11

Advanced Name and Address Conversions

11.1 Introduction

The two functions described in Chapter 9, `gethostbyname` and `gethostbyaddr`, are protocol dependent. When using the former, we must know which member of the socket address structure to move the result into (e.g., the `sin_addr` member for IPv4 or the `sin6_addr` member for IPv6), and when calling the latter, we must know which member contains the binary address. This chapter begins with the new Posix.1g `getaddrinfo` function that provides protocol independence for our applications. We cover its complement, `getnameinfo`, later in the chapter.

We then use this function and develop six functions of our own that handle the typical scenarios for TCP and UDP clients and servers. We use these functions throughout the remainder of the text instead of calling `getaddrinfo` directly.

The functions `gethostbyname` and `gethostbyaddr` are also nice examples of functions that are nonreentrant. We show why this is so and describe some replacement functions that avoid this problem. Reentrancy is a problem that we come back to in Chapter 23, but we are able to show and explain the problem now, without having to understand the details of threads.

We finish the chapter showing our complete implementation of `getaddrinfo`. This lets us understand more about the function: how it operates, what it returns, and its interaction with IPv4 and IPv6.

11.2 `getaddrinfo` Function

The `getaddrinfo` function hides all of 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 Posix.1g.

The Posix.1g definition of this function comes from an earlier proposal by Keith Sklower for a function named `getconninfo`. 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.

```
#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 11.3)

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

```
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 */
  size_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. (Recall our solution to Exercise 9.6 where we allowed 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 that 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` (`AI_PASSIVE`, `AI_CANONNAME`),
- `ai_family` (an `AF_`*xxx* value),
- `ai_socktype` (a `SOCK_`*xxx* value), and
- `ai_protocol`.

The `AI_PASSIVE` flag indicates that the socket will be used for a passive open, and the `AI_CANONNAME` flag tells the function to return the canonical name of the host.

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.

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, and
- one for the second IP address and a socket type of SOCK_DGRAM.

We show a picture of this example in Figure 11.1. There is no guaranteed order of the structures when multiple items are returned; that is, we cannot assume that TCP services are 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. For example, many DNS servers sort the returned addresses so that if the host sending the query and the name server are on the same network, then addresses on that shared network are returned first. Also, newer versions of BIND allow the resolver to specify an address sorting order in the /etc/resolv.conf file.

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.

Figure 11.1 shows the returned information if we execute

```
struct addrinfo     hints, *res;

bzero(&hints, sizeof(hints));
hints.ai_flags = AI_CANONNAME;
hints.ai_family = AF_INET;

getaddrinfo("bsdi", "domain", &hints, &res);
```

In this figure everything other than the res variable is dynamically allocated memory (e.g., from malloc). We assume that the canonical name of the host bsdi is bsdi.kohala.com and that this host has two IPv4 addresses in the DNS (Figure 1.16).

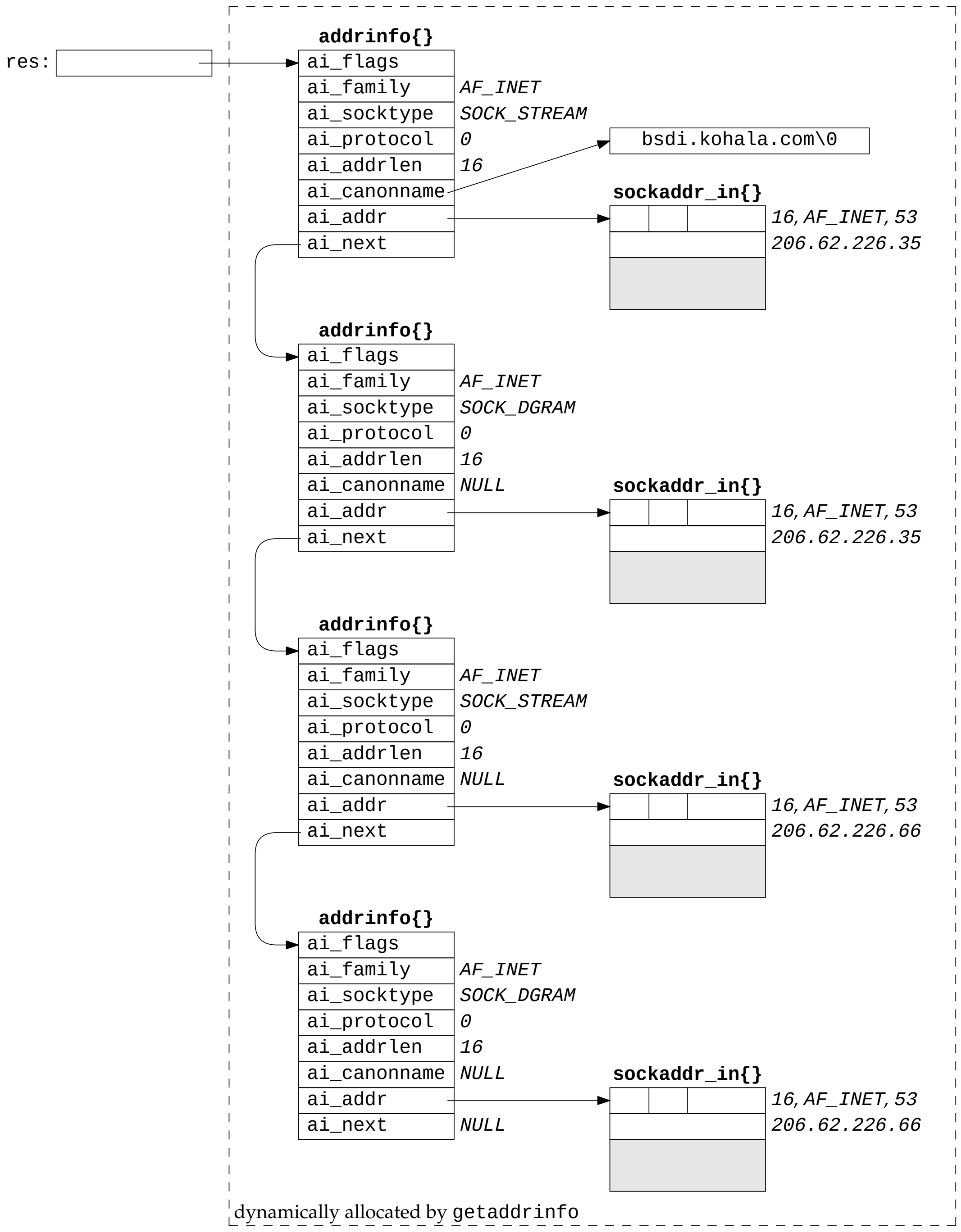

Figure 11.1 Example of information returned by `getaddrinfo`.

Port 53 is for the domain service, and realize that this port number will be in network byte order in the socket address structures. We also show the returned ai_protocol values as 0 since the combination of the ai_family and ai_socktype completely specifies the protocol for TCP and UDP. It would also be OK for getaddrinfo to return an ai_protocol of IPPROTO_TCP for the two SOCK_STREAM structures, and an ai_protocol of IPPROTO_UDP for the two SOCK_DGRAM structures.

Figure 11.2 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.

ai_socktype hint	Service is a name, service provided by:			Service is a port number
	TCP only	UDP only	TCP and UDP	
0	1	1	2	2
SOCK_STREAM	1	error	1	1
SOCK_DGRAM	error	1	1	1

Figure 11.2 Number of addrinfo structures returned per IP address.

Multiple addrinfo structures are returned for each IP address only when no ai_socktype hint is provided and either the service name is supported by TCP and UDP (as indicated in the /etc/services file) or a port number is specified.

If we enumerated all 64 possible inputs to getaddrinfo (there are six input variables), many would be invalid and some 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 show an example of this with our tcp_connect function in Figure 11.6.

  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 an error on a connected UDP socket or 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 *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`. In this scenario the server would go through the entire list of structures returned by `getaddrinfo`, create one socket per structure, and use `select`.

  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 11.4). But these two protocols are not completely independent, as we saw in Section 10.2. 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.

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 provide some simpler interfaces for the typical TCP and UDP clients and servers that we write in the remainder of this text.

`getaddrinfo` solves the problem of converting hostnames and service names into socket address structures. In Section 11.13 we describe the reverse function, `getnameinfo`, which converts socket address structures into hostnames and service names. In Section 11.16 we provide an implementation of `getaddrinfo`, `getnameinfo`, and `freeaddrinfo`.

11.3 `gai_strerror` Function

The nonzero error return values from `getaddrinfo` have the names and meanings shown in Figure 11.3. The function `gai_strerror` takes one of these values as an argument and returns a pointer to the corresponding error string.

```
#include <netdb.h>

char *gai_strerror(int error);

                        Returns: pointer to string describing error message
```

Constant	Description
EAI_ADDRFAMILY	address family for *hostname* not supported
EAI_AGAIN	temporary failure in name resolution
EAI_BADFLAGS	invalid value for ai_flags
EAI_FAIL	nonrecoverable failure in name resolution
EAI_FAMILY	ai_family not supported
EAI_MEMORY	memory allocation failure
EAI_NODATA	no address associated with *hostname*
EAI_NONAME	*hostname* nor *service* provided, or not known
EAI_SERVICE	*service* not supported for ai_socktype
EAI_SOCKTYPE	ai_socktype not supported
EAI_SYSTEM	system error returned in errno

Figure 11.3 Nonzero error return constants from getaddrinfo.

