Lecture 4 Overview

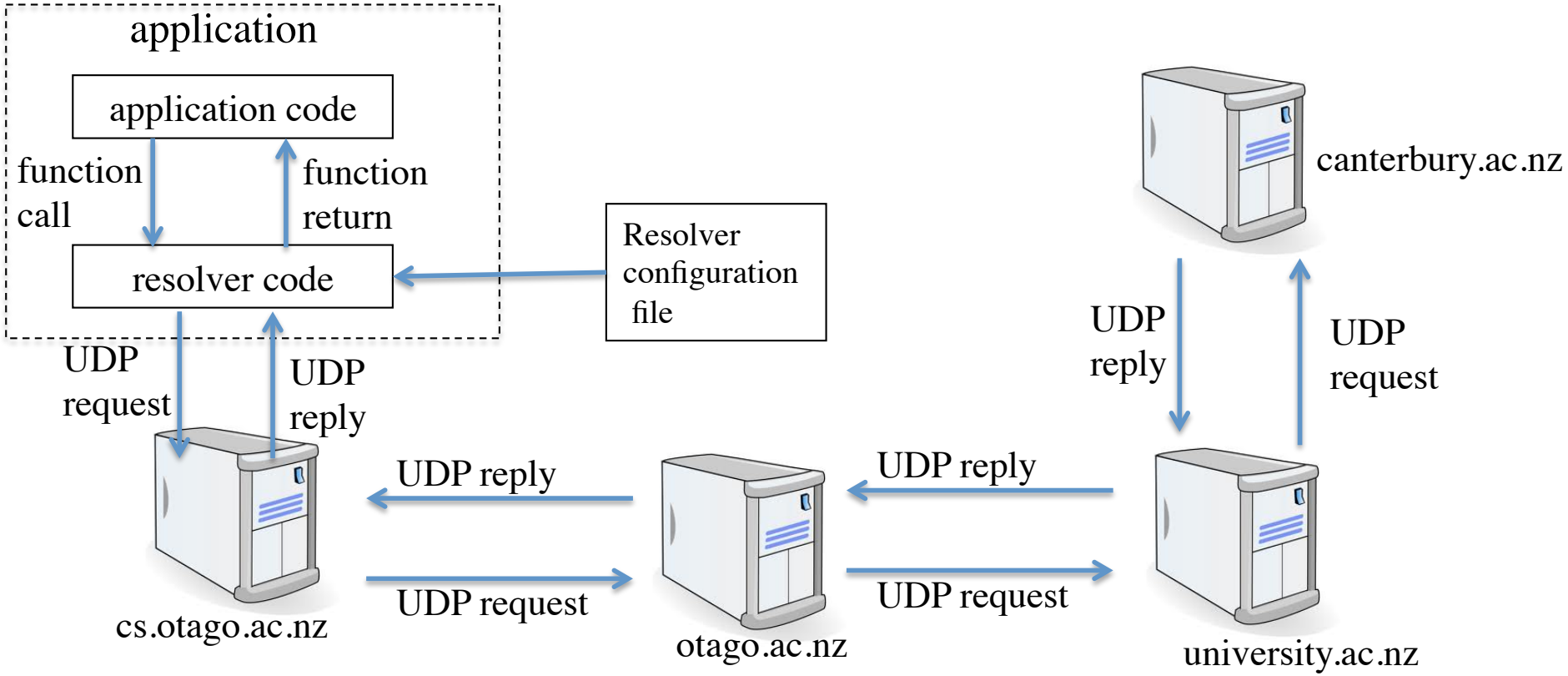
- Last Lecture
  - Socket Options and elementary UDP sockets
- This Lecture
  - Name and address conversions & IPv6
  - Source: Chapter 11
- Next Lecture
  - Multicast
  - Source: Chapter 12

Domain Name System (1)

- Low-level name: IP address
  - IPv4: 139.80.91.50
  - IPv6: fe80::7256:81ff:fea5:9ae3
- High-level name: IP name (hostname)
  - www.cs.otago.ac.nz
- How to map between IP address and IP name?
  - Domain Name System (DNS): a hierarchical distributed naming system for mappings between IP address and IP names
    - Name servers
    - Databases
    - Caches
    - Distributed

Domain Name System

- Resolver functions
  - *gethostbyname*: map a hostname into IPv4 address
  - *gethostbyaddr*: map an IPv4 address into a hostname
- Name resolution process (ping www.canterbury.ac.nz)



gethostbyname (1)

