11

FOR PUBLIC
RELEASE

Name and
Address Conversions

11.1 Introduction

All the examples so far in this text have used numeric addresses for the hosts (e.g., 206.6.226.33) and numeric port numbers to identify the servers (e.g., port 13 for the standard daytime server and port 9877 for our echo server). We should, however, use names instead of numbers for numerous reasons: Names are easier to remember; the numeric address can change but the name can remain the same; and with the move to IPv6, numeric addresses become much longer, making it much more error-prone to enter an address by hand. This chapter describes the functions that convert between names and numeric values: gethostbyname and gethostbyaddr to convert between hostnames and IPv4 addresses, and getservbyname and getservbyport to convert between service names and port numbers. It also describes two protocol-independent functions: getaddrinfo and getnameinfo, which convert between hostnames and IP addresses and between service names and port numbers.

11.2 Domain Name System (DNS)

The DNS is used primarily to map between hostnames and IP addresses. A hostname can be either a simple name, such as solaris or freebsd, or a fully qualified domain name (FQDN), such as solaris.unpbook.com.

Technically, an FQDN is also called an absolute name and must end with a period, but users often omit the ending period. The trailing period tells the resolver that this name is fully qualified and it doesn’t need to search its list of possible domains.
In this section, we will cover only the basics of the DNS that we need for network programming. Readers interested in additional details should consult Chapter 14 of TCPv1 and [Albitz and Liu 2001]. The additions required for IPv6 are in RFC 1886 [Thomson and Huitema 1995] and RFC 3152 [Bush 2001].

**Resource Records**

Entries in the DNS are known as resource records (RRs). There are only a few types of RRs that we are interested in.

**A**

An A record maps a hostname into a 32-bit IPv4 address. For example, here are the four DNS records for the host `freebsd` in the `unpbook.com` domain, the first of which is an A record:

```
freebsd IN A 12.106.32.254
IN AAAA 3ffe:b80:1f8d:1:a00:20ff:fea7:686b
IN MX 5 freebsd.unpbook.com.
IN MX 10 mailhost.unpbook.com.
```

**AAAA**

A AAAA record, called a “quad A” record, maps a hostname into a 128-bit IPv6 address. The term “quad A” was chosen because a 128-bit address is four times larger than a 32-bit address.

**PTR**

PTR records (called “pointer records”) map IP addresses into hostnames. For an IPv4 address, the 4 bytes of the 32-bit address are reversed, each byte is converted to its decimal ASCII value (0–255), and `in-addr.arpa` is then appended. The resulting string is used in the PTR query.

For an IPv6 address, the 32 4-bit nibbles of the 128-bit address are reversed, each nibble is converted to its corresponding hexadecimal ASCII value (0–9a–f), and `ip6.arpa` is appended.

For example, the two PTR records for our host `freebsd` would be `254.32.106.12.in-addr.arpa` and `b.6.8.6.7.a.e.f.f.0.2.0.0.a.0.1.0.0.0.d.8.f.1.0.8.b.0.e.f.f.3.ip6.arpa`.

Earlier standards specified that IPv6 addresses were looked up in the `ip6.int` domain. This was changed to `ip6.arpa` for consistency with IPv4. There will be a transition period during which both zones will be populated.

**MX**

An MX record specifies a host to act as a “mail exchanger” for the specified host. In the example for the host `freebsd` above, two MX records are provided: The first has a preference value of 5 and the second has a preference value of 10. When multiple MX records exist, they are used in order of preference, starting with the smallest value.

We do not use MX records in this text, but we mention them because they are used extensively in the real world.
CNAME  CNAME stands for “canonical name.” A common use is to assign CNAME records for common services, such as ftp and www. If people use these service names instead of the actual hostnames, it is transparent when a service is moved to another host. For example, the following could be CNAMEs for our host linux:

```
ftp IN CNAME linux.unpbook.com.
www IN CNAME linux.unpbook.com.
```

It is too early in the deployment of IPv6 to know what conventions administrators will use for hosts that support both IPv4 and IPv6. In our example earlier in this section, we specified both an A record and a AAAA record for our host freebsd. One possibility is to place both the A record and the AAAA record under the host’s normal name (as shown earlier) and create another RR whose name ends in -4 containing the A record, another RR whose name ends in -6 containing the AAAA record, and another RR whose name ends in -6ll containing a AAAA record with the host’s link-local address (which is sometimes handy for debugging purposes). All the records for another of our hosts are then

```
aix IN A 192.168.42.2
  IN AAAA 3ffe:b80:1f8d:2:204:acff:fe17:bf38
  IN MX 5 aix.unpbook.com.
  IN MX 10 mailhost.unpbook.com.
aix-4 IN A 192.168.42.2
aix-6 IN AAAA 3ffe:b80:1f8d:2:204:acff:fe17:bf38
aix-6ll IN AAAA fe80::204:acff:fe17:bf38
```

This gives us additional control over the protocol chosen by some applications, as we will see in the next chapter.

Resolvers and Name Servers

Organizations run one or more name servers, often the program known as BIND (Berkeley Internet Name Domain). Applications such as the clients and servers that we are writing in this text contact a DNS server by calling functions in a library known as the resolver. The common resolver functions are gethostbyname and gethostbyaddr, both of which are described in this chapter. The former maps a hostname into its IPv4 addresses, and the latter does the reverse mapping.

Figure 11.1 shows a typical arrangement of applications, resolvers, and name servers. We now write the application code. On some systems, the resolver code is contained in a system library and is link-edited into the application when the application is built. On others, there is a centralized resolver daemon that all applications share, and the system library code performs RPCs to this daemon. In either case, application code calls the resolver code using normal function calls, typically calling the functions gethostbyname and gethostbyaddr.

The resolver code reads its system-dependent configuration files to determine the location of the organization’s name servers. (We use the plural “name servers” because most organizations run multiple name servers, even though we show only one local
server in the figure. Multiple name servers are absolutely required for reliability and redundancy.) The file `/etc/resolv.conf` normally contains the IP addresses of the local name servers.

It might be nice to use the names of the name servers in the `/etc/resolv.conf` file, since the names are easier to remember and configure, but this introduces a chicken-and-egg problem of where to go to do the name-to-address conversion for the server that will do the name and address conversion!

The resolver sends the query to the local name server using UDP. If the local name server does not know the answer, it will normally query other name servers across the Internet, also using UDP. If the answers are too large to fit in a UDP packet, the resolver will automatically switch to TCP.

**DNS Alternatives**

It is possible to obtain name and address information without using the DNS. Common alternatives are static host files (normally the file `/etc/hosts`, as we describe in Figure 11.21), the Network Information System (NIS) or Lightweight Directory Access Protocol (LDAP). Unfortunately, it is implementation-dependent how an administrator configures a host to use the different types of name services. Solaris 2.x, HP-UX 10 and later, and FreeBSD 5.x and later use the file `/etc/nsswitch.conf`, and AIX uses the file `/etc/netsvc.conf`. BIND 9.2.2 supplies its own version named the Information Retrieval Service (IRS), which uses the file `/etc/irs.conf`. If a name server is to be used for hostname lookups, then all these systems use the file `/etc/resolv.conf` to specify the IP addresses of the name servers. Fortunately, these differences are normally hidden to the application programmer, so we just call the resolver functions such as `gethostbyname` and `gethostbyaddr`. 

![Diagram of name and address conversions](image-url)
11.3 gethostbyname Function

Host computers are normally known by human-readable names. All the examples that we have shown so far in this book have intentionally used IP addresses instead of names, so we know exactly what goes into the socket address structures for functions such as `connect` and `sendto`, and what is returned by functions such as `accept` and `recvfrom`. But, most applications should deal with names, not addresses. This is especially true as we move to IPv6, since IPv6 addresses (hex strings) are much longer than IPv4 dotted-decimal numbers. (The example AAAA record and `ip6.arpa` PTR record in the previous section should make this obvious.)

The most basic function that looks up a hostname is `gethostbyname`. If successful, it returns a pointer to a `hostent` structure that contains all the IPv4 addresses for the host. However, it is limited in that it can only return IPv4 addresses. See Section 11.6 for a function that handles both IPv4 and IPv6 addresses. The POSIX specification cautions that `gethostbyname` may be withdrawn in a future version of the spec.

It is unlikely that `gethostbyname` implementations will actually disappear until the whole Internet is using IPv6, which will be far in the future. However, withdrawing the function from the POSIX specification is a way to assert that it should not be used in new code. We encourage the use of `getaddrinfo` (Section 11.6) in new programs.

```c
#include <netdb.h>
struct hostent *gethostbyname(const char *hostname);
```

Returns: non-null pointer if OK, `NULL` on error with `h_errno` set

The non-null pointer returned by this function points to the following `hostent` structure:

```c
struct hostent {
    char *h_name; /* official (canonical) name of host */
    char **h_aliases; /* pointer to array of pointers to alias names */
    int h_addrtype; /* host address type: AF_INET */
    int h_length; /* length of address: 4 */
    char **h_addr_list; /* ptr to array of ptrs with IPv4 addr */
};
```

In terms of the DNS, `gethostbyname` performs a query for an A record. This function can return only IPv4 addresses.

Figure 11.2 shows the arrangement of the `hostent` structure and the information that it points to assuming the hostname that is looked up has two alias names and three IPv4 addresses. Of these fields, the official hostname and all the aliases are null-terminated C strings.

The returned `h_name` is called the *canonical* name of the host. For example, given the CNAME records shown in the previous section, the canonical name of the host `ftp.unpbook.com` would be `linux.unpbook.com`. Also, if we call `gethostbyname` from the host `aix` with an unqualified hostname, say `solaris`, the FQDN (`solaris.unpbook.com`) is returned as the canonical name.
Some versions of gethostbyname allow the *hostname* argument to be a dotted-decimal string. That is, a call of the form

```
hptr = gethostbyname("192.168.42.2");
```

will work. This code was added because the Rlogin client accepts only a hostname, calling *gethostbyname*, and will not accept a dotted-decimal string [Vixie 1996]. The POSIX specification permits, but does not require, this behavior, so a portable application cannot depend on it.

gethostbyname differs from the other socket functions that we have described in that it does not set *errno* when an error occurs. Instead, it sets the global integer *h_errno* to one of the following constants defined by including <netdb.h>:

- HOST_NOT_FOUND
- TRY_AGAIN
- NO_RECOVERY
- NO_DATA (identical to NO_ADDRESS)

The NO_DATA error means the specified name is valid, but it does not have an A record. An example of this is a hostname with only an MX record.

Most modern resolvers provide the function *hstrerror*, which takes an *h_errno* value as its only argument and returns a *const char * pointer to a description of the error. We show some examples of the strings returned by this function in the next example.

**Example**

Figure 11.3 shows a simple program that calls *gethostbyname* for any number of command-line arguments and prints all the returned information.
Section 11.3  

gethostbyname Function  

Figure 11.3  Call gethostbyname and print returned information.