11.4 freeaddrinfo Function

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

```
#include <netdb.h>

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

ai should point to the first of the addrinfo structures 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 then call 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*.

11.5 getaddrinfo Function: IPv6 and Unix Domain

Although Posix.1g defines the getaddrinfo function, it says nothing about IPv6 at all. The interaction between this function, the resolver (especially the RES_USE_INET6 option; recall Figure 9.5), and IPv6 is nontrivial. We note the following points before summarizing these interactions in Figure 11.4.

- `getaddrinfo` is dealing with two different inputs: what type of socket address structure does the caller want back and what type of records should be searched for in the DNS.
- 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 and if the caller specifies `AF_INET6`, the function must not return any `sockaddr_in` structures.
- Posix.1g says that specifying `AF_UNSPEC` shall 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 `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 as no additional information is being returned: these addresses are already being returned in `sockaddr_in` structures.
- This statement in Posix.1g 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), returned as a `sockaddr_in` structure. It also makes sense to return the IPv6 wildcard address first because we saw in Section 10.2 that an IPv6 server socket can handle both IPv6 and IPv4 clients on a dual-stack host.
- The resolver's `RES_USE_INET6` option along with which function is called (`gethostbyname` or `gethostbyname2`) dictates the type of records that are searched for in the DNS (A or AAAA) and what type of addresses are returned (IPv4, IPv6, or IPv4-mapped IPv6). We summarized this in Figure 9.5.
- 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 either is acceptable if `AF_UNSPEC` is specified, and the appropriate type of socket address structure 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 11.4 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 and we show the code that performs this action in our implementation of `getaddrinfo` in Section 11.16.

Hostname specified by caller	Address family specified by caller	Hostname string contains	Result	Action
nonnull hostname string; active or passive	AF_UNSPEC	hostname	all AAAA records returned as sockaddr_in6{}s *and* all A records returned as sockaddr_in{}s	two DNS searches (note 1): gethostbyname2(AF_INET6) with RES_USE_INET6 off gethostbyname2(AF_INET) with RES_USE_INET6 off
		hex string	one sockaddr_in6{}	inet_pton(AF_INET6)
		dotted decimal	one sockaddr_in{}	inet_pton(AF_INET)
	AF_INET6	hostname	all AAAA records returned as sockaddr_in6{}s, *else* all A records returned as IPv4-mapped IPv6 as sockaddr_in6{}s	gethostbyname() with RES_USE_INET6 on (note 2)
		hex string	one sockaddr_in6{}	inet_pton(AF_INET6)
		dotted decimal	error: EAI_ADDRFAMILY	
	AF_INET	hostname	all A records returned as sockaddr_in{}s	gethostbyname() with RES_USE_INET6 off
		hex string	error: EAI_ADDRFAMILY	
		dotted decimal	one sockaddr_in{}	inet_pton(AF_INET)
null hostname string; passive	AF_UNSPEC	implied 0::0 implied 0.0.0.0	one sockaddr_in6{} and one sockaddr_in{}	inet_pton(AF_INET6) inet_pton(AF_INET)
	AF_INET6	implied 0::0	one sockaddr_in6{}	inet_pton(AF_INET6)
	AF_INET	implied 0.0.0.0	one sockaddr_in{}	inet_pton(AF_INET)
null hostname string; active	AF_UNSPEC	implied 0::1 implied 127.0.0.1	one sockaddr_in6{} and one sockaddr_in{}	inet_pton(AF_INET6) inet_pton(AF_INET)
	AF_INET6	implied 0::1	one sockaddr_in6{}	inet_pton(AF_INET6)
	AF_INET	implied 127.0.0.1	one sockaddr_in{}	inet_pton(AF_INET)

Figure 11.4 Summary of getaddrinfo and its actions and results.

Note 1 is that when the two DNS searches are performed, either can fail (i.e., find no records of the desired type for the hostname) but at least one must succeed. But if both searches succeed (the hostname has both AAAA and A records), then both types of socket address structures are returned.

Note 2 is that this DNS search must succeed, or an error is returned. But since the RES_USE_INET6 option is enabled, gethostbyname first looks for the AAAA records, and if nothing is found, then looks for A records (Figure 9.6).

The setting and clearing of the resolver's RES_USE_INET6 option with the scenarios in notes 1 and 2 is to force the desired DNS search, given the rules in Figure 9.5.

We note that Figure 11.4 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 11.2.

Posix.1g says nothing specific about `getaddrinfo` and Unix domain sockets (which we describe in detail in Chapter 14). Nevertheless, adding support for Unix domain sockets to our implementation of `getaddrinfo` and testing applications with these protocols is a good test for protocol independence.

Our implementation makes the following assumption: if the hostname argument for `getaddrinfo` is either `/local` or `/unix` and the service name argument is an absolute pathname (one that begins with a slash), Unix domain socket structures are returned. Valid DNS hostnames cannot contain a slash and no existing IANA service names begin with a slash (Exercise 11.5). The socket address structures returned contain this absolute pathname, ready for a call to either `bind` or `connect`. If the caller specifies the `AI_CANONNAME` flag, the host's name (Section 9.7) is returned as the canonical name.

11.6 `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 the `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 11.1.

```
solaris % testga  -f inet  -c  -h bsdi  -s domain

socket(AF_INET, SOCK_STREAM, 0), ai_canonname = bsdi.kohala.com
        address: 206.62.226.35.53

socket(AF_INET, SOCK_DGRAM, 0)
        address: 206.62.226.35.53

socket(AF_INET, SOCK_STREAM, 0)
        address: 206.62.226.66.53

socket(AF_INET, SOCK_DGRAM, 0)
        address: 206.62.226.66.53
```

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

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

```
solaris % testga  -f inet  -t stream  -h gateway.tuc.noao.edu  -s daytime

socket(AF_INET, SOCK_STREAM, 0)
        address: 140.252.101.4.13

socket(AF_INET, SOCK_STREAM, 0)
        address: 140.252.102.1.13
```

```
socket(AF_INET, SOCK_STREAM, 0)
        address: 140.252.104.1.13

socket(AF_INET, SOCK_STREAM, 0)
        address: 140.252.3.6.13

socket(AF_INET, SOCK_STREAM, 0)
        address: 140.252.4.100.13

socket(AF_INET, SOCK_STREAM, 0)
        address: 140.252.1.4.13
```

Next we specify our host `alpha`, which has both a AAAA record and an A record, without specifying the address family, and a service name of `ftp`, which is provided by TCP only.

```
solaris % testga  -h alpha  -s ftp

socket(AF_INET6, SOCK_STREAM, 0)
        address: 5f1b:df00:ce3e:e200:20:800:2b37:6426.21

socket(AF_INET, SOCK_STREAM, 0)
        address: 206.62.226.42.21
```

Since we did not 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 IPv6 and one for IPv4.

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

```
solaris % testga  -p  -s 8888

socket(AF_INET6, SOCK_STREAM, 0)
        address: ::.8888

socket(AF_INET6, SOCK_DGRAM, 0)
        address: ::.8888

socket(AF_INET, SOCK_STREAM, 0)
        address: 0.0.0.0.8888

socket(AF_INET, SOCK_DGRAM, 0)
        address: 0.0.0.0.8888
```

Four 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. Since we specified a port number without a socket type, `getaddrinfo` returns one structure for each address specifying TCP and another structure for each address specifying UDP. The two IPv6 structures are returned before the two IPv4 structures, because we saw in Chapter 10 that an IPv6 client or server on a dual-stack host can communicate with either IPv6 or IPv4 peers.

As an example of Unix domain sockets, we specify `/local` as the hostname and `/tmp/test.1` as the service name.

```
solaris % testga  -c  -p  -h /local  -s /tmp/test.1

socket(AF_LOCAL, SOCK_STREAM, 0), ai_canonname = solaris.kohala.com
        address: /tmp/test.1

socket(AF_LOCAL, SOCK_DGRAM, 0)
        address: /tmp/test.1
```

Since we do not specify the socket type, two structures are returned: the first for a stream socket and the second for a datagram socket.

11.7 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.

```
#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 11.5 shows the source code for this function.

lib/host_serv.c

```
 1 #include    "unp.h"

 2 struct addrinfo *
 3 host_serv(const char *host, const char *serv, int family, int socktype)
 4 {
 5     int     n;
 6     struct addrinfo hints, *res;

 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_flags = AI_CANONNAME;  /* always return canonical name */
 9     hints.ai_family = family;   /* AF_UNSPEC, AF_INET, AF_INET6, etc. */
10     hints.ai_socktype = socktype;   /* 0, SOCK_STREAM, SOCK_DGRAM, etc. */

11     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
12         return (NULL);

13     return (res);               /* return pointer to first on linked list */
14 }
```

lib/host_serv.c

Figure 11.5 host_serv function.

7–13 The function initializes a hints structure, calls getaddrinfo, and returns a null pointer if an error occurs.

We call this function from Figure 15.17 when we want to use getaddrinfo to obtain the host and service information, but we want to establish the connection ourself.

11.8 `tcp_connect` Function

We 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.

```
#include "unp.h"

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

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

Figure 11.6 shows the source code.

lib/tcp_connect.c

```
 1 #include    "unp.h"

 2 int
 3 tcp_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_STREAM;

10     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11         err_quit("tcp_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("tcp_connect error for %s, %s", host, serv);

24     freeaddrinfo(ressave);

25     return (sockfd);
26 }
```

lib/tcp_connect.c

Figure 11.6 `tcp_connect` function: perform normal client steps.

Call `getaddrinfo`

7–13 `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

14–25 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 were 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 `getaddrinfo` in the following sections) terminates if either `getaddrinfo` fails, or if no call to `connect` succeeds. The only return is upon success. It would be hard to return an error code (one of the `EAI_`*xxx* constants) without adding another argument. This means that our wrapper function is trivial:

```
int
Tcp_connect(const char *host, const char *serv)
{
    return(tcp_connect(host, serv));
}
```

Nevertheless, we still call our wrapper function, instead of `tcp_connect`, to maintain consistency with the remainder of the text.

The problem with the return value is that descriptors are nonnegative but we do not know whether the `EAI_`*xxx* values are positive or negative. If these values were positive, we could return the negative of these values if `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 11.7 shows our daytime client from Figure 1.5 recoded to use `tcp_connect`.

Command-line arguments

9–10 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

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

Print server's address

12–15 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 define the constant `MAXSOCKADDR` in our `unp.h` header to be the size of the largest socket address structure. This is normally the size of a Unix domain socket address structure (Section 14.2), just over 100 bytes. We allocate room for a structure of this size and this is what `getpeername` fills in.

names/daytimetcpcli.c

```
#include    "unp.h"

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

    if (argc != 3)
        err_quit("usage: daytimetcpcli <hostname/IPaddress> <service/port#>");

    sockfd = Tcp_connect(argv[1], argv[2]);

    sa = Malloc(MAXSOCKADDR);
    len = MAXSOCKADDR;
    Getpeername(sockfd, sa, &len);
    printf("connected to %s\n", Sock_ntop_host(sa, len));

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

names/daytimetcpcli.c

Figure 11.7 Daytime client recoded to use `getaddrinfo`.

We call `malloc` for this structure, instead of allocating it as

```
char  sockaddr[MAXSOCKADDR];
```

for alignment reasons. `malloc` always returns a pointer with the strictest alignment required by the system, while a `char` array could be allocated on an odd-byte boundary, which could be a problem for the IP address or port number fields in the socket address structure. Another way to handle this potential alignment problem was shown in Figure 4.19 using a `union`.