- Prototype

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

  - Returns: nonnull pointer if OK,
    NULL on error with *h_errno* set.
  - Only returns IPv4 addresses

```
struct hostent {char *h_name;
                char **h_aliases;
                int h_addrtype;
                int h_length;
                char **h_addr_list; }
```

gethostbyname (2)

- Prototype (cont)
  - Note: the error code is put into *h_errno*
    - HOST_NOT_FOUND
    - TRY_AGAIN
    - NO_RECOVERY
    - NO_DATA(identical to NO_ADDRESS)
  - The error message can be printed out using either *hstrerror(h_errno)* or *herror(char *)*.
  - Performs a query for an A record in DNS
- May be withdrawn in the future

hostent structure

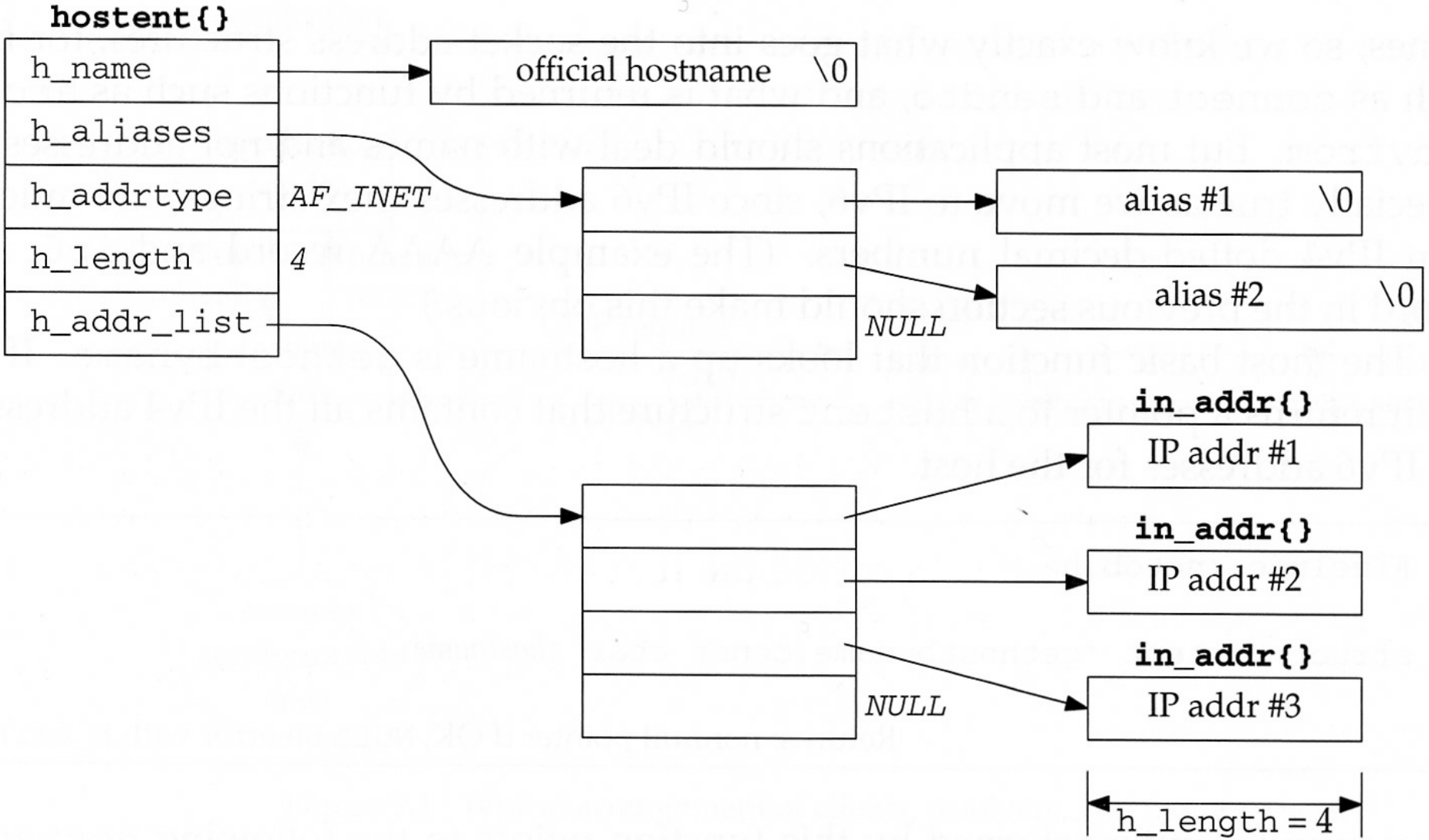


Figure 9.2 `hostent` structure and the information it contains.

gethostbyaddr (1)

- Prototype

```
#include <netdb.h>
struct hostent *gethostbyaddr(const char *addr,
                              size_t len,
                              int family)
```

  - Returns: nonnull pointer if OK,
    NULL on error with *h_errno* set.
  - The same *hostent* structure is returned, but the field of interest is *h_name*

gethostbyaddr (2)

- Prototype (cont)
  - *addr* is not a *char* *, but is a pointer to an *in_addr* structure containing the IPv4 address.
  - *len* is the size of the structure.
  - The *family* argument is AF_INET.