8–14  gethostbyname is called for each command-line argument.

15–17  The official hostname is output followed by a list of alias names.

18–24  pptr points to the array of pointers to the individual addresses. For each address, we call inet_ntop and print the returned string.

We first execute the program with the name of our host aix, which has just one IPv4 address.

```
freebsd % hostent aix
official hostname: aix.unpbook.com
address: 192.168.42.2
```

Notice that the official hostname is the FQDN. Also notice that even though this host has an IPv6 address, only the IPv4 address is returned.

```c
#include "unp.h"

int main(int argc, char **argv)
{
  char *ptr, **pptr;
  char str[INET_ADDRSTRLEN];
  struct hostent *hptr;

  while (--argc > 0) {
    ptr = **argv;
    if ( (hptr = gethostbyname(ptr)) == NULL) {
      err_msg("gethostbyname error for host: %s: %s", ptr, hstrerror(h_errno));
      continue;
    }
    printf("official hostname: %s\n", hptr->h_name);
    for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
      printf("\talias: %s\n", *pptr);
    switch (hptr->h_addrtype) {
      case AF_INET:
        pptr = hptr->h_addr_list;
        for ( ; *pptr != NULL; pptr++)
          printf("\taddress: %s\n", Inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
        break;
      default:
        err_ret("unknown address type");
        break;
    }
  }
exit(0);
```
Next is a Web server with multiple IPv4 addresses.

```
freebsd % hostent cnn.com
official hostname: cnn.com
  address: 64.236.16.20
  address: 64.236.16.52
  address: 64.236.16.84
  address: 64.236.16.116
  address: 64.236.24.4
  address: 64.236.24.12
  address: 64.236.24.20
  address: 64.236.24.28
```

Next is a name that we showed in Section 11.2 as having a CNAME record.

```
freebsd % hostent www
official hostname: linux.unpbook.com
  alias: www.unpbook.com
  address: 206.168.112.219
```

As expected, the official hostname differs from our command-line argument.

To see the error strings returned by the `hstrerror` function, we first specify a non-existent hostname, and then a name that has only an MX record.

```
freebsd % hostent nosuchname.invalid
gethostbyname error for host: nosuchname.invalid: Unknown host
```

```
freebsd % hostent uunet.uu.net
gethostbyname error for host: uunet.uu.net: No address associated with name
```

### 11.4 gethostbyaddr Function

The function `gethostbyaddr` takes a binary IPv4 address and tries to find the hostname corresponding to that address. This is the reverse of `gethostbyname`.

```c
#include <netdb.h>

struct hostent *gethostbyaddr(const char *addr, socklen_t len, int family);
```

This function returns a pointer to the same `hostent` structure that we described with `gethostbyname`. The field of interest in this structure is normally `h_name`, the canonical hostname.

The `addr` argument is not a `char*`, but is really a pointer to an `in_addr` structure containing the IPv4 address. `len` is the size of this structure: 4 for an IPv4 address. The `family` argument is `AF_INET`.

In terms of the DNS, `gethostbyaddr` queries a name server for a PTR record in the `in-addr.arpa` domain.
Services, like hosts, are often known by names, too. If we refer to a service by its name in our code, instead of by its port number, and if the mapping from the name to port number is contained in a file (normally `/etc/services`), then if the port number changes, all we need to modify is one line in the `/etc/services` file instead of having to recompile the applications. The next function, `getservbyname`, looks up a service given its name.

The canonical list of port numbers assigned to services is maintained by the IANA at http://www.iana.org/assignments/port-numbers (Section 2.9). A given `/etc/services` file is likely to contain a subset of the IANA assignments.

```c
#include <netdb.h>

struct servent *getservbyname(const char *servname, const char *protoname);
```

Returns: non-null pointer if OK, NULL on error

This function returns a pointer to the following structure:

```c
struct servent {
    char *s_name; /* official service name */
    char **s_aliases; /* alias list */
    int s_port; /* port number, network-byte order */
    char *s_proto; /* protocol to use */
};
```

The service name `servname` must be specified. If a protocol is also specified (`protoname` is a non-null pointer), then the entry must also have a matching protocol. Some Internet services are provided using either TCP or UDP (for example, the DNS and all the services in Figure 2.18), while others support only a single protocol (e.g., FTP requires TCP). If `protoname` is not specified and the service supports multiple protocols, it is implementation-dependent as to which port number is returned. Normally this does not matter, because services that support multiple protocols often use the same TCP and UDP port number, but this is not guaranteed.

The main field of interest in the `servent` structure is the port number. Since the port number is returned in network byte order, we must not call `htons` when storing this into a socket address structure.

Typical calls to this function could be as follows:

```c
struct servent *sptr;

sptr = getservbyname("domain", "udp"); /* DNS using UDP */
sptr = getservbyname("ftp", "tcp"); /* FTP using TCP */
sptr = getservbyname("ftp", NULL); /* FTP using TCP */
sptr = getservbyname("ftp", "udp"); /* this call will fail */
```

Since FTP supports only TCP, the second and third calls are the same, and the fourth call will fail. Typical lines from the `/etc/services` file are
freebsd % grep -e ^ftp -e ^domain /etc/services

ftp-data 20/tcp #File Transfer [Default Data]
ftp 21/tcp #File Transfer [Control]
domain 53/tcp #Domain Name Server
domain 53/udp #Domain Name Server
ftp-agent 574/tcp #FTP Software Agent System
ftp-agent 574/udp #FTP Software Agent System
ftps-data 989/tcp # ftp protocol, data, over TLS/SSL
ftps 990/tcp # ftp protocol, control, over TLS/SSL

The next function, getservbyport, looks up a service given its port number and an optional protocol.

```c
#include <netdb.h>

struct servent *getservbyport(int port, const char *protoname);
```

Returns: non-null pointer if OK, NULL on error

The `port` value must be network byte ordered. Typical calls to this function could be as follows:

```c
struct servent *sptr;

sptr = getservbyport(htons(53), "udp"); /* DNS using UDP */
sptr = getservbyport(htons(21), "tcp"); /* FTP using TCP */
sptr = getservbyport(htons(21), NULL); /* FTP using TCP */
sptr = getservbyport(htons(21), "udp"); /* this call will fail */
```

The last call fails because there is no service that uses port 21 with UDP.

Be aware that a few port numbers are used with TCP for one service, but the same port number is used with UDP for a totally different service. For example, the following:

```freebsd % grep 514 /etc/services
shell 514/tcp cmd #like exec, but automatic
syslog 514/udp
```

shows that port 514 is used by the `rsh` command with TCP, but with the `syslog` daemon with UDP. Ports 512–514 have this property.

**Example: Using gethostbyname and getservbyname**

We can now modify our TCP daytime client from Figure 1.5 to use gethostbyname and getservbyname and take two command-line arguments: a hostname and a service name. Figure 11.4 shows our program. This program also shows the desired behavior of attempting to connect to all the IP addresses for a multihomed server, until one succeeds or all the addresses have been tried.
#include "unp.h"

int
main(int argc, char **argv)
{
    int sockfd, n;
    char recvline[MAXLINE + 1];
    struct sockaddr_in servaddr;
    struct in_addr **pptr;
    struct in_addr *inetaddrp[2];
    struct in_addr inetaddr;
    struct hostent *hp;
    struct servent *sp;

    if (argc != 3)
        err_quit("usage: daytimetcpcli1 <hostname> <service>");

    if ( (hp = gethostbyname(argv[1])) == NULL) {
        if (inet_aton(argv[1], &inetaddr) == 0) {
            err_quit("hostname error for %s: %s", argv[1],
                     hstrerror(h_errno));
        } else {
            inetaddrp[0] = &inetaddr;
            inetaddrp[1] = NULL;
            pptr = inetaddrp;
        }
    } else {
        pptr = (struct in_addr **) hp->h_addr_list;
    }

    if ( (sp = getservbyname(argv[2], "tcp")) == NULL)
        err_quit("getservbyname error for %s", argv[2]);

    for ( ; *pptr != NULL; pptr++) {
        sockfd = Socket(AF_INET, SOCK_STREAM, 0);
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = sp->s_port;
        memcpy(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));
        printf("trying %s\n", Sock_ntop((SA *) &servaddr, sizeof(servaddr)));
        if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) == 0)
            break; /* success */
        err_ret("connect error");
        close(sockfd);
    }

    if (*pptr == NULL)
        err_quit("unable to connect");

    while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
        recvline[n] = 0; /* null terminate */
        Fputs(recvline, stdout);
    }
    exit(0);
}
Call *gethostbyname* and *getservbyname*

The first command-line argument is a hostname, which we pass as an argument to *gethostbyname*, and the second is a service name, which we pass as an argument to *getservbyname*. Our code assumes TCP, and that is what we use as the second argument to *getservbyname*. If *gethostbyname* fails to look up the name, we try using the *inet_aton* function (Section 3.6) to see if the argument was an ASCII-format address. If it was, we construct a single-element list consisting of the corresponding address.

Try each server address

We now code the calls to *socket* and *connect* in a loop that is executed for every server address until a *connect* succeeds or the list of IP addresses is exhausted. After calling *socket*, we fill in an Internet socket address structure with the IP address and port of the server. While we could move the call to *bzero* and the subsequent two assignments out of the loop, for efficiency, the code is easier to read as shown. Establishing the connection with the server is rarely a performance bottleneck for a network client.

Call *connect*

*connect* is called, and if it succeeds, *break* terminates the loop. If the connection establishment fails, we print an error and close the socket. Recall that a descriptor that fails a call to *connect* must be closed and is no longer usable.

Check for failure

If the loop terminates because no call to *connect* succeeded, the program terminates.

Read server’s reply

Otherwise, we read the server’s response, terminating when the server closes the connection.

If we run this program specifying one of our hosts that is running the daytime server, we get the expected output.

```bash
freebsd % daytimetcpcli1 aix daytime
trying 192.168.42.2:13
Sun Jul 27 22:44:19 2003
```

What is more interesting is to run the program to a multihomed system that is not running the daytime server.

```bash
freebsd % daytimetcpcli1 gateway.tuc.noao.edu daytime
trying 140.252.108.1:13
connect error: Operation timed out
trying 140.252.1.4:13
connect error: Operation timed out
trying 140.252.104.1:13
connect error: Connection refused
unable to connect
```
11.6 *getaddrinfo* Function

The *gethostbyname* and *gethostbyaddr* functions only support IPv4. The API for resolving IPv6 addresses went through several iterations, as will be described in Section 11.20; the final result is the *getaddrinfo* function. The *getaddrinfo* function handles both name-to-address and service-to-port translation, and returns *sockaddr* structures instead of a list of addresses. These *sockaddr* structures can then be used by the socket functions directly. In this way, the *getaddrinfo* function hides all the protocol dependencies in the library function, which is where they belong. The application deals only with the socket address structures that are filled in by *getaddrinfo*. This function is defined in the POSIX specification.