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.14, 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.

```
solaris % daytimetcpcli bsdi daytime
connected to 206.62.226.35
Fri May 30 12:33:32 1997
```

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

```
solaris % daytimetcpcli aix daytime
connected to 5f1b:df00:ce3e:e200:20:800:5afc:2b36
Fri May 30 12:43:43 1997
```

The IPv6 address is used because the host has both a AAAA record and an A record, and as noted in Figure 11.4, 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 9.2 is our convention for the hostname with only A records.

```
solaris % daytimetcpcli aix-4 daytime
connected to 206.62.226.43
Fri May 30 12:43:48 1997
```

11.9 `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 11.8 shows the source code.

```
#include "unp.h"

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

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

Call `getaddrinfo`

8–15 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 11.4 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

16–24 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

25–26 If all the calls to `socked` and `bind` failed, 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.

27 The socket is turned into a listening socket by `listen`.

Return size of socket address structure

28–31 If the *addrlenp* argument is nonnull, 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 11.1 also.)

lib/tcp_listen.c

```
#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);
}
```

lib/tcp_listen.c

Figure 11.8 `tcp_listen` function: perform normal server steps.

Example: Daytime Server

Figure 11.9 shows our daytime server from Figure 4.11 recoded to use `tcp_listen`.

Require service name or port number as command-line argument

11-12 We require a command-line argument to specify either the service name or the port number. This makes it easier to test our server, since binding port 13 for the daytime server requires superuser privileges.

names/daytimetcpsrv1.c

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     socklen_t addrlen, len;
 8     char    buff[MAXLINE];
 9     time_t  ticks;
10     struct sockaddr *cliaddr;

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

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

14     cliaddr = Malloc(addrlen);

15     for ( ; ; ) {
16         len = addrlen;
17         connfd = Accept(listenfd, cliaddr, &len);
18         printf("connection from %s\n", Sock_ntop(cliaddr, len));

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

22         Close(connfd);
23     }
24 }
```

names/daytimetcpsrv1.c

Figure 11.9 Daytime server recoded to use `getaddrinfo`.

Create listening socket

13–14 `tcp_listen` creates the listening socket and `malloc` allocates a buffer to hold the client's address.

Server loop

15–23 `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` (described in Section 11.13) 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 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 11.9: 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 11.4) 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 9.2), 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 upon 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, then 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 11.10 shows this final version of our daytime server.

Handle command-line arguments

11-16 The only change from Figure 11.9 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.

```
solaris % daytimetcpsrv2 0.0.0.0 9999
connection from 206.62.226.36.32789
connection from 206.62.226.35.1389
```

But now we start the server with an IPv6 socket.

```
solaris % daytimetcpsrv2 0::0 9999
connection from 5f1b:df00:ce3e:e200:20:800:2003:f642.32799
connection from 5f1b:df00:ce3e:e200:20:800:2b37:6426.1026
```

names/daytimetcpsrv2.c

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

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

    if (argc == 2)
        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
    else if (argc == 3)
        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
    else
        err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");

    cliaddr = Malloc(addrlen);

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

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

        Close(connfd);
    }
}
```

names/daytimetcpsrv2.c

Figure 11.10 Protocol-independent daytime server that uses `getaddrinfo`.

```
connection from ::ffff:206.62.226.36.32792
connection from ::ffff:206.62.226.35.1390
```

The first connection is from the host `sunos5` using IPv6 and the second is from the host `alpha` using IPv6. The next two connections are from the hosts `sunos5` and `bsdi`, 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 address, as we discussed in Section 10.2.

This server, along with the client in Figure 11.7, also work with Unix domain sockets (Chapter 14) since our implementation of `getaddrinfo` in Section 11.16 supports Unix domain sockets. For example, we start the server as

```
solaris % daytimetcpsrv2 /local /tmp/rendezvous
```

where the pathname /tmp/rendezvous is an arbitrary pathname we choose for the server to bind and to which the client connects. We then start the client on the same host, specifying /local as the hostname and /tmp/rendezvous as the service name.

```
solaris % daytimetcpcli /tmp/rendezvous 0
connected to /tmp/rendezvous
Fri May 30 16:31:37 1997
```

11.10 udp_client Function

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.

```
#include "unp.h"

int udp_client(const char *hostname, const char *service,
               void **saptr, socklen_t *lenp);

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

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 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 that 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.

> saptr should be declared as struct sockaddr **. We use the void ** datatype because we define another version of this function that uses XTI in Section 31.3 and it uses this argument to contain the address of a pointer to a different type of structure. This means our calls to this function must contain the cast (void **).

Figure 11.11 shows the source code for this function.

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 11.7 to use UDP and our udp_client function. Figure 11.12 shows the protocol-independent source code.

11-16 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 1-byte datagram and then read and print the reply.

lib/udp_client.c

```
#include    "unp.h"

int
udp_client(const char *host, const char *serv, void **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);
}
```

lib/udp_client.c

Figure 11.11 `udp_client` function: create an unconnected UDP socket.

We need to send only a 0-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 0-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.

```
solaris % daytimeudpcli1 aix daytime
sending to 5f1b:df00:ce3e:e200:20:800:5afc:2b36
Sat May 31 08:13:34 1997
```

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

```
solaris % daytimeudpcli1 206.62.226.43 daytime
sending to 206.62.226.43
Sat May 31 08:14:02 1997
```

names/daytimeudpcli1.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);
}
```

names/daytimeudpcli1.c

Figure 11.12 UDP daytime client using our `udp_client` function.

11.11 udp_connect Function

Our `udp_connect` function creates a connected UDP socket.

```
#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 11.13 shows the source code.

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.

lib/udp_connect.c

```
#include    "unp.h"

int
udp_connect(const char *host, const char *serv)
{
    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_connect 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;           /* ignore this one */

        if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
            break;              /* success */

        Close(sockfd);          /* ignore this one */
    } while ( (res = res->ai_next) != NULL);

    if (res == NULL)            /* errno set from final connect() */
        err_sys("udp_connect error for %s, %s", host, serv);

    freeaddrinfo(ressave);

    return (sockfd);
}
```

lib/udp_connect.c

Figure 11.13 `udp_connect` function: create a connected UDP socket.

11.12 `udp_server` Function

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

```
#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 11.14 shows the source code.

lib/udp_server.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);
}
```

lib/udp_server.c

Figure 11.14 udp_server function: create 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 11.10 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 11.15 shows our daytime server, modified from Figure 11.10 to use UDP.

names/daytimeudpsrv2.c

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

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

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

    cliaddr = Malloc(addrlen);

    for ( ; ; ) {
        len = addrlen;
        n = Recvfrom(sockfd, buff, MAXLINE, 0, cliaddr, &len);
        printf("datagram from %s\n", Sock_ntop(cliaddr, len));

        ticks = time(NULL);
        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
        Sendto(sockfd, buff, strlen(buff), 0, cliaddr, len);
    }
}
```

names/daytimeudpsrv2.c

Figure 11.15 Protocol independent UDP daytime server.

11.13 `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.

```
#include <netdb.h>

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

Returns: 0 if OK, –1 on error

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 either `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. To help allocate arrays to hold these two strings, the constants shown in Figure 11.16 are defined by including the `<netdb.h>` header.

Constant	Description	Value
`NI_MAXHOST`	maximum size of returned host string	1025
`NI_MAXSERV`	maximum size of returned service string	32

Figure 11.16 Constants for returned string sizes from `getnameinfo`.

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 11.17 shows the five *flags* that can be specified to change the operation of `getnameinfo`.

Constant	Description
`NI_DGRAM`	datagram service
`NI_NAMEREQD`	return an error if name cannot be resolved from address
`NI_NOFQDN`	return only hostname portion of FQDN
`NI_NUMERICHOST`	return numeric string for hostname
`NI_NUMERICSERV`	return numeric string for service name

Figure 11.17 *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 exist 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 be mapped into a hostname. These servers then take this returned hostname and call `gethostbyname` and 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 were 206.62.226.42, `gethostbyaddr` would return a name of `alpha.kohala.com`. But if this flag is specified to `getnameinfo`, it returns the hostname as just `alpha`.

NI_NUMERICHOST tells getnameinfo not to call the DNS (which can take time). Instead the numeric representation of the IP address is returned, probably by calling inet_ntop. Similarly the NI_NUMERICSERV specifies that the decimal port number is to be returned, instead of looking up the service name. Servers should normally specify NI_NUMERICSERV because the client port numbers normally 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), while other combinations make no sense (e.g., NI_NAMEREQD and NI_NUMERICHOST).

getnameinfo was overlooked by Posix.1g but is specified in RFC 2133 [Gilligan et al. 1997].

11.14 Reentrant Functions

The gethostbyname function from Section 9.3 presents an interesting problem that we have not yet examined in the text: it is not *reentrant*. We will encounter this problem in general when we deal with threads in Chapter 23, 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:

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

struct hostent *
gethostbyname(const char *hostname)
{
    return(gethostbyname2(hostname, family));  /* Figure 9.6 */
}

struct hostent *
gethostbyname2(const char *hostname, int family)
{
    /* call DNS functions for A or AAAA query */

    /* fill in host structure */

    return(&host);
}

struct hostent *
gethostbyaddr(const char *addr, size_t len, int family)
{
    /* call DNS functions for PTR query in in-addr.arpa domain */

    /* 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 discussed in Exercise 9.1. (Recall from Figure 9.6 that gethostbyname2 is new with the IPv6 support in BIND 4.9.4. We will ignore the fact that gethostbyname2 is involved when we call gethostbyname, as that doesn't affect this discussion.)

The reentrancy 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 a second), the main flow of control of the process is temporarily stopped and the signal handling function is called. Consider the following.

```
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 and Chapter 9, along with the inet_*XXX* functions from Chapter 4, we note the following:

- Historically, gethostbyname, gethostbyname2, gethostbyaddr, getservbyname, and getservbyport are not reentrant because all return a pointer to a static structure.

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

Alternately, some implementations that support threads (Digital Unix 4.0 and HP-UX 10.30) provide reentrant versions of these functions using thread-specific data.

- `inet_pton` and `inet_ntop` are always reentrant.
- Historically `inet_ntoa` is not reentrant but some implementations that support threads provide a reentrant version that uses thread-specific data.
- `getaddrinfo` is reentrant only if it calls reentrant functions itself; that is, if it calls reentrant 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 reentrant.
- `getnameinfo` is reentrant only if it calls reentrant functions itself; that is, if it calls reentrant 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 reentrancy.

A similar problem occurs with the variable `errno`. Historically there has been a single copy of this integer variable per process. If the process makes a system call that returns an error, an integer error code is stored into 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 the program tests the return value of the `close` function and then prints the value of `errno` if an error occurred, as in the following:

```
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 the `close` and the `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 nonreentrant 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:

```
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 reentrancy problem because many versions of the standard I/O library are nonreentrant: standard I/O functions should not be called from signal handlers.

We revisit this problem of reentrancy in Chapter 23 and we will see how threads handle the problem of the errno variable. The next section describes some reentrant versions of the hostname functions.

11.15 gethostbyname_r and gethostbyaddr_r Functions

There are two ways to make a nonreentrant function such as gethostbyname reentrant.

1. Instead of filling in and returning a static structure, the caller allocates the structure and the reentrant function fills in the caller's structure. This is the technique used in going from the nonreentrant gethostbyname to the reentrant 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 9.2). 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 reentrancy problem that we described with errno.)

   This technique is also used by getnameinfo and inet_ntop.

2. The reentrant 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 now discuss the Solaris 2.x reentrant functions for name-to-address and address-to-name resolution.