Reentrant Functions (1)

- *gethostbyname* and *gethostbyaddr* are not reentrant
  - Uses a static structure (hostent) to store the result (part of BIND code)
  - Problem for threads and signal handlers

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

struct hostent * gethostbyname(const char *hostname)
{  /* call DNS functions for a query */
   /* fill in host structure */
    return (&host);
}
```

Reentrant Functions (2)

```
main() {
   struct hostent *hptr;
   . . .
   signal(SIGALRM, sig_alrm);
   . . .
   hptr = gethostbyname( . . .);
   . . .
}
void sig_alrm(int signo){
   struct host ent *hptr;
   . . .
   hptr = gethostbyname( . . .);
   . . .
}
```

- There is a reentrant problem with the variable *errno*.
  - Every process has only one copy of the variable
  - But a signal handler may interfere with the value of the variable
  - Better to return the error number from functions

Reentrant Functions (3)

- There are two ways to make reentrant functions
  - The caller can prepare the necessary memory space and pass it to the function. The caller should also free the memory space later
  - The function allocates the required memory space dynamically, fills the memory space, and returns the pointer of the memory space to the caller. The caller should call some function to release the memory space later; otherwise there will be memory leakage

Reentrant functions (4)

```
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)
```

- Returns: nonnull pointer if OK, NULL on error

Get local host name 1

- *uname* returns the name of the current host

  `int uname(struct utsname *name)`

  – Returns: nonnegative value if OK, -1 on error

  – It is often used along with *gethostbyname*

```
struct utsname {char sysname[];
                char nodename[];
                char release[];
                char version[];
                char machine[];}
```

Get local host name 2

- *gethostname* returns the name of current host

```
int gethostname(char *name,
                size_t namelen)
```

  - Returns: 0 if OK, -1 on error

getservbyname

- Converts a service name to a port number

```
struct servent *getservbyname(const char *servname,
                              const char *protoname)
struct servent {char *s_name;
                char **s_aliases;
                int s_port;   //network-byte order
                char *s_proto;}
```

- Returns: nonnull pointer if OK, NULL on error
- *protoname* can be NULL or point to “udp” or “tcp”

getservbyport

- Converts a port number to a service name

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

- Returns: nonnull pointer if OK, NULL on error

getaddrinfo (1)

- *getaddrinfo* handles both name to address and service to port translation.
- It returns *sockaddr* structures instead of a list of addresses, be used by the socket functions directly.

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

  - Returns: 0 if OK, nonzero on error
  - *hostname* is either a hostname or an address string
  - *service* is either a service name or a decimal port number string

getaddrinfo (2)

- *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