The POSIX definition of this function comes from an earlier proposal by Keith Sklower for a function named *getconinfo*. This function was the result of discussions with Eric Allman, William Durst, Michael Karels, and Steven Wise, and from an early implementation written by Eric Allman. The observation that specifying a hostname and a service name would suffice for connecting to a service independent of protocol details was made by Marshall Rose in a proposal to X/Open.

```c
#include <netdb.h>

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

Returns: 0 if OK, nonzero on error (see Figure 11.7)

This function returns through the *result* pointer a pointer to a linked list of *addrinfo* structures, which is defined by including `<netdb.h>`.

```c
struct addrinfo {
    int ai_flags; /* AI_PASSIVE, AI_CANONNAME */
    int ai_family; /* AF_xxx */
    int ai_socktype; /* SOCK_xxx */
    int ai_protocol; /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
    socklen_t ai_addrlen; /* length of ai_addr */
    char *ai_canonname; /* ptr to canonical name for host */
    struct sockaddr *ai_addr; /* ptr to socket address structure */
    struct addrinfo *ai_next; /* ptr to next structure in linked list */
};
```

The *hostname* is either a hostname or an address string (dotted-decimal for IPv4 or a hex string for IPv6). The *service* is either a service name or a decimal port number string. (See also Exercise 11.4, where we want to allow an address string for the host or a port number string for the service.)

*hints* is either a null pointer or a pointer to an *addrinfo* structure that the caller fills in with hints about the types of information the caller wants returned. For example, if the specified service is provided for both TCP and UDP (e.g., the domain service, which refers to a DNS server), the caller can set the *ai_socktype* member of the *hints* structure to *SOCK_DGRAM*. The only information returned will be for datagram sockets.
The members of the `hints` structure that can be set by the caller are:

- `ai_flags` (zero or more `AI_XXX` values OR'ed together)
- `ai_family` (an `AF_XXX` value)
- `ai_socktype` (a `SOCK_XXX` value)
- `ai_protocol`

The possible values for the `ai_flags` member and their meanings are:

- **AI_PASSIVE** The caller will use the socket for a passive open.
- **AI_CANONNAME** Tells the function to return the canonical name of the host.
- **AI_NUMERICHOST** Prevents any kind of name-to-address mapping; the `hostname` argument must be an address string.
- **AI_NUMERICSERV** Prevents any kind of name-to-service mapping; the `service` argument must be a decimal port number string.
- **AI_V4MAPPED** If specified along with an `ai_family` of `AF_INET6`, then returns IPv4-mapped IPv6 addresses corresponding to A records if there are no available AAAA records.
- **AI_ALL** If specified along with `AI_V4MAPPED`, then returns IPv4-mapped IPv6 addresses in addition to any AAAA records belonging to the name.
- **AI_ADDRCONFIG** Only looks up addresses for a given IP version if there is one or more interface that is not a loopback interface configured with an IP address of that version.

If the `hints` argument is a null pointer, the function assumes a value of 0 for `ai_flags`, `ai_socktype`, and `ai_protocol`, and a value of `AF_UNSPEC` for `ai_family`.

If the function returns success (0), the variable pointed to by the `result` argument is filled in with a pointer to a linked list of `addrinfo` structures, linked through the `ai_next` pointer. There are two ways that multiple structures can be returned:

1. If there are multiple addresses associated with the `hostname`, one structure is returned for each address that is usable with the requested address family (the `ai_family` hint, if specified).
2. If the service is provided for multiple socket types, one structure can be returned for each socket type, depending on the `ai_socktype` hint. (Note that most `getaddrinfo` implementations consider a port number string to be implemented only by the socket type requested in `ai_socktype`; if `ai_socktype` is not specified, an error is returned instead.)

For example, if no hints are provided and if the domain service is looked up for a host with two IP addresses, four `addrinfo` structures are returned:
• One for the first IP address and a socket type of SOCK_STREAM
• One for the first IP address and a socket type of SOCK_DGRAM
• One for the second IP address and a socket type of SOCK_STREAM
• One for the second IP address and a socket type of SOCK_DGRAM

We show this example in Figure 11.5. There is no guaranteed order of the structures when multiple items are returned; that is, we cannot assume that TCP services will be returned before UDP services.

Although not guaranteed, an implementation should return the IP addresses in the same order as they are returned by the DNS. Some resolvers allow the administrator to specify an address sorting order in the /etc/resolv.conf file. IPv6 specifies address selection rules (RFC 3484 [Draves 2003]), which could affect the order of addresses returned by getaddrinfo.

The information returned in the addrinfo structures is ready for a call to socket and then either a call to connect or sendto (for a client), or bind (for a server). The arguments to socket are the members ai_family, ai_socktype, and ai_protocol. The second and third arguments to either connect or bind are ai_addr (a pointer to a socket address structure of the appropriate type, filled in by getaddrinfo) and ai_addrlen (the length of this socket address structure).

If the AI_CANONNAME flag is set in the hints structure, the ai_canonname member of the first returned structure points to the canonical name of the host. In terms of the DNS, this is normally the FQDN. Programs like telnet commonly use this flag to be able to print the canonical hostname of the system to which they are connecting, so that if the user supplied a shortcut or an alias, he or she knows what got looked up.

Figure 11.5 shows the returned information if we execute the following:

```c
struct addrinfo hints, *res;
bzero(&hints, sizeof(hints));
hints.ai_flags = AI_CANONNAME;
hints.ai_family = AF_INET;
getaddrinfo("freebsd4", "domain", &hints, &res);
```

In this figure, everything except the res variable is dynamically allocated memory (e.g., from malloc). We assume that the canonical name of the host freebsd4 is freebsd4.unpbook.com and that this host has two IPv4 addresses in the DNS.
Figure 11.5 Example of information returned by `getaddrinfo`. 

```c
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    int ai_addrlen;
    char *ai_canonname;
    struct sockaddr *ai_addr;
    struct addrinfo *ai_next;
} ai_flags
ai_family = AF_INET
ai_socktype = SOCK_DGRAM
ai_protocol = IPPROTO_UDP
ai_addrlen = 16
ai_canonname = freebsd4.unpbook.com
ai_addr = sockaddr_in
ai_next = NULL

sockaddr_in{
    int sin_family;
    __ Carey AS 10.2
    socklen_t sin_port;
    char *sin_addr;
    __ Carey AS 10.2
    int sin_zero[8];
} sin_family = AF_INET
sin_port = 53
sin_addr = 135.197.17.100

sockaddr_in{
    int sin_family;
    __ Carey AS 10.2
    socklen_t sin_port;
    char *sin_addr;
    __ Carey AS 10.2
    int sin_zero[8];
} sin_family = AF_INET
sin_port = 53
sin_addr = 172.24.37.94

sockaddr_in{
    int sin_family;
    __ Carey AS 10.2
    socklen_t sin_port;
    char *sin_addr;
    __ Carey AS 10.2
    int sin_zero[8];
} sin_family = AF_INET
sin_port = 53
sin_addr = 173.24.37.94
```
Port 53 is for the domain service. This port number will be in network byte order in the socket address structures. We also show the returned ai_protocol values as IPPROTO_TCP or IPPROTO_UDP. It would also be acceptable for getaddrinfo to return an ai_protocol of 0 for the two SOCK_STREAM structures if that is sufficient to specify TCP (it is not sufficient if the system implements SCTP, for example), and an ai_protocol of 0 for the two SOCK_DGRAM structures if the system doesn’t implement any other SOCK_DGRAM protocols for IP (as of this writing, none are yet standardized, but two are in development in the IETF). It is safest for getaddrinfo to always return the specific protocol.

Figure 11.6 summarizes the number of addrinfo structures returned for each address that is being returned, based on the specified service name (which can be a decimal port number) and any ai_socktype hint.

<table>
<thead>
<tr>
<th>ai_socktype hint</th>
<th>Service is a name, service provided by:</th>
<th>Service is a port number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TCP only</td>
<td>UDP only</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SOCK_STREAM</td>
<td>error</td>
<td>1</td>
</tr>
<tr>
<td>SOCK_DGRAM</td>
<td>1</td>
<td>error</td>
</tr>
<tr>
<td>SOCK_SEQPACKET</td>
<td>error</td>
<td>1</td>
</tr>
</tbody>
</table>

Multiple addrinfo structures are returned for each IP address only when no ai_socktype hint is provided and the service name is supported by multiple transport protocols (as indicated in the /etc/services file).

If we were to enumerate all 64 possible inputs to getaddrinfo (there are six input variables), many would be invalid and some would make little sense. Instead, we will look at the common cases.

- Specify the hostname and service. This is normal for a TCP or UDP client. On return, a TCP client loops through all returned IP addresses, calling socket and connect for each one, until the connection succeeds or until all addresses have been tried. We will show an example of this with our tcp_connect function in Figure 11.10.

For a UDP client, the socket address structure filled in by getaddrinfo would be used in a call to sendto or connect. If the client can tell that the first address doesn’t appear to work (either by receiving an error on a connected UDP socket or by experiencing a timeout on an unconnected socket), additional addresses can be tried.

If the client knows it handles only one type of socket (e.g., Telnet and FTP clients handle only TCP; TFTP clients handle only UDP), then the ai_socktype member of the hints structure should be specified as either SOCK_STREAM or SOCK_DGRAM.
• A typical server specifies the service but not the hostname, and specifies the AI_PASSIVE flag in the hints structure. The socket address structures returned should contain an IP address of INADDR_ANY (for IPv4) or IN6ADDR_ANY_INIT (for IPv6). A TCP server then calls socket, bind, and listen. If the server wants to malloc another socket address structure to obtain the client’s address from accept, the returned ai_addrlen value specifies this size.

A UDP server would call socket, bind, and then recvfrom. If the server wants to malloc another socket address structure to obtain the client’s address from recvfrom, the returned ai_addrlen value specifies this size.

As with the typical client code, if the server knows it only handles one type of socket, the ai_socktype member of the hints structure should be set to either SOCK_STREAM or SOCK_DGRAM. This avoids having multiple structures returned, possibly with the wrong ai_socktype value.

• The TCP servers that we have shown so far create one listening socket, and the UDP servers create one datagram socket. That is what we assume in the previous item. An alternate server design is for the server to handle multiple sockets using select or poll. In this scenario, the server would go through the entire list of structures returned by getaddrinfo, create one socket per structure, and use select or poll.

The problem with this technique is that one reason for getaddrinfo returning multiple structures is when a service can be handled by IPv4 and IPv6 (Figure 11.8). But, these two protocols are not completely independent, as we will see in Section 12.2. That is, if we create a listening IPv6 socket for a given port, there is no need to also create a listening IPv4 socket for that same port, because connections arriving from IPv4 clients are automatically handled by the protocol stack and by the IPv6 listening socket, assuming that the IPV6_V6ONLY socket option is not set.

Despite the fact that getaddrinfo is “better” than the gethostbyname and getservbyname functions (it makes it easier to write protocol-independent code; one function handles both the hostname and the service; and all the returned information is dynamically allocated, not statically allocated), it is still not as easy to use as it could be. The problem is that we must allocate a hints structure, initialize it to 0, fill in the desired fields, call getaddrinfo, and then traverse a linked list trying each one. In the next sections, we will provide some simpler interfaces for the typical TCP and UDP clients and servers that we will write in the remainder of this text.

getaddrinfo solves the problem of converting hostnames and service names into socket address structures. In Section 11.17, we will describe the reverse function, getnameinfo, which converts socket address structures into hostnames and service names.

11.7 gai_strerror Function

The nonzero error return values from getaddrinfo have the names and meanings shown in Figure 11.7. The function gai_strerror takes one of these values as an argument and returns a pointer to the corresponding error string.
Section 11.8

11.8 freeaddrinfo Function

All the storage returned by getaddrinfo, the addrinfo structures, the ai_addr structures, and the ai_canonname string are obtained dynamically (e.g., from malloc). This storage is returned by calling freeaddrinfo.