```
#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: nonnull pointer if OK, NULL on error
```

Four additional arguments are required for each function. *result* is a `hostent` structure allocated by the caller which 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 manual 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, 35 address pointers, and internally use an 8192-byte buffer to hold the 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, and not through the global `h_errno`.

Unfortunately this problem of reentrancy is even worse than it appears. First, there is no standard regarding reentrancy and `gethostbyname` and `gethostbyaddr`. Posix.1g specifies both functions but says nothing about thread safety. 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. But 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 reentrant, by using thread-specific data (Section 23.5). An interesting history of the development of the Solaris 2.x _r functions is in [Maslen 1997].

Lastly, while a reentrant version of gethostbyname may provide safety from different threads calling it at the same time, this says nothing about the reentrancy of the underlying resolver functions. As of this writing, the resolver functions in BIND are not reentrant.

11.16 Implementation of getaddrinfo and getnameinfo Functions

We now look at an implementation of getaddrinfo and getnameinfo. Developing an implementation of the former will let us look at how it operates in more detail. Our implementation also supports Unix domain sockets, as we mentioned in Section 11.5.

Note: All of the appropriate portions of the code that we look at in this section that are dependent on IPv4, IPv6, or Unix domain support, are bounded by an #ifdef and #endif of the appropriate constant: IPV4, IPV6, or UNIXDOMAIN. This allows the code to be compiled on a system that supports any combination of these three protocols. But we have removed all these preprocessor statements from the code that we show because they add nothing to our discussion and make the code harder to follow.

We also note that we do not cover Unix domain sockets in detail until Chapter 14.

Figure 11.18 shows the functions that are called by getaddrinfo. All begin with the ga_ prefix.

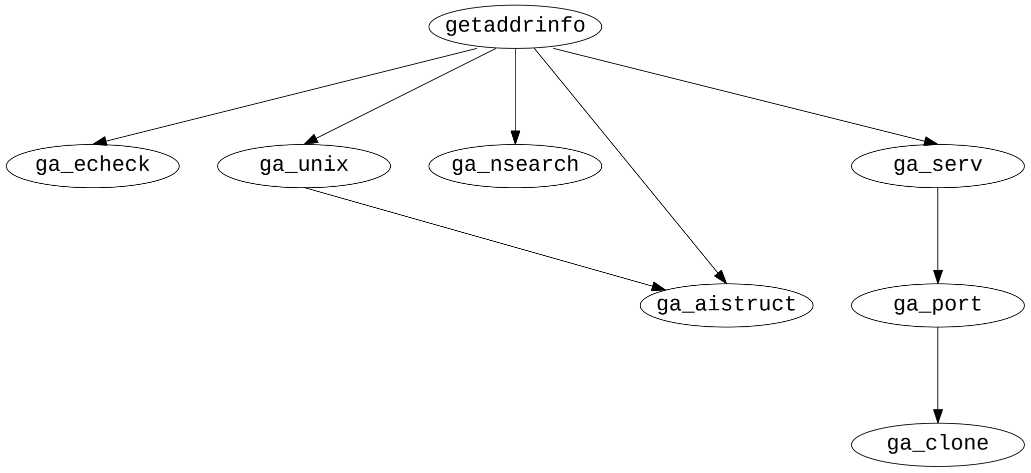

Figure 11.18 Functions called by our implementation of getaddrinfo.

The first file is our gai_hdr.h header, shown in Figure 11.19, which is included by all our source files.

We include our normal unp.h header and one additional header. We will see the use of our AI_CLONE flag and our search structure shortly. The remainder of the header defines the function prototypes for the various functions we show in this section.

libgai/gai_hdr.c

```
#include    "unp.h"
#include    <ctype.h>           /* isxdigit(), etc. */

        /* following internal flag cannot overlap with other AI_xxx flags */
#define AI_CLONE         4      /* clone this entry for other socket types */

struct search {
    const char *host;           /* hostname or address string */
    int     family;             /* AF_xxx */
};

            /* function prototypes for our own internal functions */
int     ga_aistruct(struct addrinfo ***, const struct addrinfo *,
                    const void *, int);
struct addrinfo *ga_clone(struct addrinfo *);
int     ga_echeck(const char *, const char *, int, int, int, int);
int     ga_nsearch(const char *, const struct addrinfo *, struct search *);
int     ga_port(struct addrinfo *, int, int);
int     ga_serv(struct addrinfo *, const struct addrinfo *, const char *);
int     ga_unix(const char *, struct addrinfo *, struct addrinfo **);

int     gn_ipv46(char *, size_t, char *, size_t, void *, size_t,
                 int, int, int);
```

libgai/gai_hdr.c

Figure 11.19 `gai_hdr.h` header.

Figure 11.20 shows the first part of the `getaddrinfo` function.

Define `error` macro

13–17 At more than a dozen points throughout this function if we encounter an error, we want to free all the memory that we have allocated and return the appropriate return code. To simplify the code we define this macro that stores the return code in the variable `error` and branches to the label `bad` at the end of the function (Figure 11.26).

Initialize automatic variables

18–20 Some automatic variables are initialized. We describe the `aihead` and `aipnext` pointers in Figure 11.34.

Copy caller's `hints` structure

21–25 If the caller provides a *hints* structure, we copy it into our own local variable, so we can modify it later. Otherwise we start with a structure that is all zero, other than `ai_family`, which is initialized to `AF_UNSPEC`. The latter is normally defined to be 0, but this is not required by Posix.1g.

Check arguments

26–29 We call our `ga_echeck` function, shown in Figure 11.39, to validate some of the arguments.

Check for Unix domain pathname

30–34 If the hostname is either `/local` or `/unix` and the service name begins with a slash, we process this argument as a Unix domain pathname. Our function `ga_unix` (Figure 11.33) completely processes the pathname.

libgai/getaddrinfo.c

```
#include    "gai_hdr.h"
#include    <arpa/nameser.h>    /* needed for <resolv.h> */
#include    <resolv.h>          /* res_init, _res */

int
getaddrinfo(const char *hostname, const char *servname,
            const struct addrinfo *hintsp, struct addrinfo **result)
{
    int     rc, error, nsearch;
    char  **ap, *canon;
    struct hostent *hptr;
    struct search search[3], *sptr;
    struct addrinfo hints, *aihead, **aipnext;

    /*
     * If we encounter an error we want to free() any dynamic memory
     * that we've allocated.  This is our hack to simplify the code.
     */
#define error(e) { error = (e); goto bad; }

    aihead = NULL;              /* initialize automatic variables */
    aipnext = &aihead;
    canon = NULL;

    if (hintsp == NULL) {
        bzero(&hints, sizeof(hints));
        hints.ai_family = AF_UNSPEC;
    } else
        hints = *hintsp;        /* struct copy */

        /* first some basic error checking */
    if ( (rc = ga_echeck(hostname, servname, hints.ai_flags, hints.ai_family,
                        hints.ai_socktype, hints.ai_protocol)) != 0)
        error(rc);

        /* special case Unix domain first */
    if (hostname != NULL &&
     (strcmp(hostname, "/local") == 0 || strcmp(hostname, "/unix") == 0) &&
        (servname != NULL && servname[0] == '/'))
        return (ga_unix(servname, &hints, result));
```

libgai/getaddrinfo.c

Figure 11.20 getaddrinfo function: first part, initialization.

The remainder of our getaddrinfo function (which continues in Figure 11.24) deals with IPv4 and IPv6 sockets. Our function ga_nsearch, the first part of which is shown in Figure 11.21, calculates the number of times that we look up a hostname. If the caller specifies an address family of AF_INET or AF_INET6, then we look up the hostname only one time. But if the address family is unspecified, AF_UNSPEC, then we do two lookups: once for an IPv6 hostname, and again for an IPv4 hostname. We show the function in three parts:

- no hostname and AI_PASSIVE specified,
- no hostname and AI_PASSIVE not specified (i.e., active), and
- hostname specified.

These three parts correspond to the three major portions of Figure 11.4.

libgai/ga_nsearch.c

```
 6 int
 7 ga_nsearch(const char *hostname, const struct addrinfo *hintsp,
 8            struct search *search)
 9 {
10     int     nsearch = 0;

11     if (hostname == NULL || hostname[0] == '\0') {
12         if (hintsp->ai_flags & AI_PASSIVE) {
13                 /* no hostname and AI_PASSIVE: implies wildcard bind */
14             switch (hintsp->ai_family) {
15             case AF_INET:
16                 search[nsearch].host = "0.0.0.0";
17                 search[nsearch].family = AF_INET;
18                 nsearch++;
19                 break;
20             case AF_INET6:
21                 search[nsearch].host = "0::0";
22                 search[nsearch].family = AF_INET6;
23                 nsearch++;
24                 break;
25             case AF_UNSPEC:
26                 search[nsearch].host = "0::0";  /* IPv6 first, then IPv4 */
27                 search[nsearch].family = AF_INET6;
28                 nsearch++;
29                 search[nsearch].host = "0.0.0.0";
30                 search[nsearch].family = AF_INET;
31                 nsearch++;
32                 break;
33             }
```

libgai/ga_nsearch.c

Figure 11.21 `ga_nsearch` function: no hostname and passive.

No hostname and passive socket

11-33 If the caller does not specify a hostname and specifies `AI_PASSIVE`, we return information to create one or more passive sockets that will bind the wildcard address. A `switch` is made based on the address family: an IPv4 socket needs to bind 0.0.0.0 (`INADDR_ANY`), and an IPv6 socket needs to bind 0::0 (`IN6ADDR_ANY_INIT`). If the family is `AF_UNSPEC`, we must return information to create two sockets: the first one for IPv6 and the second for IPv4. The reason for the ordering of IPv6 first, and then IPv4 is because an IPv6 socket on a dual-stack host can handle both IPv6 and IPv4 clients. In this scenario, if the caller creates only one socket from the returned list of `addrinfo` structures, it should be the IPv6 socket.

This function creates an array of `search` structures (Figure 11.19) with each entry specifying the hostname to look up and the address family. The pointer to the caller's array of `search` structures is the last argument to this function. The return value is the number of these structures that are created, and this will always be one or two.

The next part of this function, shown in Figure 11.22, handles the case of no hostname and AI_PASSIVE not set. This implies that the caller wants to create an active socket to the local host.

libgai/ga_nsearch.c