```
struct addrinfo {int ai_flags;
                 int ai_family;
                 int ai_socktype;
                 int ai_protocol;
                 size_t ai_addrlen;
                 char *ai_canonname;
                 struct sockaddr *ai_addr;
                 struct addrinfo *ai_next;}
```

Getaddrinfo (3)

- ai_flags
  - AI_PASSIVE, AI_CANONNAME
- ai_family
  - AF_xxx
- ai_socktype
  - SOCK_DGRAM, SOCK_STREAM
- ai_protocol
  - IPROTO_UDP, IPROTO_TCP

getaddrinfo (4)

- ai_addrlen
  - length of ai_addr
- ai_canonname
- ai_addr
  - socket structure containing information
- ai_next
  - pointer to the next addrinfo structure or NULL

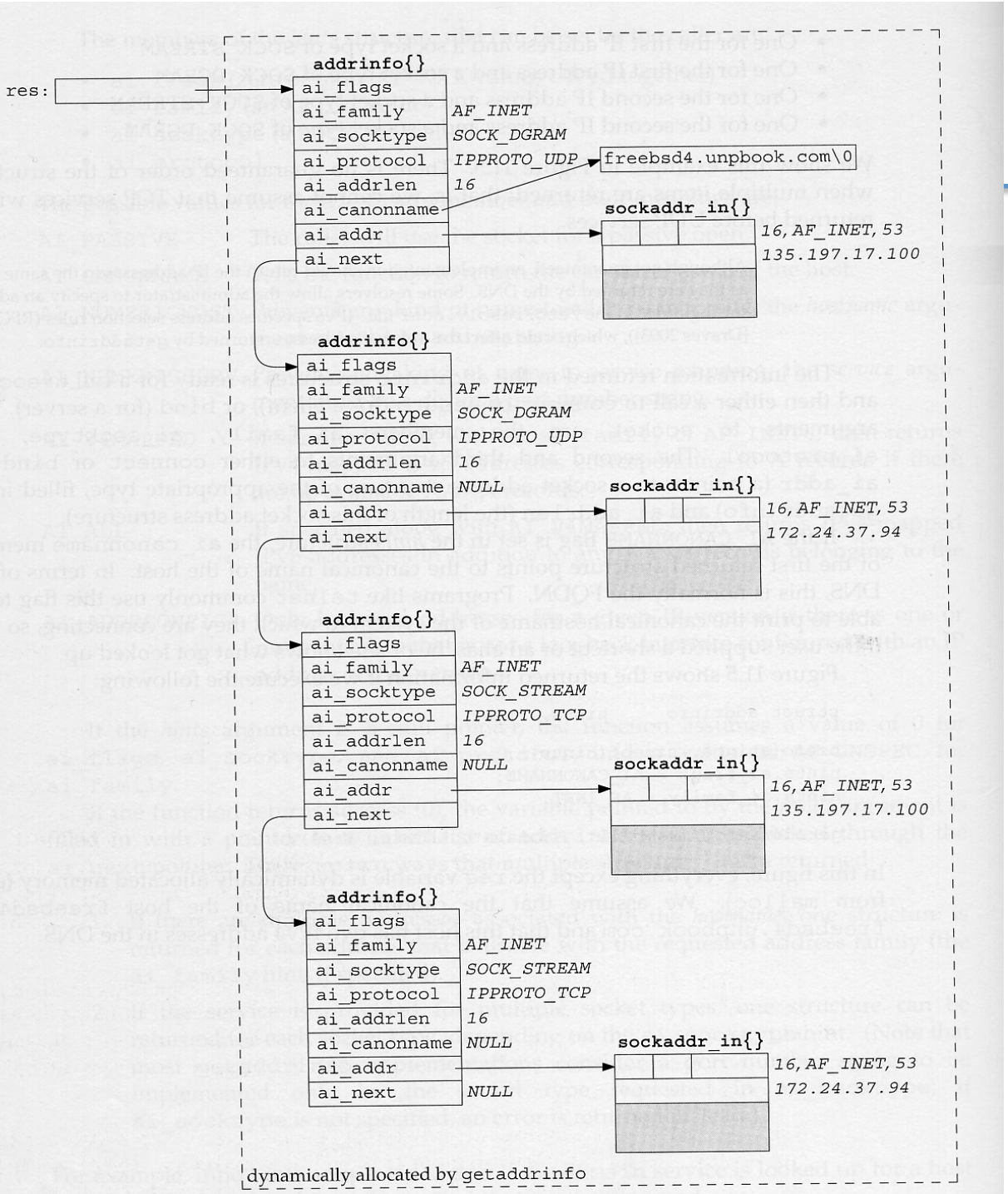


Figure 11.5 Example of information returned by `getaddrinfo`.

Getaddrinfo (5)

- If *hints* is NULL, *addrinfo* assumes a value of 0 for *ai_flags*, *as_socktype*, *ai_protocol*, and AF__UNSPEC for *ai_family*
- Multiple structures returned if:
  - Multiple addresses for *hostname*
  - Service is provided by multiple socket types, depending on the *ai_socktype* hint
- Order not determined

Getaddrinfo (6)

- What to do with the returned results
  - Arguments for *socket* are *ai_family*, *ai_socktype*, *ai_protocol*
  - 2nd and 3rd arguments to c*onnect* or *bind* are a*i_addr* and *ai_len*

getaddrinfo (7)

ai_socktype hint	Service is a name, service provided by:						Service is a port number
	TCP only	UDP only	SCTP only	TCP and UDP	TCP and SCTP	TCP, UDP, and SCTP	
0	1	1	1	2	2	3	error
SOCK_STREAM	1	error	1	1	2	2	2
SOCK_DGRAM	error	1	error	1	error	1	1
SOCK_SEQPACKET	error	error	1	error	1	1	1

Figure 11.6 Number of `addrinfo` structures returned per IP address.

gai_strerror

- Errors returned from *getaddrinfo*

Constant	Description
EAI_AGAIN	Temporary failure in name resolution
EAI_BADFLAGS	Invalid value for ai_flags
EAI_FAIL	Unrecoverable failure in name resolution
EAI_FAMILY	ai_family not supported
EAI_MEMORY	Memory allocation failure
EAI_NONAME	*hostname* or *service* not provided, or not known
EAI_OVERFLOW	User argument buffer overflowed (*getnameinfo*() only)
EAI_SERVICE	*service* not supported for ai_socktype
EAI_SOCKTYPE	ai_socktype not supported
EAI_SYSTEM	System error returned in errno

Figure 11.7 Nonzero error return constants from getaddrinfo.

- *gai_strerror* takes one of these values and returns a pointer to the error string