```
#include <netdb.h>

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

ai should point to the first addrinfo structure returned by getaddrinfo. All the structures in the linked list are freed, along with any dynamic storage pointed to by those structures (e.g., socket address structures and canonical hostnames).

Assume we call getaddrinfo, traverse the linked list of addrinfo structures, and find the desired structure. If we then try to save a copy of the information by copying just the addrinfo structure and calling freeaddrinfo, we have a lurking bug. The reason is that the addrinfo structure itself points to dynamically allocated memory (for the socket address structure and possibly the canonical name), and memory pointed to by our saved structure is returned to the system when freeaddrinfo is called and can be used for something else.

Making a copy of just the addrinfo structure and not the structures that it in turn points to is called a shallow copy. Copying the addrinfo structure and all the structures that it points to is called a deep copy.
11.9  **getaddrinfo Function: IPv6**

The POSIX specification defines the `getaddrinfo` function and the information it returns for both IPv4 and IPv6. We note the following points before summarizing these return values in Figure 11.8.

- **getaddrinfo** is dealing with two different inputs: the type of socket address structure the caller wants back and the type of records that should be searched for in the DNS or other database.

- The address family in the `hints` structure provided by the caller specifies the type of socket address structure that the caller expects to be returned. If the caller specifies `AF_INET`, the function must not return any `sockaddr_in6` structures; if the caller specifies `AF_INET6`, the function must not return any `sockaddr_in` structures.

- POSIX says that specifying `AF_UNSPEC` will return addresses that can be used with any protocol family that can be used with the hostname and service name. This implies that if a host has both AAAA records and A records, the AAAA records are returned as `sockaddr_in6` structures and the A records are returned as `sockaddr_in` structures. It makes no sense to also return the A records as IPv4-mapped IPv6 addresses in `sockaddr_in6` structures because no additional information is being returned: These addresses are already being returned in `sockaddr_in` structures.

- This statement in the POSIX specification also implies that if the `AI_PASSIVE` flag is specified without a hostname, then the IPv6 wildcard address (`IN6ADDR_ANY_INIT` or `0::0`) should be returned as a `sockaddr_in6` structure, along with the IPv4 wildcard address (INADDR_ANY or `0.0.0.0`), which is returned as a `sockaddr_in` structure. It also makes sense to return the IPv6 wildcard address first because we will see in Section 12.2 that an IPv6 server socket can handle both IPv6 and IPv4 clients on a dual-stack host.

- The address family specified in the `hint` structure’s `ai_family` member, along with the flags such as `AI_V4MAPPED` and `AI_ALL` specified in the `ai_flags` member, dictate the type of records that are searched for in the DNS (A and/or AAAA) and what type of addresses are returned (IPv4, IPv6, and/or IPv4-mapped IPv6). We summarize this in Figure 11.8.

- The hostname can also be either an IPv6 hex string or an IPv4 dotted-decimal string. The validity of this string depends on the address family specified by the caller. An IPv6 hex string is not acceptable if `AF_INET` is specified, and an IPv4 dotted-decimal string is not acceptable if `AF_INET6` is specified. But, if `AF_UNSPEC` is specified, either is acceptable and the appropriate type of socket address structure is returned.

  One could argue that if `AF_INET6` is specified, then a dotted-decimal string should be returned as an IPv4-mapped IPv6 address in a `sockaddr_in6` structure. But, another way to obtain this result is to prefix the dotted-decimal string with `0:ffff:`.

Figure 11.8 summarizes how we expect `getaddrinfo` to handle IPv4 and IPv6
addresses. The “Result” column is what we want returned to the caller, given the variables in the first three columns. The “Action” column is how we obtain this result.

<table>
<thead>
<tr>
<th>Hostname specified by caller</th>
<th>Address family specified by caller</th>
<th>Hostname string contains</th>
<th>Result</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-null hostname string; active or passive</td>
<td>AF_UNSPEC</td>
<td>hostname</td>
<td>All AAAA records returned as \texttt{sockaddr_in6{}\s as all A records returned as \texttt{sockaddr_in{}\s</td>
<td>AAAA record search and A record search</td>
</tr>
<tr>
<td></td>
<td>hex string</td>
<td>One \texttt{sockaddr_in6{}</td>
<td>\texttt{inet_pton(AF_INET6)}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dotted-decimal</td>
<td>One \texttt{sockaddr_in{}</td>
<td>\texttt{inet_pton(AF_INET)}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AF_INET6</td>
<td>hostname</td>
<td>All AAAA records returned as \texttt{sockaddr_in6{}\s</td>
<td>AAAA record search</td>
</tr>
<tr>
<td></td>
<td>If \texttt{ai_flags} contains \texttt{AI_V4MAPPED}, all AAAA records returned as \texttt{sockaddr_in6{}\s</td>
<td>AAAA record search if no results then A record search</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>else all A records returned as IPv4-mapped IPv6 \texttt{sockaddr_in6{}</td>
<td>A record search</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>hex string</td>
<td>One \texttt{sockaddr_in6{}</td>
<td>\texttt{inet_pton(AF_INET6)}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dotted-decimal</td>
<td>One \texttt{sockaddr_in{}</td>
<td>\texttt{inet_pton(AF_INET)}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AF_INET</td>
<td>hostname</td>
<td>All A records returned as \texttt{sockaddr_in{}\s</td>
<td>A record search</td>
</tr>
<tr>
<td></td>
<td>hex string</td>
<td>Looked up as hostname</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dotted-decimal</td>
<td>One \texttt{sockaddr_in{}</td>
<td>\texttt{inet_pton(AF_INET)}</td>
<td></td>
</tr>
<tr>
<td>null hostname string; passive</td>
<td>AF_UNSPEC</td>
<td>implied 0::0 implied 0.0.0.0</td>
<td>One \texttt{sockaddr_in6{} and \texttt{sockaddr_in{}</td>
<td>\texttt{inet_pton(AF_INET6)}</td>
</tr>
<tr>
<td></td>
<td>AF_INET6</td>
<td>implied 0::0</td>
<td>One \texttt{sockaddr_in6{}</td>
<td>\texttt{inet_pton(AF_INET6)}</td>
</tr>
<tr>
<td></td>
<td>AF_INET</td>
<td>implied 0.0.0</td>
<td>One \texttt{sockaddr_in{}</td>
<td>\texttt{inet_pton(AF_INET)}</td>
</tr>
<tr>
<td>null hostname string; active</td>
<td>AF_UNSPEC</td>
<td>implied 0::1 implied 127.0.0.0</td>
<td>One \texttt{sockaddr_in6{} and \texttt{sockaddr_in{}</td>
<td>\texttt{inet_pton(AF_INET6)}</td>
</tr>
<tr>
<td></td>
<td>AF_INET6</td>
<td>implied 0::1</td>
<td>One \texttt{sockaddr_in6{}</td>
<td>\texttt{inet_pton(AF_INET6)}</td>
</tr>
<tr>
<td></td>
<td>AF_INET</td>
<td>implied 127.0.0.0</td>
<td>One \texttt{sockaddr_in{}</td>
<td>\texttt{inet_pton(AF_INET)}</td>
</tr>
</tbody>
</table>

Figure 11.8 Summary of getaddrinfo and its actions and results.

Note that Figure 11.8 specifies only how getaddrinfo handles IPv4 and IPv6; that is, the number of addresses returned to the caller. The actual number of addrinfo structures returned to the caller also depends on the socket type specified and the service name, as summarized earlier in Figure 11.6.
11.10 getaddrinfo Function: Examples

We will now show some examples of getaddrinfo using a test program that lets us enter all the parameters: the hostname, service name, address family, socket type, and AI_CANONNAME and AI_PASSIVE flags. (We do not show this test program, as it is about 350 lines of uninteresting code. It is provided with the source code for the book, as described in the Preface.) The test program outputs information on the variable number of addrinfo structures that are returned, showing the arguments for a call to socket and the address in each socket address structure.

We first show the same example as in Figure 11.5.

```
freebsd % testga -f inet -c -h freebsd4 -s domain
socket(AF_INET, SOCK_DGRAM, 17), ai_canonname = freebsd4.unpbook.com
    address: 135.197.17.100:53
socket(AF_INET, SOCK_DGRAM, 17)
    address: 172.24.37.94:53
socket(AF_INET, SOCK_STREAM, 6), ai_canonname = freebsd4.unpbook.com
    address: 135.197.17.100:53
socket(AF_INET, SOCK_STREAM, 6)
    address: 172.24.37.94:53
```

The -f inet option specifies the address family, -c says to return the canonical name, -h freebsd4 specifies the hostname, and -s domain specifies the service name.

The common client scenario is to specify the address family, socket type (the -t option), hostname, and service name. The following example shows this for a multi-homed host with three IPv4 addresses:

```
freebsd % testga -f inet -t stream -h gateway.tuc.noao.edu -s daytime
socket(AF_INET, SOCK_STREAM, 6)
    address: 140.252.108.1:13
socket(AF_INET, SOCK_STREAM, 6)
    address: 140.252.1.4:13
socket(AF_INET, SOCK_STREAM, 6)
    address: 140.252.104.1:13
```

Next, we specify our host aix, which has both a AAAA record and an A record. We do not specify the address family, but we provide a service name of ftp, which is provided by TCP only.

```
freebsd % testga -f inet -t stream -h aix -s ftp
```
Since we didn’t specify the address family, and since we ran this example on a host that supports both IPv4 and IPv6, two structures are returned: one for IPv4 and one for IPv6.

Next, we specify the AI_PASSIVE flag (the -p option); we do not specify an address family or hostname (implying the wildcard address). We also specify a port number of 8888 and a stream socket.

