /* Port number (in network byte order) */
    char  *s_proto;         /* Protocol */
};
```

Typically, we call *getservbyname()* only in order to obtain the port number, which is returned in the *s_port* field.

The *getservbyport()* function performs the converse of *getservbyname()*. It returns a *servent* record containing information from the /etc/services record whose port number matches *port* and whose protocol matches *proto*. Again, we can specify *proto* as NULL, in which case the call will return any record whose port number matches

the one specified in *port*. (This may not return the desired result in the few cases mentioned above where the same port number maps to different service names in UDP and TCP.)

> An example of the use of the *getservbyname()* function is provided in the file `files/t_getservbyname.c` in the source code distribution for this book.

59.14 UNIX Versus Internet Domain Sockets

When writing applications that communicate over a network, we must necessarily use Internet domain sockets. However, when using sockets to communicate between applications on the same system, we have the choice of using either Internet or UNIX domain sockets. In the case, which domain should we use and why?

Writing an application using just Internet domain sockets is often the simplest approach, since it will work on both a single host and across a network. However, there are some reasons why we may choose to use UNIX domain sockets:

- On some implementations, UNIX domain sockets are faster than Internet domain sockets.
- We can use directory (and, on Linux, file) permissions to control access to UNIX domain sockets, so that only applications with a specified user or group ID can connect to a listening stream socket or send a datagram to a datagram socket. This provides a simple method of authenticating clients. With Internet domain sockets, we need to do rather more work if we wish to authenticate clients.
- Using UNIX domain sockets, we can pass open file descriptors and sender credentials, as summarized in Section 61.13.3.

59.15 Further Information

There is a wealth of printed and online resources on TCP/IP and the sockets API:

- The key book on network programming with the sockets API is [Stevens at al., 2004]. [Snader, 2000] adds some useful guidelines on sockets programming.
- [Stevens, 1994] and [Wright & Stevens, 1995] describe TCP/IP in detail. [Comer, 2000], [Comer & Stevens, 1999], [Comer & Stevens, 2000], [Kozierok, 2005], and [Goralksi, 2009] also provide good coverage of the same material.
- [Tanenbaum, 2002] provides general background on computer networks.
- [Herbert, 2004] describes the details of the Linux 2.6 TCP/IP stack.
- The GNU C library manual (online at *http://www.gnu.org/*) has an extensive discussion of the sockets API.
- The IBM Redbook, *TCP/IP Tutorial and Technical Overview*, provides lengthy coverage of networking concepts, TCP/IP internals, the sockets API, and a host of related topics. It is freely downloadable from *http://www.redbooks.ibm.com/*.
- [Gont, 2008] and [Gont, 2009b] provide security assessments of IPv4 and TCP.
- The Usenet newsgroup *comp.protocols.tcp-ip* is dedicated to questions related to the TCP/IP networking protocols.

- [Sarolahti & Kuznetsov, 2002] describes congestion control and other details of the Linux TCP implementation.
- Linux-specific information can be found in the following manual pages: *socket(7)*, *ip(7)*, *raw(7)*, *tcp(7)*, *udp(7)*, and *packet(7)*.
- See also the RFC list in Section 58.7.

59.16 Summary

Internet domain sockets allow applications on different hosts to communicate via a TCP/IP network. An Internet domain socket address consists of an IP address and a port number. In IPv4, an IP address is a 32-bit number; in IPv6, it is a 128-bit number. Internet domain datagram sockets operate over UDP, providing connectionless, unreliable, message-oriented communication. Internet domain stream sockets operate over TCP, and provide a reliable, bidirectional, byte-stream communication channel between two connected applications.

Different computer architectures use different conventions for representing data types. For example, integers may be stored in little-endian or big-endian form, and different computers may use different numbers of bytes to represent numeric types such as *int* or *long*. These differences mean that we need to employ some architecture-independent representation when transferring data between heterogeneous machines connected via a network. We noted that various marshalling standards exist to deal this problem, and also described a simple solution used by many applications: encoding all transmitted data in text form, with fields delimited by a designated character (usually a newline).

We looked at a range of functions that can be used to convert between (numeric) string representations of IP addresses (dotted-decimal for IPv4 and hex-string for IPv6) and their binary equivalents. However, it is generally preferable to use host and service names rather than numbers, since names are easier to remember and continue to be usable, even if the corresponding number is changed. We looked at various functions that convert host and service names to their numeric equivalents and vice versa. The modern function for translating host and service names into socket addresses is *getaddrinfo()*, but it is common to see the historical functions *gethostbyname()* and *getservbyname()* in existing code.

Consideration of hostname conversions led us into a discussion of DNS, which implements a distributed database for a hierarchical directory service. The advantage of DNS is that the management of the database is not centralized. Instead, local zone administrators update changes for the hierarchical component of the database for which they are responsible, and DNS servers communicate with one another in order to resolve a hostname.

59.17 Exercises

59-1. When reading large quantities of data, the *readLine()* function shown in Listing 59-1 is inefficient, since a system call is required to read each character. A more efficient interface would read a block of characters into a buffer and extract a line at a time from this buffer. Such an interface might consist of two functions. The first of these functions, which might be called *readLineBufInit(fd, &rlbuf)*, initializes the bookkeeping

data structure pointed to by *rlbuf*. This structure includes space for a data buffer, the size of that buffer, and a pointer to the next "unread" character in that buffer. It also includes a copy of the file descriptor given in the argument *fd*. The second function, *readLineBuf(&rlbuf)*, returns the next line from the buffer associated with *rlbuf*. If required, this function reads a further block of data from the file descriptor saved in *rlbuf*. Implement these two functions. Modify the programs in Listing 59-6 (`is_seqnum_sv.c`) and Listing 59-7 (`is_seqnum_cl.c`) to use these functions.

59-2. Modify the programs in Listing 59-6 (`is_seqnum_sv.c`) and Listing 59-7 (`is_seqnum_cl.c`) to use the *inetListen()* and *inetConnect()* functions provided in Listing 59-9 (`inet_sockets.c`).

59-3. Write a UNIX domain sockets library with an API similar to the Internet domain sockets library shown in Section 59.12. Rewrite the programs in Listing 57-3 (`us_xfr_sv.c`, on page 1168) and Listing 57-4 (`us_xfr_cl.c`, on page 1169) to use this library.

59-4. Write a network server that stores name-value pairs. The server should allow names to be added, deleted, modified, and retrieved by clients. Write one or more client programs to test the server. Optionally, implement some kind of security mechanism that allows only the client that created the name to delete it or to modify the value associated with it.

59-5. Suppose that we create two Internet domain datagram sockets, bound to specific addresses, and connect the first socket to the second. What happens if we create a third datagram socket and try to send (*sendto()*) a datagram via that socket to the first socket? Write a program to determine the answer.