```
const char *gai_strerror(int error)
```

freeaddrinfo

- All storage returned by *getaddrinfo* is dynamically allocated. This storage is returned by calling *freeaddrinfo*

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

- *ai* should point to the first *addrinfo* structure

Hostname specified by caller	Address family specified by caller	Hostname string contains	Result	Action
non-null 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	AAAA record search *and* A record search
		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	AAAA record search
			If ai_flags contains AI_V4MAPPED, all AAAA records returned as sockaddr_in6{}s *else* all A records returned as IPv4-mapped IPv6 sockaddr_in6{}s	AAAA record search if no results then A record search
			If ai_flags contains AI_V4MAPPED and AI_ALL, all AAAA records returned as sockaddr_in6{}s *and* all A records returned as IPv4-mapped IPv6 sockaddr_in6{}s	AAAA record search *and* A record search
		hex string	One sockaddr_in6{}	inet_pton(AF_INET6)
		dotted-decimal	Looked up as hostname	
	AF_INET	hostname	All A records returned as sockaddr_in{}s	A record search
		hex string	Looked up as hostname	
		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 11.8 Summary of getaddrinfo and its actions and results.

Function Example (host_serv)

- host_serv (host_serv.c): does not require the caller to alloate a *hints* structure and fill it in. The two fields, the address family and the socket type, are arguments.

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

  - Returns an addrinfo structure for the given host and service
  - It always returns a canonical name

Function Example (host_serv)

```
struct addrinfo *
host_serv(const char *host, const char *serv, int family, int
socktype)
{
        int                             n;
        struct addrinfo     hints, *res;

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

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

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

Function Example (tcp_connect)

- tcp_connect (tcp_connect.c): performs normal client steps: create a TCP socket and connect to a server.

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

  - Returns an connected socket descriptor
  - See attachment for source code

Function Example (tcp_listen)

- tcp_listen (lib/tcp_listen.c)

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

  - Returns a listening socket descriptor
  - See attachment for source code

Function Example (udp_client)

- udp_client (udp_client.c): creating an unconnected UPD socket and returning three items: socket descriptor, address pointer to a socket address structure and socket length

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

  - Returns an unconnected socket descriptor
  - See attachment for source code

Function Example (udp_connect)

- udp_connect (udp_connect.c): creating a connected UPD socket.

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

  - Returns a connected socket descriptor
  - See attachment for source code

Function Example (udp_server)

- udp_server (udp_server.c): the arguments are the same as *tcp_listen*

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

  - Returns an unconnected socket descriptor
  - *hostname* is optional and *service* is required
  - See attachment for source code

Getnameinfo (1)