Two structures are returned. Since we ran this on a host that supports IPv6 and IPv4 without specifying an address family, getaddrinfo returns the IPv6 wildcard address and the IPv4 wildcard address. The IPv6 structure is returned before the IPv4 structure, because we will see in Chapter 12 that an IPv6 client or server on a dual-stack host can communicate with either IPv6 or IPv4 peers.

### 11.11 host_serv Function

Our first interface to getaddrinfo does not require the caller to allocate a hints structure and fill it in. Instead, the two fields of interest, the address family and the socket type, are arguments to our host_serv function.

```c
#include "unp.h"

struct addrinfo *host_serv(const char *hostname, const char *service,
    int family, int socktype);
```

Returns: pointer to addrinfo structure if OK, NULL on error

Figure 11.9 shows the source code for this function.
The function initializes a `hints` structure, calls `getaddrinfo`, and returns a null pointer if an error occurs.

We will call this function from Figure 16.17 when we want to use `getaddrinfo` to obtain the host and service information, but we want to establish the connection ourselves.

### 11.12 tcp_connect Function

We will now write two functions that use `getaddrinfo` to handle most scenarios for the TCP clients and servers that we write. The first function, `tcp_connect`, performs the normal client steps: create a TCP socket and connect to a server.

```c
#include "unp.h"

int tcp_connect(const char *hostname, const char *service);
```

Returns: connected socket descriptor if OK, no return on error

Figure 11.10 shows the source code.
Figure 11.10 tcp_connect function: performs normal client steps.

Call getaddrinfo

getaddrinfo is called once and we specify the address family as AF_UNSPEC and the socket type as SOCK_STREAM.

Try each addrinfo structure until success or end of list

Each returned IP address is then tried. socket and connect are called. It is not a fatal error for socket to fail, as this could happen if an IPv6 address is returned but the host kernel does not support IPv6. If connect succeeds, a break is made out of the loop. Otherwise, when all the addresses have been tried, the loop also terminates. freeaddrinfo returns all the dynamic memory.
This function (and our other functions that provide a simpler interface to \texttt{getaddrinfo} in the following sections) terminates if either \texttt{getaddrinfo} fails or no call to \texttt{connect} succeeds. The only return is upon success. It would be hard to return an error code (one of the \texttt{EAI}xx constants) without adding another argument. This means that our wrapper function is trivial.

\begin{verbatim}
int Tcp_connect(const char *host, const char *serv)
{
    return(tcp_connect(host, serv));
}
\end{verbatim}

Nevertheless, we still call our wrapper function instead of \texttt{tcp_connect}, to maintain consistency with the remainder of the text.

The problem with the return value is that descriptors are non-negative, but we do not know whether the \texttt{EAI}xx values are positive or negative. If these values were positive, we could return the negative of these values if \texttt{getaddrinfo} fails, but we also have to return some other negative value to indicate that all the structures were tried without success.

**Example: Daytime Client**

Figure 11.11 shows our daytime client from Figure 1.5 recoded to use \texttt{tcp_connect}.

\begin{verbatim}
1 #include "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5    int sockfd, n;
6    char recvline[MAXLINE + 1];
7    socklen_t len;
8    struct sockaddr_storage ss;
9    if (argc != 3)
10       err_quit
11       ("usage: daytimetcpcli <hostname/IPaddress> <service/port#>");
12    sockfd = Tcp_connect(argv[1], argv[2]);
13    len = sizeof(ss);
14    Getpeername(sockfd, (SA *) &ss, &len);
15    printf("connected to %s\n", Sock_ntop_host((SA *) &ss, len));
16    while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
17       recvline[n] = 0; /* null terminate */
18       Puts(recvline, stdout);
19    }
20    exit(0);
21 }
\end{verbatim}

Figure 11.11 Daytime client recoded to use \texttt{tcp_connect}.
Command-line arguments

We now require a second command-line argument to specify either the service name or the port number, which allows our program to connect to other ports.

Connect to server

All the socket code for this client is now performed by tcp_connect.

Print server’s address

We call getpeername to fetch the server’s protocol address and print it. We do this to verify the protocol being used in the examples we are about to show.

Note that tcp_connect does not return the size of the socket address structure that was used for the connect. We could have added a pointer argument to return this value, but one design goal for this function was to reduce the number of arguments compared to getaddrinfo. What we do instead is use a sockaddr_storage socket address structure, which is large enough to hold and fulfills the alignment constraints of any socket address type the system supports.

This version of our client works with both IPv4 and IPv6, while the version in Figure 1.5 worked only with IPv4 and the version in Figure 1.6 worked only with IPv6. You should also compare our new version with Figure E.12, which we coded to use gethostbyname and getservbyname to support both IPv4 and IPv6.

We first specify the name of a host that supports only IPv4.

```
freebsd % daytimetcpcli linux daytime
connected to 206.168.112.96
Sun Jul 27 23:06:24 2003
```

Next, we specify the name of a host that supports both IPv4 and IPv6.

```
freebsd % daytimetcpcli aix daytime
connected to 3ffe:b80:1f8d:2:204:acff:fe17:bf38
Sun Jul 27 23:17:13 2003
```

The IPv6 address is used because the host has both a AAAA record and an A record, and as noted in Figure 11.8, since tcp_connect sets the address family to AF_UNSPEC, AAAA records are searched for first, and only if this fails is a search made for an A record.

In the next example, we force the use of the IPv4 address by specifying the hostname with our -4 suffix, which we noted in Section 11.2 is our convention for the hostname with only A records.

```
freebsd % daytimetcpcli aix-4 daytime
connected to 192.168.42.2
Sun Jul 27 23:17:48 2003
```
11.13 tcp_listen Function

Our next function, tcp_listen, performs the normal TCP server steps: create a TCP socket, bind the server’s well-known port, and allow incoming connection requests to be accepted. Figure 11.12 shows the source code.

```
#include "unp.h"

int tcp_listen(const char *hostname, const char *service, socklen_t *addrlenp);

Returns: connected socket descriptor if OK, no return on error
```

We initialize an addrinfo structure with our hints: AI_PASSIVE, since this function is for a server, AF_UNSPEC for the address family, and SOCK_STREAM. Recall from Figure 11.8 that if a hostname is not specified (which is common for a server that wants to bind the wildcard address), the AI_PASSIVE and AF_UNSPEC hints will cause two socket address structures to be returned: the first for IPv6 and the next for IPv4 (assuming a dual-stack host).

Create socket and bind address

The socket and bind functions are called. If either call fails, we just ignore this addrinfo structure and move on to the next one. As stated in Section 7.5, we always set the SO_REUSEADDR socket option for a TCP server.

Check for failure

If all the calls to socket and bind fail, we print an error and terminate. As with our tcp_connect function in the previous section, we do not try to return an error from this function.

The socket is turned into a listening socket by listen.

Return size of socket address structure

If the addrlenp argument is non-null, we return the size of the protocol addresses through this pointer. This allows the caller to allocate memory for a socket address structure to obtain the client’s protocol address from accept. (See Exercise 11.7 also.)

Example: Daytime Server

Figure 11.13 shows our daytime server from Figure 4.11, recoded to use tcp_listen.

Require service name or port number as command-line argument

We require a command-line argument to specify either the service name or port number. This makes it easier to test our server, since binding port 13 for the daytime server requires superuser privileges.

Create listening socket

tcp_listen creates the listening socket. We pass a NULL pointer as the third argument because we don’t care what size address structure the address family uses; we will use sockaddr_storage.
#include "unp.h"

int tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
{
  int listenfd, n;
  const int on = 1;
  struct addrinfo hints, *res, *ressave;
  bzero(&hints, sizeof(struct addrinfo));
  hints.ai_flags = AI_PASSIVE;
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
    err_quit("tcp_listen error for %s, %s: %s", host, serv, gai_strerror(n));
  ressave = res;
  do {
    listenfd =
    socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (listenfd < 0)
      continue; /* error, try next one */
    Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
    if (bind(listenfd, res->ai_addr, res->ai_addrlen) == 0)
      break; /* success */
    Close(listenfd); /* bind error, close and try next one */
  } while ( (res = res->ai_next) != NULL);
  if (res == NULL) /* errno from final socket() or bind() */
    err_sys("tcp_listen error for %s, %s", host, serv);
  Listen(listenfd, LISTENQ);
  if (addrlenp)
    *addrlenp = res->ai_addrlen; /* return size of protocol address */
  freeaddrinfo(ressave);
  return (listenfd);
}

Server loop

accept waits for each client connection. We print the client address by calling
sock_ntop. In the case of either IPv4 or IPv6, this function prints the IP address and
port number. We could use the function getnameinfo (Section 11.17) to try to obtain
the hostname of the client, but that involves a PTR query in the DNS, which can take
some time, especially if the PTR query fails. Section 14.8 of TCPv3 notes that on a busy
Web server, almost 25% of all clients connecting to that server did not have PTR records
```c
#include "unp.h"
#include <time.h>

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t len;
    char buff[MAXLINE];
    time_t ticks;
    struct sockaddr_storage cliaddr;

    if (argc != 2)
        err_quit("usage: daytimetcpsrv1 <service or port#>");

    listenfd = Tcp_listen(NULL, argv[1], NULL);

    for ( ; ; ) {
        len = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &len);
        printf("connection from %s\n", Sock_ntop((SA *) &cliaddr, len));

        ticks = time(NULL);
        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
        Write(connfd, buff, strlen(buff));
        Close(connfd);
    }
}
```

**Figure 11.13** Daytime server recoded to use `tcp_listen` (see also Figure 11.14).

in the DNS. Since we do not want a server (especially an iterative server) to wait seconds for a PTR query, we just print the IP address and port.

### Example: Daytime Server with Protocol Specification

There is a slight problem with Figure 11.13: The first argument to `tcp_listen` is a null pointer, which combined with the address family of `AF_UNSPEC` that `tcp_listen` specifies might cause `getaddrinfo` to return a socket address structure with an address family other than what is desired. For example, the first socket address structure returned will be for IPv6 on a dual-stack host (Figure 11.8), but we might want our server to handle only IPv4.

Clients do not have this problem since the client must always specify either an IP address or a hostname. Client applications normally allow the user to enter this as a command-line argument. This gives us the opportunity to specify a hostname that is associated with a particular type of IP address (recall our `-4` and `-6` hostnames in Section 11.2), or to specify either an IPv4 dotted-decimal string (forcing IPv4) or an IPv6 hex string (forcing IPv6).
But there is a simple technique for servers that lets us force a given protocol on a server, either IPv4 or IPv6: Allow the user to enter either an IP address or a hostname as a command-line argument to the program and pass this to getaddrinfo. In the case of an IP address, an IPv4 dotted-decimal string differs from an IPv6 hex string. The following calls to inet_pton either fail or succeed, as indicated:

```
inet_pton(AF_INET, "0.0.0.0", &foo);    /* succeeds */
inet_pton(AF_INET, "0::0", &foo);       /* fails */
inet_pton(AF_INET6, "0.0.0.0", &foo);    /* fails */
inet_pton(AF_INET6, "0::0", &foo);      /* succeeds */
```

Therefore, if we change our servers to accept an optional argument, and if we enter

```
% server
```

it defaults to IPv6 on a dual-stack host, but entering

```
% server 0.0.0.0
```

explicitly specifies IPv4 and

```
% server 0::0
```

explicitly specifies IPv6.

