KWANGWOON UNIVERSITY

Chapter .11

Name and Address Conversions

DNS(Donmain Name System)

- The DNS is used primarily to **map between hostnames and IP addresses.**

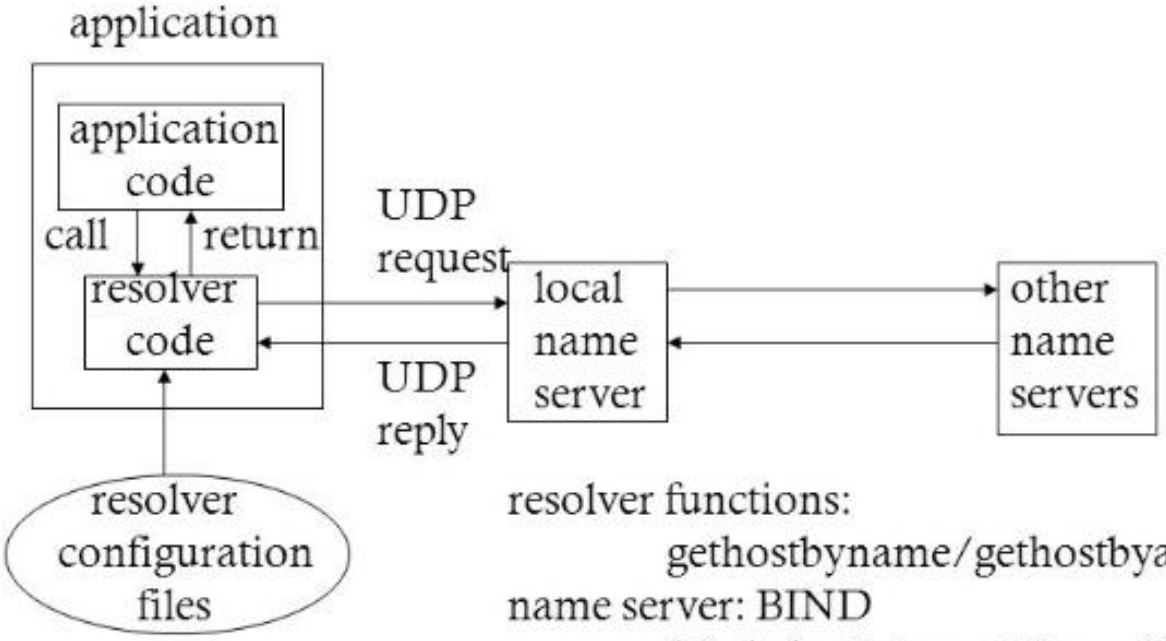
- A hostname can be either a simple name, such as solaris or freebsd, or a fully qualified domain name '(FQDN), such as solaris.unpbook.com.

Resouce record

- Entries in DNS: resource records (RRs) for a host
  - A record: maps a hostname to a 32-bit IPv4 addr
  - AAAA (quad A) record: maps to a 128-bit IPv6 addr
  - PTR record: maps IP addr to hostname
  - MX record: specifies a mail exchanger of the host
  - CNAME record: assigns canonical name for common services

e.g.	solaris	IN	A	206.62.226.33
		IN	AAAA	5f1b:df00:ce3e:e200:0020:0800:2078:e3e3
		IN	MX	5 solaris.kohala.com
		IN	MX	10 mailhost.kohala.com
		IN	PTR	33.226.62.206.in-addr.arpa
	www	IN	CNAME	bsdi.kohala.com

Resolver and name server



resolver functions:
gethostbyname/gethostbyaddr
name server: BIND
(Berkeley Internet Name Domain)
static hosts files (DNS alternatives):
/etc/hosts
resolver configuration file (specifies name server IPs):
/etc/resolv.conf