- The complement of the *getaddrinfo*: it takes a socket address and returns a character string describing the host and another string describing the service.
- Provide this information in a protocol-independent fashion

```
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

getnameinfo (2)

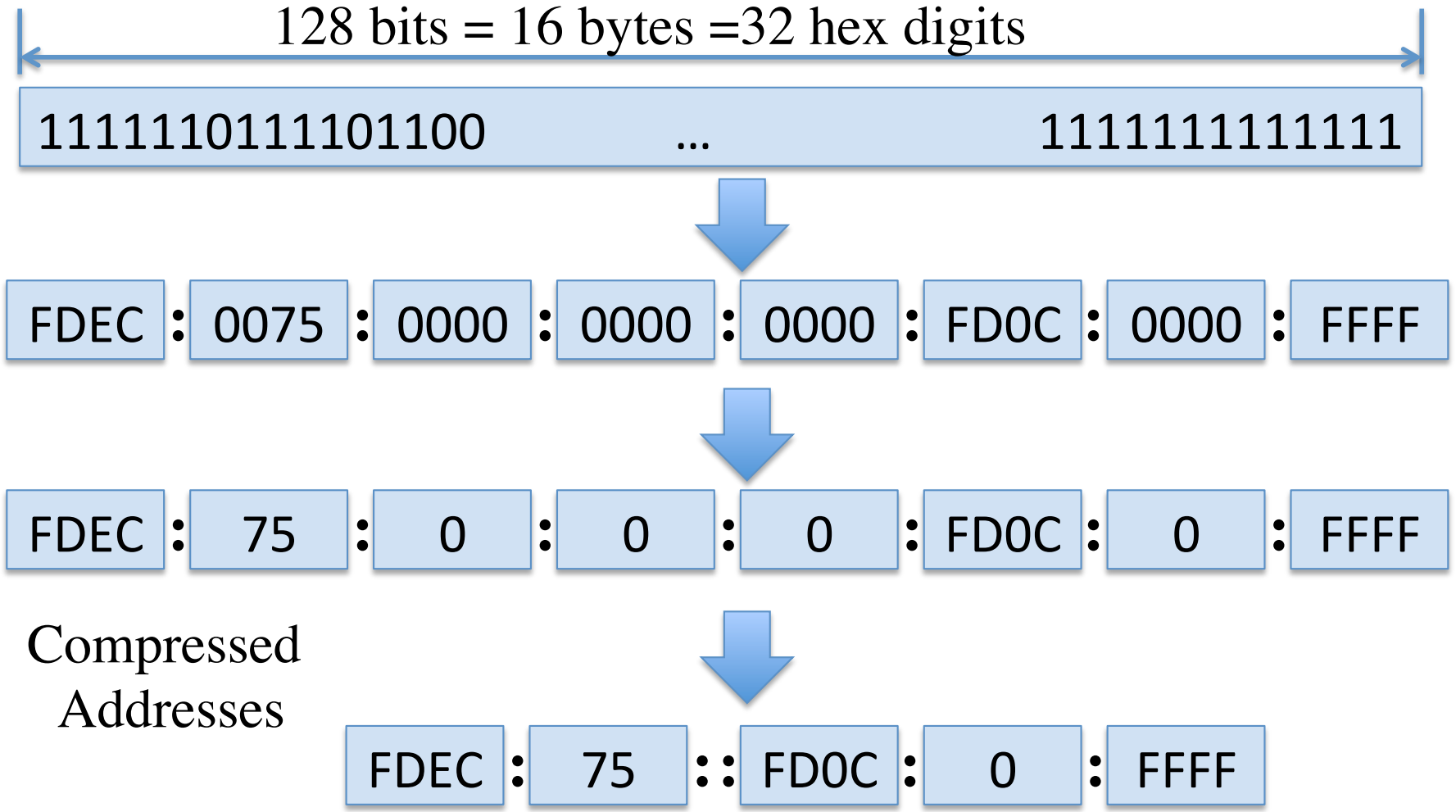
- Flags for *getnameinfo*

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`.

IPv6 Addresses

- Hexadecimal Colon Notation



Address Space Allocation

- IPv6 address space: 2^{128}
- The 128-bit address space is divided based on the most significant bits (Format Prefix):
  - 0000 0000, reserved
  - 0000 001, reserved for NSAP addresses
  - 0000 010, reserved for IPX addresses
  - 001, global unicast address
  - 100, reserved for geographic-based addresses
  - 1111 1110 10 (FE80), link local addresses
  - 1111 110 (FC), unique local unicast
  - 1111 1111 (FF), multicast addresses
- The rest addresses are unassigned (85%)

More on Compressed IPv6 Addresses

- CIDR notation

  IPv6-address / prefix-length

- Example:

  12AB:0000:0000:CD30:0000:0000:0000:0000/60

  The following are legal representations

  - 12AB::CD30:0:0:0:0/60
  - 12AB:0:0:CD30::/60

  The following are **not legal** representations

  - 12AB::CD30/60
  - 12AB::CD3/60
  - 12AB::CD30::/60

Typical Host Addresses (1)

- Loopback – ::1 (belong to reserved 0x00)
  - it is used by a node to send an IPv6 packet to itself
  - It must never be assigned to any interface
- Link local – fe80::/10
  - Used on each link for address autoconfiguration and for neighbor discovery functions