Figure 11.14 shows this final version of our daytime server.

Handle command-line arguments

The only change from Figure 11.13 is the handling of the command-line arguments, allowing the user to specify either a hostname or an IP address for the server to bind, in addition to a service name or port.

We first start this server with an IPv4 socket and then connect to the server from clients on two other hosts on the local subnet.

```
freebsd % daytimetcpsrv2 0.0.0.0 9999
connection from 192.168.42.2:32961
connection from 192.168.42.3:1389
```

Now we start the server with an IPv6 socket.

```
freebsd % daytimetcpsrv2 0::0 9999
connection from [3ffe:b80:1f8d:2:204:acff:fe17:bf38]:32964
connection from [::ffff:192.168.42.2]:32967
connection from [::ffff:192.168.42.3]:49602
```

The first connection is from the host aix using IPv6 and the second is from the host macosx using IPv6. The next two connections are from the hosts aix and macosx, but using IPv4, not IPv6. We can tell this because the client’s addresses returned by accept are both IPv4-mapped IPv6 addresses.

What we have just shown is that an IPv6 server running on a dual-stack host can handle either IPv4 or IPv6 clients. The IPv4 client addresses are passed to the IPv6 server as IPv4-mapped IPv6 addresses, as we will discuss in Section 12.2.
Our functions that provide a simpler interface to getaddrinfo change with UDP because we provide one client function that creates an unconnected UDP socket, and another in the next section that creates a connected UDP socket.

```c
#include "unp.h"
int udp_client(const char *hostname, const char *service, 
    struct sockaddr **saptr, socklen_t *lenp);
```

This function creates an unconnected UDP socket, returning three items. First, the return value is the socket descriptor. Second, `saptr` is the address of a pointer (declared
Section 11.14  udp_client Function

by the caller) to a socket address structure (allocated dynamically by udp_client), and in that structure, the function stores the destination IP address and port for future calls to sendto. The size of the socket address structure is returned in the variable pointed to by lenp. This final argument cannot be a null pointer (as we allowed for the final argument to tcp_listen) because the length of the socket address structure is required in any calls to sendto and recvfrom.

Figure 11.15 shows the source code for this function.

```c
#include "unp.h"

int udp_client(const char *host, const char *serv, SA **saptr, socklen_t *lenp)
{
    int sockfd, n;
    struct addrinfo hints, *res, *ressave;
    bzero(&hints, sizeof(struct addrinfo));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;
    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
        err_quit("udp_client error for %s, %s: %s", host, serv, gai_strerror(n));
    ressave = res;
    do {
        sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
        if (sockfd >= 0)
            break; /* success */
    } while ( (res = res->ai_next) != NULL);
    if (res == NULL) /* errno set from final socket() */
        err_sys("udp_client error for %s, %s", host, serv);
    *saptr = Malloc(res->ai_addrlen);
    memcpy(*saptr, res->ai_addr, res->ai_addrlen);
    *lenp = res->ai_addrlen;
    freeaddrinfo(ressave);
    return (sockfd);
}
```

Figure 11.15  udp_client function: creates an unconnected UDP socket.

getaddrinfo converts the hostname and service arguments. A datagram socket is created. Memory is allocated for one socket address structure, and the socket address structure corresponding to the socket that was created is copied into the memory.

Example: Protocol-Independent Daytime Client

We now recode our daytime client from Figure 11.11 to use UDP and our udp_client function. Figure 11.16 shows the protocol-independent source code.
```c
#include "unp.h"

int main(int argc, char **argv)
{
    int sockfd, n;
    char recvline[MAXLINE + 1];
    socklen_t salen;
    struct sockaddr *sa;

    if (argc != 3)
        err_quit("usage: daytimeudpcli1 <hostname/IPaddress> <service/port#>");
    sockfd = Udp_client(argv[1], argv[2], (void **) &sa, &salen);
    printf("sending to %s\n", Sock_ntop_host(sa, salen));
    Sendto(sockfd, ",", 1, 0, sa, salen); /* send 1-byte datagram */
    n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
    recvline[n] = '\0'; /* null terminate */
    Fputs(recvline, stdout);
    exit(0);
}
```

Figure 11.16  UDP daytime client using our udp_client function.

We call our udp_client function and then print the IP address and port of the server to which we will send the UDP datagram. We send a one-byte datagram and then read and print the reply.

We need to send only a zero-byte UDP datagram, as what triggers the daytime server’s response is just the arrival of a datagram, regardless of its length and contents. But, many SVR4 implementations do not allow a zero-length UDP datagram.

We run our client specifying a hostname that has a AAAA record and an A record. Since the structure with the AAAA record is returned first by getaddrinfo, an IPv6 socket is created.

```bash
freebsd % daytimeudpcli1 aix daytime
sending to 3ffe:b80:1f8d:2:204:acff:fe17:bf38
Sun Jul 27 23:21:12 2003
```

Next, we specify the dotted-decimal address of the same host, resulting in an IPv4 socket.

```bash
freebsd % daytimeudpcli1 192.168.42.2 daytime
sending to 192.168.42.2
Sun Jul 27 23:21:40 2003
```
11.15 udp_connect Function

Our udp_connect function creates a connected UDP socket.

```c
#include "unp.h"

int udp_connect(const char *hostname, const char *service);

Returns: connected socket descriptor if OK, no return on error
```

With a connected UDP socket, the final two arguments required by udp_client are no longer needed. The caller can call write instead of sendto, so our function need not return a socket address structure and its length.

Figure 11.17 shows the source code.

```c
1 #include "unp.h"

2 int
3 udp_connect(const char *host, const char *serv)
4 {
5     int sockfd, n;
6     struct addrinfo hints, *res, *ressave;
7     bzero(&hints, sizeof(struct addrinfo));
8     hints.ai_family = AF_UNSPEC;
9     hints.ai_socktype = SOCK_DGRAM;
10    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11       err_quit("udp_connect error for %s, %s: %s",
12              host, serv, gai_strerror(n));
13     ressave = res;
14     do {
15         sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16         if (sockfd < 0)
17             continue; /* ignore this one */
18         if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
19             break; /* success */
20         Close(sockfd); /* ignore this one */
21     } while ( (res = res->ai_next) != NULL);
22    if (res == NULL) /* errno set from final connect() */
23       err_sys("udp_connect error for %s, %s", host, serv);
24    freeaddrinfo(ressave);
25    return (sockfd);
26 }
```

This function is nearly identical to tcp_connect. One difference, however, is that the call to connect with a UDP socket does not send anything to the peer. If something is wrong (the peer is unreachable or there is no server at the specified port), the caller does not discover that until it sends a datagram to the peer.
11.16 `udp_server` Function

Our final UDP function that provides a simpler interface to `getaddrinfo` is `udp_server`.

```c
#include "unp.h"
int udp_server(const char *hostname, const char *service, socklen_t *lenptr);
```

Returns: unconnected socket descriptor if OK, no return on error

The arguments are the same as for `tcp_listen`: an optional `hostname`, a required `service` (so its port number can be bound), and an optional pointer to a variable in which the size of the socket address structure is returned.

Figure 11.18 shows the source code.

```c
#include "unp.h"
int udp_server(const char *host, const char *serv, socklen_t *addrlenp)
{
  int sockfd, n;
  struct addrinfo hints, *res, *ressave;
  bzero(&hints, sizeof(struct addrinfo));
  hints.ai_flags = AI_PASSIVE;
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_DGRAM;
  if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
    err_quit("udp_server error for %s, %s: %s",
     host, serv, gai_strerror(n));
  ressave = res;
  do {
    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (sockfd < 0)
      continue; /* error - try next one */
    if (bind(sockfd, res->ai_addr, res->ai_addrlen) == 0)
      break; /* success */
    Close(sockfd); /* bind error - close and try next one */
  } while ( (res = res->ai_next) != NULL);
  if (res == NULL) /* errno from final socket() or bind() */
    err_sys("udp_server error for %s, %s", host, serv);
  if (addrlenp)
    *addrlenp = res->ai_addrlen; /* return size of protocol address */
  freeaddrinfo(ressave);
  return (sockfd);
}
```

Figure 11.18  `udp_server` function: creates an unconnected socket for a UDP server.
This function is nearly identical to tcp_listen, but without the call to listen. We set the address family to AF_UNSPEC, but the caller can use the same technique that we described with Figure 11.14 to force a particular protocol (IPv4 or IPv6).

We do not set the SO_REUSEADDR socket option for the UDP socket because this socket option can allow multiple sockets to bind the same UDP port on hosts that support multicasting, as we described in Section 7.5. Since there is nothing like TCP’s TIME_WAIT state for a UDP socket, there is no need to set this socket option when the server is started.

Example: Protocol-Independent Daytime Server

Figure 11.19 shows our daytime server, modified from Figure 11.14 to use UDP.

```c
#include "unp.h"
#include <time.h>

int main(int argc, char **argv)
{
    int sockfd;
    ssize_t n;
    char buf[MAXLINE];
    time_t ticks;
    socklen_t len;
    struct sockaddr_storage cliaddr;

    if (argc == 2)
        sockfd = Udp_server(NULL, argv[1], NULL);
    else if (argc == 3)
        sockfd = Udp_server(argv[1], argv[2], NULL);
    else
        err_quit("usage: daytimeudpsrv [ <host> ] <service or port>");

    for ( ; ; ) {
        len = sizeof(cliaddr);
        n = Recvfrom(sockfd, buf, MAXLINE, 0, (SA *) &cliaddr, &len);
        printf("datagram from %s\n", Sock_ntop((SA *) &cliaddr, len));
        ticks = time(NULL);
        snprintf(buf, sizeof(buf), ".24s\r\n", ctime(&ticks));
        Sendto(sockfd, buf, strlen(buf), 0, (SA *) &cliaddr, len);
    }
}
```

Figure 11.19 Protocol-independent UDP daytime server.
11.17 getNameinfo Function

This function is the complement of getaddrinfo: It takes a socket address and returns a character string describing the host and another character string describing the service. This function provides this information in a protocol-independent fashion; that is, the caller does not care what type of protocol address is contained in the socket address structure, as that detail is handled by the function.

```c
#include <netdb.h>

#include <netdb.h>

int getnameinfo(const struct sockaddr *sockaddr, socklen_t addrlen,
               char *host, socklen_t hostlen,
               char *serv, socklen_t servlen, int flags);
```

Returns: 0 if OK, nonzero on error (see Figure 11.7)

`sockaddr` points to the socket address structure containing the protocol address to be converted into a human-readable string, and `addrlen` is the length of this structure. This structure and its length are normally returned by `accept`, `recvfrom`, `getsockname`, or `getpeername`.