```
34         } else {
35                 /* no host and not AI_PASSIVE: connect to local host */
36             switch (hintsp->ai_family) {
37             case AF_INET:
38                 search[nsearch].host = "localhost";     /* 127.0.0.1 */
39                 search[nsearch].family = AF_INET;
40                 nsearch++;
41                 break;
42             case AF_INET6:
43                 search[nsearch].host = "0::1";
44                 search[nsearch].family = AF_INET6;
45                 nsearch++;
46                 break;
47             case AF_UNSPEC:
48                 search[nsearch].host = "0::1";  /* IPv6 first, then IPv4 */
49                 search[nsearch].family = AF_INET6;
50                 nsearch++;
51                 search[nsearch].host = "localhost";
52                 search[nsearch].family = AF_INET;
53                 nsearch++;
54                 break;
55             }
56         }
```

libgai/ga_nsearch.c

Figure 11.22 ga_nsearch function: no hostname and not passive.

34–56 For IPv4 we assume the hostname localhost will return the loopback address, normally 127.0.0.1. There is no common hostname for the local host with IPv6, so we return the loopback address of 0::1. As with the passive case, if no address family is specified we return two structures: first one for IPv6 and then one for IPv4.

Figure 11.23 shows the final part of this function, the else clause of the original if statement. This code is executed when a hostname is specified.

57–82 The AI_PASSIVE flag does not matter in this scenario; the hostname needs to be looked up. If the caller creates a passive socket, then the resulting socket address structure will be used in a call to bind, but if the caller creates an active socket, the socket address structure will be used in a call to connect. We create one or two search structures: one if the address family is specified and two if it is not specified. As with the previous two scenarios, if two structures are returned, the first is for IPv6 and the second for IPv4.

We now return to our getaddrinfo function, in Figure 11.24, which starts with a call to ga_nsearch.

libgai/ga_nsearch.c

```
    } else {                    /* host is specified */
        switch (hintsp->ai_family) {
        case AF_INET:
            search[nsearch].host = hostname;
            search[nsearch].family = AF_INET;
            nsearch++;
            break;
        case AF_INET6:
            search[nsearch].host = hostname;
            search[nsearch].family = AF_INET6;
            nsearch++;
            break;
        case AF_UNSPEC:
            search[nsearch].host = hostname;
            search[nsearch].family = AF_INET6;  /* IPv6 first */
            nsearch++;
            search[nsearch].host = hostname;
            search[nsearch].family = AF_INET;   /* then IPv4 */
            nsearch++;
            break;
        }
    }
    if (nsearch < 1 || nsearch > 2)
        err_quit("nsearch = %d", nsearch);
    return (nsearch);
}
```

libgai/ga_nsearch.c

Figure 11.23 `ga_nsearch` function: hostname specified.

Call `ga_nsearch`

36 We call our `ga_nsearch` function, filling in our `search` array, and returning the number of structures in the array: one or two.

Loop through all the `search` structures

37 We loop through each `search` structure that was created by `ga_nsearch`.

Check for IPv4 dotted-decimal string

39–44 If the first character of the hostname is a digit, we check whether or not the hostname is really a dotted-decimal string. We call `inet_pton` to do this check and conversion. If it succeeds but the caller specifies an address family other than `AF_INET`, this is an error.

45–46 We check that the `family` of the `search` structure is also `AF_INET`, but a mismatch here only causes this `search` structure to be ignored. This scenario can happen, for example, if the caller specifies a hostname of 192.3.4.5 but no address family. `ga_nsearch` creates two `search` structures: one for IPv6 and one for IPv4. The first time through the `for` loop the call to `inet_pton` succeeds, but since the `family` of the `search` structure is `AF_INET6`, we want to ignore this structure, and not generate an error.

libgai/getaddrinfo.c

```
35            /* remainder of function for IPv4/IPv6 */
36        nsearch = ga_nsearch(hostname, &hints, &search[0]);
37        for (sptr = &search[0]; sptr < &search[nsearch]; sptr++) {
38                /* check for an IPv4 dotted-decimal string */
39            if (isdigit(sptr->host[0])) {
40                struct in_addr inaddr;

41                if (inet_pton(AF_INET, sptr->host, &inaddr) == 1) {
42                    if (hints.ai_family != AF_UNSPEC &&
43                        hints.ai_family != AF_INET)
44                        error(EAI_ADDRFAMILY);
45                    if (sptr->family != AF_INET)
46                        continue;   /* ignore */
47                    rc = ga_aistruct(&aipnext, &hints, &inaddr, AF_INET);
48                    if (rc != 0)
49                        error(rc);
50                    continue;
51                }
52            }
53                /* check for an IPv6 hex string */
54            if ((isxdigit(sptr->host[0]) || sptr->host[0] == ':') &&
55                (strchr(sptr->host, ':') != NULL)) {
56                struct in6_addr in6addr;

57                if (inet_pton(AF_INET6, sptr->host, &in6addr) == 1) {
58                    if (hints.ai_family != AF_UNSPEC &&
59                        hints.ai_family != AF_INET6)
60                        error(EAI_ADDRFAMILY);
61                    if (sptr->family != AF_INET6)
62                        continue;   /* ignore */
63                    rc = ga_aistruct(&aipnext, &hints, &in6addr, AF_INET6);
64                    if (rc != 0)
65                        error(rc);
66                    continue;
67                }
68            }
```

libgai/getaddrinfo.c

Figure 11.24 getaddrinfo function: check for IPv4 or IPv6 address string.

Create addrinfo structure

47–52 Our function ga_aistruct creates an addrinfo structure and adds it to the linked list that is being built (the aipnext pointer).

Check for IPv6 address string

53–60 If the first character of the hostname is either a hexadecimal digit or a colon and the string contains a colon, we check whether the hostname is an IPv6 address string by calling inet_pton. If it succeeds but the caller specifies an address family other than AF_INET6, this is an error.

61–62 We check that the family of the search structure is also AF_INET6, but a mismatch here only causes this search structure to be ignored.

Create `addrinfo` structure

63-68 Our function `ga_aistruct` creates an `addrinfo` structure and adds it to the linked list that is being built.

The first two tests in the loop (Figure 11.24) handle an IPv4 dotted-decimal string or an IPv6 address string. The remainder of the loop, shown in Figure 11.25, looks up the hostname by calling either `gethostbyname` or `gethostbyname2`.

Initialize resolver first time

70-71 We call the resolver's `res_init` function if it has not been called before.

Call `gethostbyname2` if two searches are being performed

72-74 If `nsearch` is 2, then we are going through the `for` loop twice: once for IPv6 and again for IPv4. If the hostname argument has an address in only one of the two families, we want to return only that address. For example, our host `solaris` in Section 9.2 has a AAAA record and an A record in the DNS. The first time around the loop we want to find the AAAA record, and the second time the A record. But if the hostname has only an A record, we do not want to process that record the first time around the loop when the `family` member of the `search` structure is `AF_INET6`. That is, since we know that we will be searching for an A record for this host, do not search for a AAAA record using `gethostbyname` and possibly return the IPv4-mapped IPv6 address corresponding to the A record. Looking at Figure 9.5 the way to search for only A records when the family is `AF_INET` and to search for only AAAA records when the family is `AF_INET6` is to call `gethostbyname2` instead of `gethostbyname`, with the `RES_USE_INET6` option off.

Call `gethostbyname` if one search is being performed

75-81 If only one search is being performed, we call `gethostbyname` with the `RES_USE_INET6` option set if the family is `AF_INET6` or the option cleared if the family is `AF_INET`. For example, if the caller specifies a hostname that has only an A record, but specifies a family of `AF_INET6`, we want to return the IPv4-mapped IPv6 address.

Handle resolver failure

82-97 If the call to the resolver failed, but `nsearch` is two, this is not an error, as one of the passes through the loop may succeed. (We check at the end of the loop that at least one `addrinfo` structure is being returned.) But if this was the only call to the resolver we return an error corresponding to the resolver's `h_errno`.

Check for address family mismatch

98-100 If the caller specifies an address family, but the family returned by the resolver differs, this is an error.

Save canonical name

101-106 If the caller specifies a hostname and the `AI_CANONNAME` flag, we save the first canonical name returned by the resolver. (Recall from Figure 11.22 that we call the resolver for the name `localhost` even of the caller does not specify a hostname.) We duplicate the string returned by the resolver and save its pointer in `canon`.

libgai/getaddrinfo.c

```
            /* remainder of for() to look up hostname */
        if ((_res.options & RES_INIT) == 0)
            res_init();         /* need this to set _res.options */

        if (nsearch == 2) {
            _res.options &= ˜RES_USE_INET6;
            hptr = gethostbyname2(sptr->host, sptr->family);
        } else {
            if (sptr->family == AF_INET6)
                _res.options |= RES_USE_INET6;
            else
                _res.options &= ˜RES_USE_INET6;
            hptr = gethostbyname(sptr->host);
        }
        if (hptr == NULL) {
            if (nsearch == 2)
                continue;       /* failure OK if multiple searches */

            switch (h_errno) {
            case HOST_NOT_FOUND:
                error(EAI_NONAME);
            case TRY_AGAIN:
                error(EAI_AGAIN);
            case NO_RECOVERY:
                error(EAI_FAIL);
            case NO_DATA:
                error(EAI_NODATA);
            default:
                error(EAI_NONAME);
            }
        }
            /* check for address family mismatch if one specified */
        if (hints.ai_family != AF_UNSPEC && hints.ai_family != hptr->h_addrtype)
            error(EAI_ADDRFAMILY);

            /* save canonical name first time */
        if (hostname != NULL && hostname[0] != '\0' &&
            (hints.ai_flags & AI_CANONNAME) && canon == NULL) {
            if ( (canon = strdup(hptr->h_name)) == NULL)
                error(EAI_MEMORY);
        }
            /* create one addrinfo{} for each returned address */
        for (ap = hptr->h_addr_list; *ap != NULL; ap++) {
            rc = ga_aistruct(&aipnext, &hints, *ap, hptr->h_addrtype);
            if (rc != 0)
                error(rc);
        }
    }
    if (aihead == NULL)
        error(EAI_NONAME);      /* nothing found */
```

libgai/getaddrinfo.c

Figure 11.25 getaddrinfo function: lookup hostname.

Create one `addrinfo` structure per address

107-112 For each address returned by the resolver in the `h_addr_list` array, we call our `ga_aistruct` function to create an `addrinfo` structure and append it to the linked list of structures being created.

Check for no matches

114-115 If the head of the linked list of `addrinfo` structures is still a null pointer, all iterations through the `for` loop failed.

Figure 11.26 shows the final part of the `getaddrinfo` function.

libgai/getaddrinfo.c

```
116         /* return canonical name */
117     if (hostname != NULL && hostname[0] != '\0' &&
118         hints.ai_flags & AI_CANONNAME) {
119         if (canon != NULL)
120             aihead->ai_canonname = canon;   /* strdup'ed earlier */
121         else {
122             if ( (aihead->ai_canonname = strdup(search[0].host)) == NULL)
123                 error(EAI_MEMORY);
124         }
125     }
126         /* now process the service name */
127     if (servname != NULL && servname[0] != '\0') {
128         if ( (rc = ga_serv(aihead, &hints, servname)) != 0)
129             error(rc);
130     }
131     *result = aihead;           /* pointer to first structure in linked list */
132     return (0);