  - May specify interface name using % suffix
  - Used for LAN communication without a router

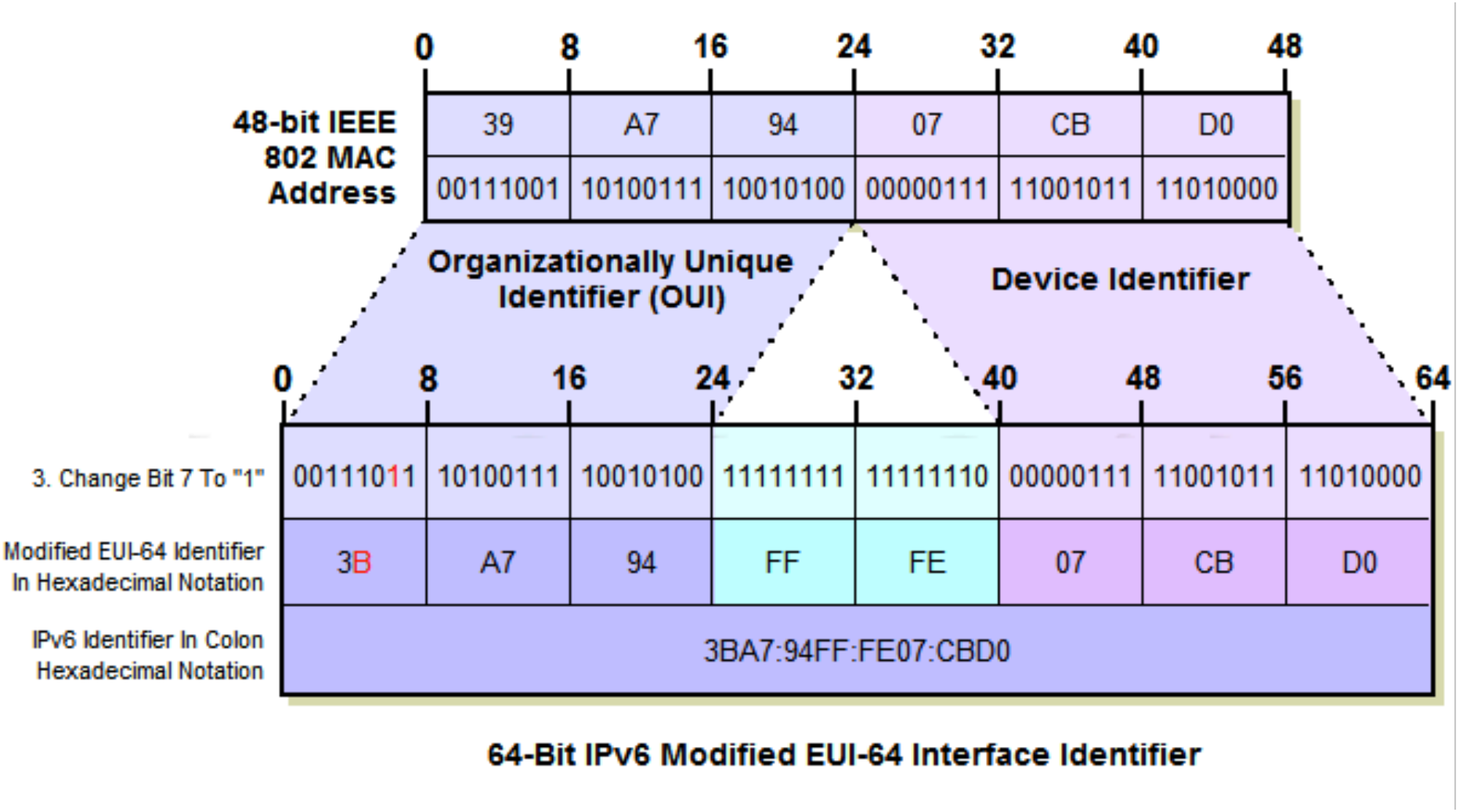
Typical Host Addresses (2)

- Site local – fec0::/10
  - RFC 1884 (1995)
  - The term “site” was somewhat ambiguous
  - Has been officially replaced by unique-local addresses
- Unique local – fc::/7
  - RFC 4193 (2005)
  - Used for local communication in private networks, or spanning a limited number of sites or organizations
  - They are not routable in the global IPv6 Internet

IPv6 address types -- Unicast

- Unicast
  - Address of a single interface
  - An interface may have multiple unicast addresses
  - Any unicast address consists of a 64-bit prefix and 64-bit EUI-64 ID based on the MAC address
  - The 64-bit prefix is from the router.
  - The prefix consists of 48(or 56)-bit prefix from ISP and 16(or 8)-bit subnet id set by the user.

EUI-64


0
8
16
24
32
40
48
48-bit IEEE 802 MAC Address
39
A7
94
07
CB
D0
00111001
10100111
10010100
00000111
11001011
11010000
Organizationally Unique Identifier (OUI)
Device Identifier
0
8
16
24
32
40
48
56
64
3. Change Bit 7 To "1"
00111011
10100111
10010100
11111111
11111110
00000111
11001011
11010000
Modified EUI-64 Identifier In Hexadecimal Notation
3B
A7
94
FF
FE
07
CB
D0
IPv6 Identifier In Colon Hexadecimal Notation
3BA7:94FF:FE07:CBD0
64-Bit IPv6 Modified EUI-64 Interface Identifier

Global Unicast Addresses

- Starts with 001 ie. 2* to 3*
- Includes anycast addresses
- Further partitioned
  - 2002::/16 for 6to4 tunneling
- Example
  - inet6 2001:388:c004:2:20d:93ff:feea:ee7a prefixlen 64