The caller allocates space for the two human-readable strings: `host` and `hostlen` specify the host string, and `serv` and `servlen` specify the service string. If the caller does not want the host string returned, a `hostlen` of 0 is specified. Similarly, a `servlen` of 0 specifies not to return information on the service.

The difference between `sock_ntop` and `getnameinfo` is that the former does not involve the DNS and just returns a printable version of the IP address and port number. The latter normally tries to obtain a name for both the host and service.

Figure 11.20 shows the six flags that can be specified to change the operation of `getnameinfo`.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NI_DGRAM</td>
<td>Datagram service</td>
</tr>
<tr>
<td>NI_NAMEREQD</td>
<td>Return an error if name cannot be resolved from address</td>
</tr>
<tr>
<td>NI_NOFQDN</td>
<td>Return only hostname portion of FQDN</td>
</tr>
<tr>
<td>NI_NUMERICHOST</td>
<td>Return numeric string for hostname</td>
</tr>
<tr>
<td>NI_NUMERICSCOPE</td>
<td>Return numeric string for scope identifier</td>
</tr>
<tr>
<td>NI_NUMERICSERV</td>
<td>Return numeric string for service name</td>
</tr>
</tbody>
</table>

Figure 11.20 flags for `getnameinfo`.

NI_DGRAM should be specified when the caller knows it is dealing with a datagram socket. The reason is that given only the IP address and port number in the socket address structure, `getnameinfo` cannot determine the protocol (TCP or UDP). There are a few port numbers that are used for one service with TCP and a completely different service with UDP. An example is port 514, which is the rsh service with TCP, but the syslog service with UDP.

NI_NAMEREQD causes an error to be returned if the hostname cannot be resolved
using the DNS. This can be used by servers that require the client’s IP address to be mapped into a hostname. These servers then take this returned hostname and call getaddrinfo, and then verify that one of the returned addresses is the address in the socket address structure.

NI_NOFQDN causes the returned hostname to be truncated at the first period. For example, if the IP address in the socket address structure was 192.168.42.2, gethostbyaddr would return a name of aix.unpbook.com. But if this flag was specified to getnameinfo, it would return the hostname as just aix.

NI_NUMERICHOST tells getnameinfo not to call the DNS (which can take time). Instead, the numeric representation of the IP address is returned as a string, probably by calling inet_ntop. Similarly, NI_NUMERICSERV specifies that the decimal port number is to be returned as a string instead of looking up the service name, and NI_NUMERICSCOPE specifies that the numeric form of the scope identifier is to be returned instead of its name. Servers should normally specify NI_NUMERICSERV because the client port numbers typically have no associated service name—they are ephemeral ports.

The logical OR of multiple flags can be specified if they make sense together (e.g., NI_DGRAM and NI_NUMERICHOST).

11.18 Re-entrant Functions

The gethostbyname function from Section 11.3 presents an interesting problem that we have not yet examined in the text: It is not re-entrant. We will encounter this problem in general when we deal with threads in Chapter 26, but it is interesting to examine the problem now (without having to deal with the concept of threads) and to see how to fix it.

First, let us look at how the function works. If we look at its source code (which is easy since the source code for the entire BIND release is publicly available), we see that one file contains both gethostbyname and gethostbyaddr, and the file has the following general outline:

```c
static struct hostent host; /* result stored here */

struct hostent *
gethostbyname(const char *hostname)
{
    return(gethostbyname2(hostname, family));
}

struct hostent *
gethostbyname2(const char *hostname, int family)
{
    /* call DNS functions for A or AAAA query */
    /* fill in host structure */
    return(&host);
}
```
We highlight the `static` storage class specifier of the result structure because that is the basic problem. The fact that these three functions share a single `host` variable presents yet another problem that we will discuss in Exercise 11.1. (`gethostbyname2` was introduced with the IPv6 support in BIND 4.9.4. It has since been deprecated; see Section 11.20 for more detail. We will ignore the fact that `gethostbyname2` is involved when we call `gethostbyname`, as that doesn’t affect this discussion.)

The re-entrancy problem can occur in a normal Unix process that calls `gethostbyname` or `gethostbyaddr` from both the main flow of control and from a signal handler. When the signal handler is called (say it is a `SIGALRM` signal that is generated once per second), the main flow of control of the process is temporarily stopped and the signal handling function is called. Consider the following:

```c
main()
{
    struct hostent *hptr;
    ...
    signal(SIGALRM, sig_alrm);
    ...
    hptr = gethostbyname( ... );
    ...
}

void sig_alrm(int signo)
{
    struct hostent *hptr;
    ...
    hptr = gethostbyname( ... );
    ...
}
```

If the main flow of control is in the middle of `gethostbyname` when it is temporarily stopped (say the function has filled in the `host` variable and is about to return), and the signal handler then calls `gethostbyname`, since only one copy of the variable `host` exists in the process, it is reused. This overwrites the values that were calculated for the call from the main flow of control with the values calculated for the call from the signal handler.
If we look at the name and address conversion functions presented in this chapter, along with the `inet_XXX` functions from Chapter 4, we note the following:

- Historically, `gethostbyname`, `gethostbyaddr`, `getservbyname`, and `getservbyport` are not re-entrant because all return a pointer to a static structure.

  Some implementations that support threads (Solaris 2.x) provide re-entrant versions of these four functions with names ending with the `_r` suffix, which we will describe in the next section.

  Alternately, some implementations that support threads (HP-UX 10.30 and later) provide re-entrant versions of these functions using thread-specific data (Section 26.5).

- `inet_pton` and `inet_ntop` are always re-entrant.

- Historically, `inet_ntoa` is not re-entrant, but some implementations that support threads provide a re-entrant version that uses thread-specific data.

- `getaddrsinfo` is re-entrant only if it calls re-entrant functions itself; that is, if it calls re-entrant versions of `gethostbyname` for the hostname and `getservbyname` for the service name. One reason that all the memory for the results is dynamically allocated is to allow it to be re-entrant.

- `getnameinfo` is re-entrant only if it calls re-entrant functions itself; that is, if it calls re-entrant versions of `gethostbyaddr` to obtain the hostname and `getservbyport` to obtain the service name. Notice that both result strings (for the hostname and the service name) are allocated by the caller to allow this re-entrancy.

A similar problem occurs with the variable `errno`. Historically, there has been a single copy of this integer variable per process. If a process makes a system call that returns an error, an integer error code is stored in this variable. For example, when the function named `close` in the standard C library is called, it might execute something like the following pseudocode:

- Put the argument to the system call (an integer descriptor) into a register
- Put a value in another register indicating the `close` system call is being called
- Invoke the system call (switch to the kernel with a special instruction)
- Test the value of a register to see if an error occurred
- If no error, `return(0)`
- Store the value of some other register into `errno`
- `return(-1)`

First, notice that if an error does not occur, the value of `errno` is not changed. That is why we cannot look at the value of `errno` unless we know that an error has occurred (normally indicated by the function returning `-1`).
Assume a program tests the return value of the close function and then prints the value of errno if an error occurred, as in the following:

```c
if (close(fd) < 0) {
    fprintf(stderr, "close error, errno = %d\n", errno)
    exit(1);
}
```

There is a small window of time between the storing of the error code into errno when the system call returns and the printing of this value by the program, during which another thread of execution within this process (i.e., a signal handler) can change the value of errno. For example, if, when the signal handler is called, the main flow of control is between close and fprintf and the signal handler calls some other system call that returns an error (say write), then the errno value stored from the write system call overwrites the value stored by the close system call.

In looking at these two problems with regard to signal handlers, one solution to the problem with gethostbyname (returning a pointer to a static variable) is to not call nonre-entrant functions from a signal handler. The problem with errno (a single global variable that can be changed by the signal handler) can be avoided by coding the signal handler to save and restore the value of errno in the signal handler as follows:

```c
void
sig_alrm(int signo)
{
    int errno_save;
    errno_save = errno; /* save its value on entry */
    if (write( ... ) != nbytes)
        fprintf(stderr, "write error, errno = %d\n", errno);
    errno = errno_save; /* restore its value on return */
}
```

In this example code, we also call fprintf, a standard I/O function, from the signal handler. This is yet another re-entrancy problem because many versions of the standard I/O library are non-re-entrant: Standard I/O functions should not be called from signal handlers.

We will revisit this problem of re-entrancy in Chapter 26 and we will see how threads handle the problem of the errno variable. The next section describes some re-entrant versions of the hostname functions.

### 11.19 gethostbyname_r and gethostbyaddr_r Functions

There are two ways to make a nonre-entrant function such as gethostbyname re-entrant.

1. Instead of filling in and returning a static structure, the caller allocates the structure and the re-entrant function fills in the caller’s structure. This is the technique used in going from the non-re-entrant gethostbyname to the re-entrant gethostbyname_r. But, this solution gets more complicated because not only
must the caller provide the hostent structure to fill in, but this structure also points to other information: the canonical name, the array of alias pointers, the alias strings, the array of address pointers, and the addresses (e.g., Figure 11.2). The caller must provide one large buffer that is used for this additional information and the hostent structure that is filled in then contains numerous pointers into this other buffer. This adds at least three arguments to the function: a pointer to the hostent structure to fill in, a pointer to the buffer to use for all the other information, and the size of this buffer. A fourth additional argument is also required: a pointer to an integer in which an error code can be stored, since the global integer h_errno can no longer be used. (The global integer h_errno presents the same re-entrancy problem that we described with errno.)

This technique is also used by getnameinfo and inet_ntop.

2. The re-entrant function calls malloc and dynamically allocates the memory. This is the technique used by getaddrinfo. The problem with this approach is that the application calling this function must also call freeaddrinfo to free the dynamic memory. If the free function is not called, a memory leak occurs: Each time the process calls the function that allocates the memory, the memory use of the process increases. If the process runs for a long time (a common trait of network servers), the memory usage just grows and grows over time.