133   bad:
134     freeaddrinfo(aihead);       /* free any alloc'ed memory */
135     return (error);
136 }
```

libgai/getaddrinfo.c

Figure 11.26 `getaddrinfo` function: process service name.

Return canonical name

116-125 If the caller specifies a hostname and the `AI_CANONNAME` flag, and if we saved a copy to the canonical name in our `canon` pointer, that pointer is returned in the `ai_canonname` member of the first `addrinfo` structure. If no canonical name was found by the resolver (perhaps the hostname was an address string), then a copy of the hostname argument is returned instead.

Process service name

126-130 If the caller specifies a service name, it is now processed by calling our `ga_serv` function.

Return pointer to linked list

131-132 The pointer to the head of the linked list of `addrinfo` structures that have been created is returned, along with a function return value of 0.

Error return

133–135 If an error was encountered, freeaddrinfo is called to free all the memory that was allocated, and the return value is the EAI_*xxx* value.

Our ga_serv function, which was called from Figure 11.26 to process the service name argument, is shown in Figure 11.27.

Check for port number string

12–27 If the first character of the service name is a digit, we assume the service name is a port number and call atoi to convert it to binary. If the caller specifies a socket type (SOCK_STREAM or SOCK_DGRAM), then our ga_port function is called once for that socket type. But if no socket type is specified, our ga_port function is called twice, once for TCP and once for UDP. (Recall Figure 11.2.)

Try getservbyname for TCP

28–36 If no socket type is specified, or a TCP socket is specified, getservbyname is called with a second argument of "tcp". If this succeeds, our ga_port function is called. If this call fails, that is OK, as the service name could be valid for UDP. We keep a counter of the number of times that ga_port returns success and return an error only if this is 0 at the end of the function.

Try getservbyname for UDP

37–44 If no socket type is specified, or a UDP socket is specified, we call getservbyname with a second argument of "udp". If this succeeds, we call our ga_port function.

Check for error

45–51 If our nfound counter is nonzero, we had success. Otherwise an error is returned.

Our ga_port function, which we show in Figure 11.28, was called from Figure 11.27 each time a port number was found.

Loop through all addrinfo structures

33 We loop through all the addrinfo structures that were created by the calls to ga_aistruct in Figures 11.24 and 11.25. The AI_CLONE flag is always set by ga_aistruct when no socket type is specified by the caller. That is an indication that this addrinfo structure might need to be cloned for both TCP and UDP.

Check AI_CLONE flag

34–42 If the AI_CLONE flag is set and if the socket type is nonzero, another addrinfo structure is cloned from this one by our ga_clone function. We show an example of this shortly.

Set port number in socket address structure

44–47 The port number in the socket address structure is set and our counter nfound is incremented.

libgai/ga_serv.c

```c
int
ga_serv(struct addrinfo *aihead, const struct addrinfo *hintsp,
        const char *serv)
{
    int     port, rc, nfound;
    struct servent *sptr;

    nfound = 0;
    if (isdigit(serv[0])) {     /* check for port number string first */
        port = htons(atoi(serv));
        if (hintsp->ai_socktype) {
                /* caller specifies socket type */
            if ( (rc = ga_port(aihead, port, hintsp->ai_socktype)) < 0)
                return (EAI_MEMORY);
            nfound += rc;
        } else {
                /* caller does not specify socket type */
            if ( (rc = ga_port(aihead, port, SOCK_STREAM)) < 0)
                return (EAI_MEMORY);
            nfound += rc;
            if ( (rc = ga_port(aihead, port, SOCK_DGRAM)) < 0)
                return (EAI_MEMORY);
            nfound += rc;
        }
    } else {
            /* try service name, TCP then UDP */
        if (hintsp->ai_socktype == 0 || hintsp->ai_socktype == SOCK_STREAM) {
            if ( (sptr = getservbyname(serv, "tcp")) != NULL) {
                if ( (rc = ga_port(aihead, sptr->s_port, SOCK_STREAM)) < 0)
                    return (EAI_MEMORY);
                nfound += rc;
            }
        }
        if (hintsp->ai_socktype == 0 || hintsp->ai_socktype == SOCK_DGRAM) {
            if ( (sptr = getservbyname(serv, "udp")) != NULL) {
                if ( (rc = ga_port(aihead, sptr->s_port, SOCK_DGRAM)) < 0)
                    return (EAI_MEMORY);
                nfound += rc;
            }
        }
    }

    if (nfound == 0) {
        if (hintsp->ai_socktype == 0)
            return (EAI_NONAME);    /* all calls to getservbyname() failed */
        else
            return (EAI_SERVICE);   /* service not supported for socket type */
    }
    return (0);
}
```

libgai/ga_serv.c

Figure 11.27 `ga_serv` function.

libgai/ga_port.c

```
int
ga_port(struct addrinfo *aihead, int port, int socktype)
        /* port must be in network byte order */
{
    int     nfound = 0;
    struct addrinfo *ai;

    for (ai = aihead; ai != NULL; ai = ai->ai_next) {
        if (ai->ai_flags & AI_CLONE) {
            if (ai->ai_socktype != 0) {
                if ( (ai = ga_clone(ai)) == NULL)
                    return (-1);    /* memory allocation error */
                /* ai points to newly cloned entry, which is what we want */
            }
        } else if (ai->ai_socktype != socktype)
            continue;           /* ignore if mismatch on socket type */

        ai->ai_socktype = socktype;

        switch (ai->ai_family) {
        case AF_INET:
            ((struct sockaddr_in *) ai->ai_addr)->sin_port = port;
            nfound++;
            break;
        case AF_INET6:
            ((struct sockaddr_in6 *) ai->ai_addr)->sin6_port = port;
            nfound++;
            break;
        }
    }
    return (nfound);
}
```

libgai/ga_port.c

Figure 11.28 ga_port function.

Consider an example. In Figure 11.1 we assumed a call to getaddrinfo for a host with two IP addresses, a service name of domain (port 53 for both TCP and UDP), and no specification of the socket type. The loop in our getaddrinfo function (Figure 11.25) creates two addrinfo structures, one for each IP address returned by gethostbyname. The AI_CLONE flag is also set in each structure, because no socket type is specified. We show the resulting linked list in Figure 11.29.

ga_serv is called from Figure 11.26. Since the domain service name is valid for both TCP and UDP, getservbyname is called two times, and ga_port is called two times: first with a final argument of SOCK_STREAM and again with a final argument of SOCK_DGRAM. The first time ga_port is called it starts with the linked list shown in Figure 11.29. In Figure 11.28 the AI_CLONE flag is set for both structures, but the socket type is 0. Therefore all that happens to each addrinfo structure the first time ga_port is called is to set the ai_socktype member to SOCK_STREAM and the port number in the socket address structure to 53. The AI_CLONE flag remains set. This gives us the linked list shown in Figure 11.30.

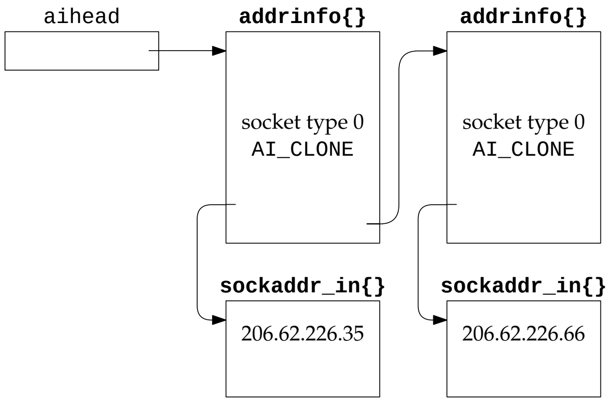

Figure 11.29 `addrinfo` structures when `ga_port` is called first time.

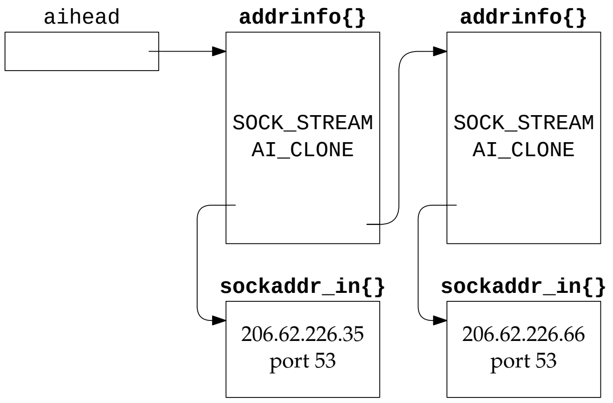

Figure 11.30 `addrinfo` structures after first call to `ga_port`.

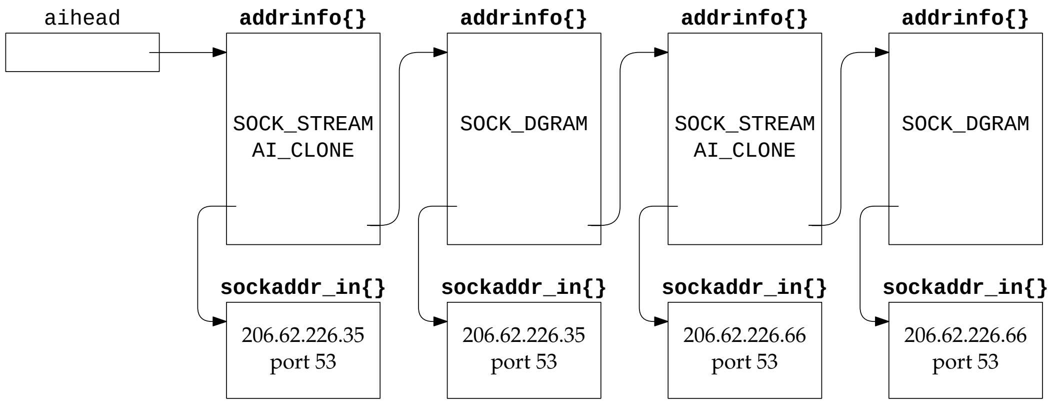

Figure 11.31 `addrinfo` structures after second call to `ga_port`.

But the second time ga_port is called (with a final argument of SOCK_DGRAM), since the AI_CLONE flag is set and the socket type is not 0, ga_clone is called for each addrinfo structure. The ai_socktype member in each of the newly cloned structures is set to SOCK_DGRAM and we end up with the linked list shown in Figure 11.31. In this figure the second addrinfo structure and its socket address structure are cloned from the first set of structures, and the fourth addrinfo structure and its socket address structure are cloned from the third set of structures.

Figure 11.32 shows our ga_clone function, which was called from Figure 11.28 to clone a new addrinfo structure and its socket address structure from an existing set of structures.

libgai/ga_clone.c