```c
/* include host_serv */
#include    "unp.h"

struct addrinfo *
host_serv(const char *host, const char *serv, int family, int socktype)
{
    int             n;
    struct addrinfo hints, *res;

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

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

    return(res);    /* return pointer to first on linked list */
}
/* end host_serv */

/*
 * There is no easy way to pass back the integer return code from
 * getaddrinfo() in the function above, short of adding another argument
 * that is a pointer, so the easiest way to provide the wrapper function
 * is just to duplicate the simple function as we do here.
 */

struct addrinfo *
Host_serv(const char *host, const char *serv, int family, int socktype)
{
    int             n;
    struct addrinfo hints, *res;

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

    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
        err_quit("host_serv error for %s, %s: %s",
                 (host == NULL) ? "(no hostname)" : host,
                 (serv == NULL) ? "(no service name)" : serv,
                 gai_strerror(n));

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

```
#include    "unp.h"

int
main(int argc, char **argv)
{
    char            *ptr, **pptr;
    char            str[INET6_ADDRSTRLEN];
    struct hostent  *hptr;

    while (--argc > 0) {
        ptr = *++argv;
        if ( (hptr = gethostbyname(ptr)) == NULL) {
            err_msg("gethostbyname error for host: %s: %s",
                    ptr, hstrerror(h_errno));
            continue;
        }
        printf("official hostname: %s\n", hptr->h_name);

        for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
            printf("\talias: %s\n", *pptr);

        switch (hptr->h_addrtype) {
        case AF_INET:
#ifdef  AF_INET6
        case AF_INET6:
#endif
            pptr = hptr->h_addr_list;
            for ( ; *pptr != NULL; pptr++)
                printf("\taddress: %s\n",
                    Inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
            break;

        default:
            err_ret("unknown address type");
            break;
        }
    }
    exit(0);
}
```

```
#include    "unp.h"

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

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

    if ( (hp = gethostbyname(argv[1])) == NULL)
        err_quit("hostname error for %s: %s", argv[1], hstrerror(h_errno))
            ;

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

    pptr = (struct in_addr **) hp->h_addr_list;
    for ( ; *pptr != NULL; pptr++) {
        sockfd = Socket(AF_INET, SOCK_STREAM, 0);

        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = sp->s_port;
        memcpy(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));
        printf("trying %s\n",
               Sock_ntop((SA *) &servaddr, sizeof(servaddr)));

        if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) == 0)
            break;      /* success */
        err_ret("connect error");
        close(sockfd);
    }
    if (*pptr == NULL)
        err_quit("unable to connect");

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

```c
/* include tcp_connect */
#include    "unp.h"

int
tcp_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_STREAM;

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

    freeaddrinfo(ressave);

    return(sockfd);
}
/* end tcp_connect */

/*
 * We place the wrapper function here, not in wraplib.c, because some
 * XTI programs need to include wraplib.c, and it also defines
 * a Tcp_connect() function.
 */

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

```c
/* include tcp_listen */
#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);
}
/* end tcp_listen */

/*
 * We place the wrapper function here, not in wraplib.c, because some
 * XTI programs need to include wraplib.c, and it also defines
 * a Tcp_listen() function.
 */

int
Tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
{
    return(tcp_listen(host, serv, addrlenp));
}
```

```c
/* include udp_client */
#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);
}
/* end udp_client */

int
Udp_client(const char *host, const char *serv, void **saptr, socklen_t *
    lenptr)
{
    return(udp_client(host, serv, saptr, lenptr));
}
```

```c
/* include udp_connect */
#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);
}
/* end udp_connect */

int
Udp_connect(const char *host, const char *serv)
{
    int     n;

    if ( (n = udp_connect(host, serv)) < 0) {
        err_quit("udp_connect error for %s, %s: %s",
                     host, serv, gai_strerror(-n));
    }
    return(n);
}
```

```c
/* include udp_server */
#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);
}
/* end udp_server */

int
Udp_server(const char *host, const char *serv, socklen_t *addrlenp)
{
    return(udp_server(host, serv, addrlenp));
}
```