We will now discuss the Solaris 2.x re-entrant functions for name-to-address and address-to-name resolution.

```c
#include <netdb.h>

struct hostent *gethostbyname_r(const char *hostname,
                                 struct hostent *result,
                                 char *buf, int buflen, int *h_errnop);

struct hostent *gethostbyaddr_r(const char *addr, int len,
                                 int type,
                                 struct hostent *result,
                                 char *buf, int buflen, int *h_errnop);
```

Both return: non-null pointer if OK, NULL on error

Four additional arguments are required for each function. result is a hostent structure allocated by the caller. It is filled in by the function. On success, this pointer is also the return value of the function.

buf is a buffer allocated by the caller and buflen is its size. This buffer will contain the canonical hostname, the alias pointers, the alias strings, the address pointers, and the actual addresses. All the pointers in the structure pointed to by result point into this buffer. How big should this buffer be? Unfortunately, all that most man pages say is something vague like, “The buffer must be large enough to hold all of the data associated with the host entry.” Current implementations of gethostbyname can return up to 35 alias pointers and 35 address pointers, and internally use an 8192-byte buffer to hold alias names and addresses. So, a buffer size of 8192 bytes should be adequate.

If an error occurs, the error code is returned through the h_errnop pointer, not
Unfortunately, this problem of re-entrancy is even worse than it appears. First, there is no standard regarding re-entrancy and gethostbyname and gethostbyaddr. The POSIX specification says that gethostbyname and gethostbyaddr need not be re-entrant. Unix 98 just says that these two functions need not be thread-safe.

Second, there is no standard for the _r functions. What we have shown in this section (for example purposes) are two of the _r functions provided by Solaris 2.x. Linux provides similar _r functions, except that instead of returning the hostent as the return value of the function, the hostent is returned using a value-result parameter as the next to last function argument. It returns the success of the lookup as the return value from the function as well as in the h_errno argument. Digital Unix 4.0 and HP-UX 10.30 have versions of these functions with different arguments. The first two arguments for gethostbyname_r are the same as the Solaris version, but the remaining three arguments for the Solaris version are combined into a new hostent_data structure (which must be allocated by the caller), and a pointer to this structure is the third and final argument. The normal functions gethostbyname and gethostbyaddr in Digital Unix 4.0 and HP-UX 10.30 are re-entrant by using thread-specific data (Section 26.5). An interesting history of the development of the Solaris 2.x _r functions is in [Maslen 1997].

Lastly, while a re-entrant version of gethostbyname may provide safety from different threads calling it at the same time, this says nothing about the re-entrancy of the underlying resolver functions.

### 11.20 Obsolete IPv6 Address Lookup Functions

While IPv6 was being developed, the API to request the lookup of an IPv6 address went through several iterations. The resulting API was complicated and not sufficiently flexible, so it was deprecated in RFC 2553 [Gilligan et al. 1999]. RFC 2553 introduced its own new functions, which were finally simply replaced by getaddrinfo and getnameinfo in RFC 3493 [Gilligan et al. 2003]. This section briefly describes some of the old API to assist in the conversion of programs using the old API.

#### The RES_USE_INET6 Constant

Since gethostbyname doesn’t have an argument to specify what address family is of interest (like getaddrinfo’s hints.ai_family struct entry), the first revision of the API used the RES_USE_INET6 constant, which had to be added to the resolver flags using a private, internal interface. This API was not very portable since systems that used a different internal resolver interface had to mimic the BIND resolver interface to provide it.

Enabling RES_USE_INET6 caused gethostbyname to look up AAAA records first, and only look up A records if a name had no AAAA records. Since the hostent structure only has one address length field, gethostbyname could only return either IPv6 or IPv4 addresses, but not both.

Enabling RES_USE_INET6 also caused gethostbyname2 to return IPv4 addresses as IPv4-mapped IPv6 addresses. We will describe gethostbyname2 next.
The `gethostbyname2` Function

The `gethostbyname2` function adds an address family argument to `gethostbyname`.

```c
#include <sys/socket.h>
#include <netdb.h>
struct hostent *gethostbyname2(const char *name, int af);
```

Returns: non-null pointer if OK, NULL on error with `h_errno` set.

When the `af` argument is `AF_INET`, `gethostbyname2` behaves just like `gethostbyname`, looking up and returning IPv4 addresses. When the `af` argument is `AF_INET6`, `gethostbyname2` looks up and returns only AAAA records for IPv6 addresses.

The `getipnodebyname` Function

RFC 2553 [Gilligan et al. 1999] deprecated `RES_USE_INET6` and `gethostbyname2` because of the global nature of the `RES_USE_INET6` flag and the wish to provide more control over the returned information. It introduced the `getipnodebyname` function to solve some of these problems.

```c
#include <sys/socket.h>
#include <netdb.h>
struct hostent *getipnodebyname(const char *name, int af, int flags, int *error_num);
```

Returns: non-null pointer if OK, NULL on error with `error_num` set.

This function returns a pointer to the same `hostent` structure that we described with `gethostbyname`. The `af` and `flags` arguments map directly to `getaddrinfo`'s `hints.ai_family` and `hints.ai_flags` arguments. For thread safety, the return value is dynamically allocated, so it must be freed with the `freehostent` function.

```c
#include <netdb.h>
void freehostent(struct hostent *ptr);
```

The `getipnodebyname` and its matching `getipnodebyaddr` functions are deprecated by RFC 3493 [Gilligan et al. 2003] in favor of `getaddrinfo` and `getnameinfo`. 
11.21 Other Networking Information

Our focus in this chapter has been on hostnames and IP addresses and service names and their port numbers. But looking at the bigger picture, there are four types of information (related to networking) that an application might want to look up: hosts, networks, protocols, and services. Most lookups are for hosts (gethostbyname and gethostbyaddr), with a smaller number for services (getservbyname and getservbyaddr), and an even smaller number for networks and protocols.

All four types of information can be stored in a file and three functions are defined for each of the four types:

1. A getXXXent function that reads the next entry in the file, opening the file if necessary.
2. A setXXXent function that opens (if not already open) and rewinds the file.
3. An endXXXent function that closes the file.

Each of the four types of information defines its own structure, and the following definitions are provided by including the `<netdb.h>` header: the hostent, netent, protoent, and servent structures.

In addition to the three get, set, and end functions, which allow sequential processing of the file, each of the four types of information provides some keyed lookup functions. These functions go through the file sequentially (calling the getXXXent function to read each line), but instead of returning each line to the caller, these functions look for an entry that matches an argument. These keyed lookup functions have names of the form getXXXbyYYY. For example, the two keyed lookup functions for the host information are gethostbyname (look for an entry that matches a hostname) and gethostbyaddr (look for an entry that matches an IP address). Figure 11.21 summarizes this information.

<table>
<thead>
<tr>
<th>Information</th>
<th>Data file</th>
<th>Structure</th>
<th>Keyed lookup functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hosts</td>
<td>/etc/hosts</td>
<td>hostent</td>
<td>gethostbyaddr, gethostbyname</td>
</tr>
<tr>
<td>Networks</td>
<td>/etc/networks</td>
<td>netent</td>
<td>getnetbyaddr, getnetbyname</td>
</tr>
<tr>
<td>Protocols</td>
<td>/etc/protocols</td>
<td>protoent</td>
<td>getprotobyname, getprotobynumber</td>
</tr>
<tr>
<td>Services</td>
<td>/etc/services</td>
<td>servent</td>
<td>getservbyname, getservbyport</td>
</tr>
</tbody>
</table>

Figure 11.21 Four types of network-related information.

How does this apply when the DNS is being used? First, only the host and network information is available through the DNS. The protocol and service information is always read from the corresponding file. We mentioned earlier in this chapter (with Figure 11.1) that different implementations employ different ways for the administrator to specify whether to use the DNS or a file for the host and network information.
Second, if the DNS is being used for the host and network information, then only the keyed lookup functions make sense. You cannot, for example, use gethostent and expect to sequence through all entries in the DNS! If gethostent is called, it reads only the /etc/hosts file and avoids the DNS.

Although the network information can be made available through the DNS, few people set this up. [Albitz and Liu 2001] describes this feature. Typically, however, administrators build and maintain an /etc/networks file and it is used instead of the DNS. The netstat program with the -i option uses this file, if present, and prints the name for each network. However, classless addressing (Section A.4) makes these functions fairly useless, and these functions do not support IPv6 at all, so new applications should avoid using network names.

11.22 Summary

The set of functions that an application calls to convert a hostname into an IP address and vice versa is called the resolver. The two functions gethostbyname and gethostbyaddr are the historical entry points. With the move to IPv6 and threaded programming models, the getaddrinfo and getnameinfo functions are more useful, with the ability to resolve IPv6 addresses and their thread-safe calling conventions.

The commonly used function dealing with service names and port numbers is getservbyname, which takes a service name and returns a structure containing the port number. This mapping is normally contained in a text file. Additional functions exist to map protocol names into protocol numbers and network names into network numbers, but these are rarely used.

Another alternative that we have not mentioned is calling the resolver functions directly, instead of using gethostbyname and gethostbyaddr. One program that invokes the DNS this way is sendmail, which searches for an MX record, something that the gethostbyXXX functions cannot do. The resolver functions have names that begin with res; the res_init function is an example. A description of these functions and an example program that calls them can be found in Chapter 15 of [Albitz and Liu 2001] and typing man resolver should display the man pages for these functions.

Exercises

11.1 Modify the program in Figure 11.3 to call gethostbyaddr for each returned address, and then print the h_name that is returned. First run the program specifying a hostname with just one IP address and then run the program specifying a hostname that has more than one IP address. What happens?

11.2 Fix the problem shown in the preceding exercise.

11.3 Run Figure 11.4 specifying a service name of chargen.
11.4 Run Figure 11.4 specifying a dotted-decimal IP address as the hostname. Does your resolver allow this? Modify Figure 11.4 to allow a dotted-decimal IP address as the hostname and a decimal port number string as the service name. In testing the IP address for either a dotted-decimal string or a hostname, in what order should these two tests be performed?

11.5 Modify Figure 11.4 to work with either IPv4 or IPv6.

11.6 Modify Figure 8.9 to query the DNS and compare the returned IP address with all the destination host’s IP addresses. That is, call gethostbyaddr using the IP address returned by recvfrom, followed by gethostbyname to find all the IP addresses for the host.

11.7 In Figure 11.12, the caller must pass a pointer to an integer to obtain the size of the protocol address. If the caller does not do this (i.e., passes a null pointer as the final argument), how can the caller still obtain the actual size of the protocol’s addresses?

11.8 Modify Figure 11.14 to call getnameinfo instead of sock_ntop. What flags should you pass to getnameinfo?

11.9 In Section 7.5, we discussed port stealing with the SO_REUSEADDR socket option. To see how this works, build the protocol-independent UDP daytime server in Figure 11.19. Start one instance of the server in one window, binding the wildcard address and some port of your choosing. Start a client in another window and verify that this server is handling the client (note the printf in the server). Next, start another instance of the server in another window, this time binding one of the host’s unicast addresses and the same port as the first server. What problem do you immediately encounter? Fix this problem and restart this second server. Start a client, send a datagram, and verify that the second server has stolen the port from the first server. If possible, start the second server again from a different login account on the first server to see if the stealing still succeeds. Some vendors will not allow the second bind unless the user ID is the same as that of the process that has already bound the port.

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

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