```
 5 struct addrinfo *
 6 ga_clone(struct addrinfo *ai)
 7 {
 8     struct addrinfo *new;

 9     if ( (new = calloc(1, sizeof(struct addrinfo))) == NULL)
10                 return (NULL);

11     new->ai_next = ai->ai_next;
12     ai->ai_next = new;

13     new->ai_flags = 0;          /* make sure AI_CLONE is off */
14     new->ai_family = ai->ai_family;
15     new->ai_socktype = ai->ai_socktype;
16     new->ai_protocol = ai->ai_protocol;
17     new->ai_canonname = NULL;
18     new->ai_addrlen = ai->ai_addrlen;
19     if ( (new->ai_addr = malloc(ai->ai_addrlen)) == NULL)
20         return (NULL);
21     memcpy(new->ai_addr, ai->ai_addr, ai->ai_addrlen);

22     return (new);
23 }
```

libgai/ga_clone.c

Figure 11.32 ga_clone function.

Allocate addrinfo structure and insert into linked list

9–12 A new addrinfo structure is allocated and its ai_next pointer is set to the ai_next pointer of the entry being cloned (i.e., what will be the previous entry on the list). The next pointer of the entry being cloned becomes the new structure just allocated.

Initialize from cloned entry

13–22 All the fields in the new addrinfo structure are copied from the entry being cloned with the exception of ai_flags, which is set to 0, and ai_canonname, which is set to a null pointer. A pointer to the newly created structure is the return value of the function.

Our `ga_unix` function, which we shown in Figure 11.33, was called from Figure 11.20 to completely process a Unix domain pathname.

libgai/ga_unix.c

```
 3 int
 4 ga_unix(const char *path, struct addrinfo *hintsp, struct addrinfo **result)
 5 {
 6     int     rc;
 7     struct addrinfo *aihead, **aipnext;

 8     aihead = NULL;
 9     aipnext = &aihead;

10     if (hintsp->ai_family != AF_UNSPEC && hintsp->ai_family != AF_LOCAL)
11         return (EAI_ADDRFAMILY);

12     if (hintsp->ai_socktype == 0) {
13             /* no socket type specified: return stream then dgram */
14         hintsp->ai_socktype = SOCK_STREAM;
15         if ( (rc = ga_aistruct(&aipnext, hintsp, path, AF_LOCAL)) != 0)
16             return (rc);
17         hintsp->ai_socktype = SOCK_DGRAM;
18     }
19     if ( (rc = ga_aistruct(&aipnext, hintsp, path, AF_LOCAL)) != 0)
20         return (rc);

21     if (hintsp->ai_flags & AI_CANONNAME) {
22         struct utsname myname;

23         if (uname(&myname) < 0)
24             return (EAI_SYSTEM);
25         if ( (aihead->ai_canonname = strdup(myname.nodename)) == NULL)
26             return (EAI_MEMORY);
27     }
28     *result = aihead;           /* pointer to first structure in linked list */
29     return (0);
30 }
```

libgai/ga_unix.c

Figure 11.33 `ga_unix` function.

`ga_aistruct` creates structures

10-20 If a socket type is not specified, we call our `ga_aistruct` function twice to create two `addrinfo` structures: one with a socket type of `SOCK_STREAM` and another with a socket type of `SOCK_DGRAM`. But if the caller specifies a nonzero socket type, our `ga_aistruct` function is called only once, creating one `addrinfo` structure with that socket type.

Return canonical name

21-27 If the `AI_CANONNAME` flag was specified by the caller, we call `uname` to obtain the system name and return the `nodename` member (Section 9.7) as the canonical name.

We explain the `aihead` and `aipnext` pointers with the `ga_aistruct` function, which we describe next.

Our `ga_aistruct` function was called from Figures 11.24 and 11.25 to create an `addrinfo` structure for an IPv4 or IPv6 address, and from Figure 11.33 to create an `addrinfo` structure for a Unix domain socket. We show the first part of the function in Figure 11.34.

libgai/ga_aistruct.c

```
 5 int
 6 ga_aistruct(struct addrinfo ***paipnext, const struct addrinfo *hintsp,
 7             const void *addr, int family)
 8 {
 9     struct addrinfo *ai;

10     if ( (ai = calloc(1, sizeof(struct addrinfo))) == NULL)
11             return (EAI_MEMORY);
12     ai->ai_next = NULL;
13     ai->ai_canonname = NULL;
14     **paipnext = ai;
15     *paipnext = &ai->ai_next;

16     if ( (ai->ai_socktype = hintsp->ai_socktype) == 0)
17         ai->ai_flags |= AI_CLONE;

18     ai->ai_protocol = hintsp->ai_protocol;
```

libgai/ga_aistruct.c

Figure 11.34 `ga_aistruct` function: first part.

Allocate `addrinfo` structure and add to linked list

10–15 An `addrinfo` structure is allocated and added to the linked list being built. Two pointers are used to build the linked list: `aihead` and `aipnext`. Both were allocated and initialized in Figure 11.20 for an IPv4 or an IPv6 socket, or in Figure 11.33 for a Unix domain socket. `aihead` is initialized to a null pointer and `aipnext` is initialized to point to `aihead`. We show this in Figure 11.35.

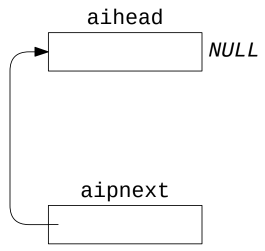

Figure 11.35 Initialization of linked list pointers.

`aihead` always points to the first `addrinfo` structure on the linked list (therefore its datatype is `struct addrinfo *`). `aipnext` normally points to the `ai_next` member of the last structure on the linked list (therefore its datatype is `struct addrinfo **`). We use the qualifier "normally" with regard to `aipnext` because upon initialization it really points to `aihead`, but after the first structure is allocated and placed onto the list, it always points to the `ai_next` member.

Returning to our `ga_aistruct` function, after a new structure is allocated the two statements

```
**paipnext = ai;
*paipnext = &ai->ai_next;
```

are executed. The first statement sets the `ai_next` pointer of the last structure on the list (or `aihead` if this new structure is the first on the list) to point to the newly allocated structure, and the second statement sets `aipnext` to point to the `ai_next` member of the newly allocated structure. The extra level of indirection is needed in both statements because the address of `aipnext` is an argument to the function (see Exercise 11.4). When the first structure is added to the list, we have the data structures shown in Figure 11.36.

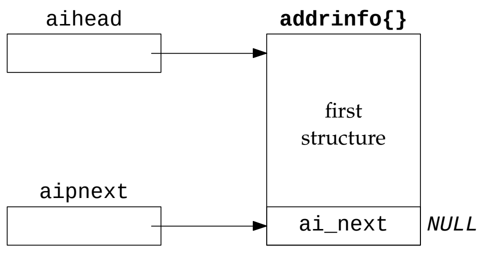

Figure 11.36 Linked list after first structure added.

When our `ga_aistruct` function is called the next time to allocate a second structure and add it to the list, we have the data structures shown in Figure 11.37.

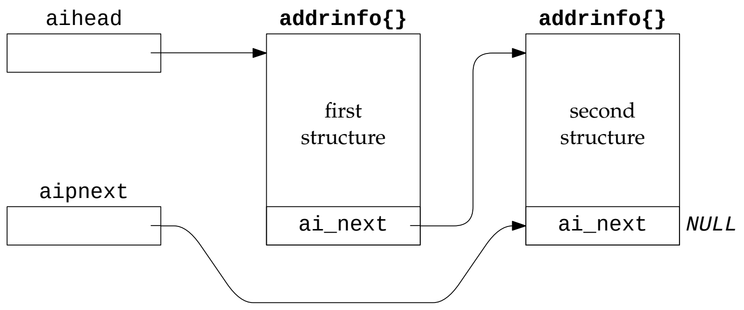

Figure 11.37 Linked list after second structure added.

Set socket type

16–17 The `ai_socktype` member is set to the socket type provided by the caller and if this is 0, the `AI_CLONE` flag is set.

Figure 11.38 shows the second part of the function: a `switch` with one `case` per address family to allocate a socket address structure and initialize it.

libgai/ga_aistruct.c

```
    switch ((ai->ai_family = family)) {
    case AF_INET:{
            struct sockaddr_in *sinptr;

                /* allocate sockaddr_in{} and fill in all but port */
            if ( (sinptr = calloc(1, sizeof(struct sockaddr_in))) == NULL)
                        return (EAI_MEMORY);
#ifdef  HAVE_SOCKADDR_SA_LEN
            sinptr->sin_len = sizeof(struct sockaddr_in);
#endif
            sinptr->sin_family = AF_INET;
            memcpy(&sinptr->sin_addr, addr, sizeof(struct in_addr));
            ai->ai_addr = (struct sockaddr *) sinptr;
            ai->ai_addrlen = sizeof(struct sockaddr_in);
            break;
        }
    case AF_INET6:{
            struct sockaddr_in6 *sin6ptr;

                /* allocate sockaddr_in6{} and fill in all but port */
            if ( (sin6ptr = calloc(1, sizeof(struct sockaddr_in6))) == NULL)
                        return (EAI_MEMORY);
#ifdef  HAVE_SOCKADDR_SA_LEN
            sin6ptr->sin6_len = sizeof(struct sockaddr_in6);
#endif
            sin6ptr->sin6_family = AF_INET6;
            memcpy(&sin6ptr->sin6_addr, addr, sizeof(struct in6_addr));
            ai->ai_addr = (struct sockaddr *) sin6ptr;
            ai->ai_addrlen = sizeof(struct sockaddr_in6);
            break;
        }
    case AF_LOCAL:{
            struct sockaddr_un *unp;

                /* allocate sockaddr_un{} and fill in */
            if (strlen(addr) >= sizeof(unp->sun_path))
                return(EAI_SERVICE);
            if ( (unp = calloc(1, sizeof(struct sockaddr_un))) == NULL)
                return(EAI_MEMORY);

            unp->sun_family = AF_LOCAL;
            strcpy(unp->sun_path, addr);
#ifdef  HAVE_SOCKADDR_SA_LEN
            unp->sun_len = SUN_LEN(unp);
#endif
            ai->ai_addr = (struct sockaddr *) unp;
            ai->ai_addrlen = sizeof(struct sockaddr_un);
            if (hintsp->ai_flags & AI_PASSIVE)
                unlink(unp->sun_path);  /* OK if this fails */
            break;
        }
    }
    return (0);
}
```

libgai/ga_aistruct.c

Figure 11.38 ga_aistruct function: second part.

Allocate IPv4 socket address structure and initialize

20-33 A `sockaddr_in` structure is allocated and the `ai_addr` pointer in the `addrinfo` structure is set to point to it. The IP address, address family, and length members of the socket address structure are all initialized. The port number is not initialized until `ga_serv` is called, which in turn calls `ga_port`.

Allocate IPv6 socket address structure and initialize

34-47 A `sockaddr_in6` structure is allocated and initialized, similar to the IPv4 case.

Allocate Unix domain socket address structure and initialize

48-65 A `sockaddr_un` structure is allocated and initialized. The address is a pathname and if the `AI_PASSIVE` flag was specified by the caller, we try to `unlink` the pathname to prevent an error return when the caller calls `bind`. But it is not an error if the `unlink` fails.

Our `ga_echeck` function, which we show in Figure 11.39, was called from Figure 11.20 to perform some initial error checking on the caller's arguments.

libgai/ga_echeck.c

```
int
ga_echeck(const char *hostname, const char *servname,
          int flags, int family, int socktype, int protocol)
{
    if (flags & ~(AI_PASSIVE | AI_CANONNAME))
        return (EAI_BADFLAGS);  /* unknown flag bits */

    if (hostname == NULL || hostname[0] == '\0') {
        if (servname == NULL || servname[0] == '\0')
            return (EAI_NONAME);    /* host or service must be specified */
    }
    switch (family) {
    case AF_UNSPEC:
        break;
    case AF_INET:
        if (socktype != 0 &&
            (socktype != SOCK_STREAM &&
             socktype != SOCK_DGRAM &&
             socktype != SOCK_RAW))
            return (EAI_SOCKTYPE);  /* invalid socket type */
        break;
    case AF_INET6:
        if (socktype != 0 &&
            (socktype != SOCK_STREAM &&
             socktype != SOCK_DGRAM &&
             socktype != SOCK_RAW))
            return (EAI_SOCKTYPE);  /* invalid socket type */
        break;
    case AF_LOCAL:
        if (socktype != 0 &&
            (socktype != SOCK_STREAM &&
             socktype != SOCK_DGRAM))
            return (EAI_SOCKTYPE);  /* invalid socket type */
        break;
38     default:
39         return (EAI_FAMILY);    /* unknown protocol family */
40     }
41     return (0);
42 }
```
libgai/ga_echeck.c

Figure 11.39 ga_echeck function.

9–14 The flags are verified and either a hostname or a service name must be specified.

15–41 Depending on the address family, only certain types of sockets are supported, and this verifies the socket type.

We do not check the caller's ai_protocol hint, if any, as few applications specify this value (which becomes the third argument to socket). Should an invalid combination be specified, such as a socket type of SOCK_DGRAM and a protocol of IPPROTO_TCP, the protocol hint is returned to the caller in Figure 11.34 and if the caller uses this value in a call to socket, an error of EPROTONOSUPPORT will be returned.

We have finished with the getaddrinfo function, and all the internal functions that it calls. Figure 11.40 shows the freeaddrinfo function, which releases all the memory in the linked list. We called this function from Figure 11.26 if an error occurred, and the user also calls it to release a linked list of structures.

libgai/freeaddrinfo.c
```
 1 #include    "gai_hdr.h"

 2 void
 3 freeaddrinfo(struct addrinfo *aihead)
 4 {
 5     struct addrinfo *ai, *ainext;

 6     for (ai = aihead; ai != NULL; ai = ainext) {
 7         if (ai->ai_addr != NULL)
 8             free(ai->ai_addr);  /* socket address structure */

 9         if (ai->ai_canonname != NULL)
10             free(ai->ai_canonname);

11         ainext = ai->ai_next;   /* can't fetch ai_next after free() */
12         free(ai);               /* the addrinfo{} itself */
13     }
14 }
```
libgai/freeaddrinfo.c

Figure 11.40 freeaddrinfo function: first part.

6–13 The linked list of addrinfo structures is traversed. If a socket address structure has been allocated, it is freed. If a canonical name string has been allocated, it is freed. Finally, the addrinfo structure itself is freed. We must be careful to save the contents of the structure's ai_next pointer before freeing the structure, as we cannot reference the structure after free returns.

Figure 11.41 shows our implementation of the getnameinfo function. It consists of a switch statement with one case per address family.

libgai/getnameinfo.c

```
 2 int
 3 getnameinfo(const struct sockaddr *sa, socklen_t salen,
 4             char *host, size_t hostlen,
 5             char *serv, size_t servlen, int flags)
 6 {
 7     switch (sa->sa_family) {
 8     case AF_INET:{
 9             struct sockaddr_in *sain = (struct sockaddr_in *) sa;

10             return (gn_ipv46(host, hostlen, serv, servlen,
11                              &sain->sin_addr, sizeof(struct in_addr),
12                              AF_INET, sain->sin_port, flags));
13         }

14     case AF_INET6:{
15             struct sockaddr_in6 *sain = (struct sockaddr_in6 *) sa;

16             return (gn_ipv46(host, hostlen, serv, servlen,
17                              &sain->sin6_addr, sizeof(struct in6_addr),
18                              AF_INET6, sain->sin6_port, flags));
19         }

20     case AF_LOCAL:{
21             struct sockaddr_un *un = (struct sockaddr_un *) sa;

22             if (hostlen > 0)
23                 snprintf(host, hostlen, "%s", "/local");
24             if (servlen > 0)
25                 snprintf(serv, servlen, "%s", un->sun_path);
26             return (0);
27         }

28     default:
29         return (1);
30     }
31 }
```

libgai/getnameinfo.c

Figure 11.41 `getnameinfo` function.

Handle IPv4 and IPv6 socket address structures

8–19 We call our `gn_ipv46` function (shown next) to handle IPv4 and IPv6 socket address structures.

Handle Unix domain socket address structures

20–27 For a Unix domain socket address structure we return `/local` as the hostname and the pathname that is bound to the socket as the service name. If no pathname is bound to the socket, then the returned service name will be a null string.

We return the hostname and service name using snprintf instead of strncpy. If we used the latter we could write

```
strncpy(host, "/local", hostlen);
```

While this guarantees that we do not overflow the caller's buffer, if hostlen is less than or equal to 6, then the caller's buffer will not be null terminated. But we are writing a library routine and we should always return a null-terminated string if that is what the caller expects. This could cause problems for the caller at a later time in the program. Therefore we should always write

```
strncpy(host, "/local", hostlen-1);
host[hostlen-1] = '\0';
```

which guarantees that we do not overwrite the caller's buffer and that the result is null terminated. We use snprintf instead of these two statements, since it will not overflow the destination and it guarantees that the destination is null terminated. An alternate design would be to define our own library function that calls strncpy and null terminates the result, but calling the existing snprintf seems simpler.

Figure 11.42 is our gn_ipv46 function, which handles IPv4 and IPv6 socket address structures for getnameinfo.

Return hostname

12–23 If the NI_NUMERICHOST flag is specified, we call inet_ntop to return the presentation format of the IP address; otherwise gethostbyaddr searches for the hostname corresponding to the IP address. If gethostbyaddr succeeds and the NI_NOFQDN (no fully qualified domain name) flag is specified, the hostname is terminated at the first period in the name.

Handle failure of gethostbyaddr

24–29 If gethostbyaddr fails (which, unfortunately is all too common given the number of misconfigured DNS servers on the Internet; see Section 14.8 of TCPv3) and the NI_NAMEREQD flag was specified, an error is returned. Otherwise the address string corresponding to the IP address is formed by inet_ntop.

Return service string

32–42 If the NI_NUMERICSERV flag is specified, just the decimal port number is returned. Otherwise getservbyport is called. The final argument is a null pointer unless the NI_DGRAM flag is specified. If getservbyport fails, the decimal port number is returned instead.

libgai/gn_ipv46.c

```
int
gn_ipv46(char *host, size_t hostlen, char *serv, size_t servlen,
         void *aptr, size_t alen, int family, int port, int flags)
{
    char   *ptr;
    struct hostent *hptr;
    struct servent *sptr;

    if (hostlen > 0) {
        if (flags & NI_NUMERICHOST) {
            if (inet_ntop(family, aptr, host, hostlen) == NULL)
                return (1);
        } else {
            hptr = gethostbyaddr(aptr, alen, family);
            if (hptr != NULL && hptr->h_name != NULL) {
                if (flags & NI_NOFQDN) {
                    if ( (ptr = strchr(hptr->h_name, '.')) != NULL)
                        *ptr = 0;   /* overwrite first dot */
                }
                snprintf(host, hostlen, "%s", hptr->h_name);
            } else {
                if (flags & NI_NAMEREQD)
                    return (1);
                if (inet_ntop(family, aptr, host, hostlen) == NULL)
                    return (1);
            }
        }
    }
    if (servlen > 0) {
        if (flags & NI_NUMERICSERV) {
            snprintf(serv, servlen, "%d", ntohs(port));
        } else {
            sptr = getservbyport(port, (flags & NI_DGRAM) ? "udp" : NULL);
            if (sptr != NULL && sptr->s_name != NULL)
                snprintf(serv, servlen, "%s", sptr->s_name);
            else
                snprintf(serv, servlen, "%d", ntohs(port));
        }
    }
    return (0);
}
```

libgai/gn_ipv46.c

Figure 11.42 `gn_ipv46` function: handle IPv4 and IPv6 socket address structures.

11.17 Summary

`getaddrinfo` is a useful function that lets us write protocol-independent code. But calling it directly takes a few steps, and there are still repetitive details that must be handled for different scenarios: go through all the returned structures, ignore error returns from `socket`, set the `SO_REUSEADDR` socket option for TCP servers, and the like. We simplify all these details with our five functions `tcp_connect`, `tcp_listen`, `udp_client`, `udp_connect`, and `udp_server`. We showed the use of these functions in writing protocol-independent versions of our TCP and UDP daytime clients and daytime servers.

`gethostbyname` and `gethostbyaddr` are also examples of functions that are not normally reentrant. The two functions share a static result structure to which both return a pointer. We encounter this problem of reentrancy again with threads in Chapter 23 and discuss ways around the problem. We discussed the `_r` versions of these two functions that some vendors provide, which is one solution, but it requires a change in all the applications that call the functions.

Exercises

11.1 In Figure 11.8 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?

11.2 Modify Figure 11.10 to call `getnameinfo` instead of `sock_ntop`. What flags should you pass to `getnameinfo`?

11.3 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 11.15. 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). Then 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 from the first server, to see if the stealing still succeeds, because 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.

11.4 When discussing Figure 11.34 we noted that the address of `aipnext` is an argument to the `ga_aistruct` function, necessitating an extra level of indirection when referencing the variable. Why do we not make `aipnext` a global variable, instead of passing its address as an argument?

11.5 In our discussion of Unix domain at the end of Section 11.5 we mentioned that none of the IANA service names begin with a slash. Do any of these service names contain a slash?

11.6 At the end of Section 2.10 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?

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