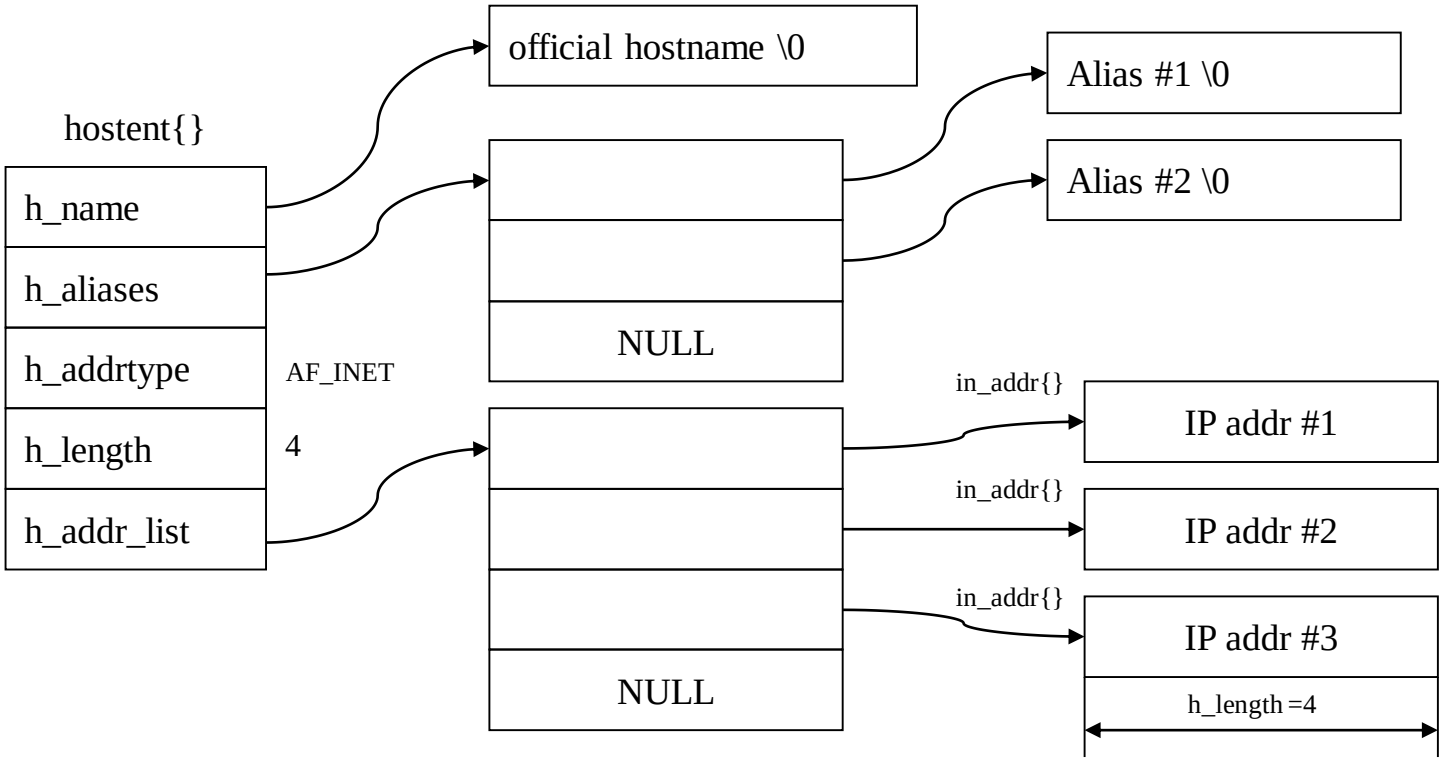
3

gethostbyname

- **Gethostbyname** is the most basic function that looks up a hostname.
- If successful, it returns a **pointer to a hostent structure** that contains all the IPv4 addresses for the host. However, it is limited in that it can only return IPv4 addresses

```
#include <netdb.h>
struct hostent *gethostbyname (const char *hostname);
        returns: nonnull pointer if OK, NULL on error with h_errno set
struct hostent {
        char    *h_name;        /* official (canonical) name of host */
        char    **h_aliases;    /* ptr to array of ptrs to alias names */
        int     h_addrtype;     /* host addr type: AF_INET
        int     h_length;       /* length of address: 4
        char    **h_addr_list;  /* ptr to array of ptrs with IPv4/
};
#define h_addr h_addr_list[0]   /* first address in list */
```

gethostbyname



gethostbyname

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

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

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

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

gethostbyname

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

- Web server with multiple IPv4 addresses

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

- Name having a CNAME record

gethostbyaddr

- Prototype

```
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*

Reentrant

Reentrant functions

- *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

Reentrant

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

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

struct hostent *
gethostbyname2 (const char *hostname, int family)
{
    /* call DNS functions for A or AAAA query */
    /* fill in host structure */
    return (&host) ;
}

struct hostent *

gethostbyaddr (const char *addr, socklen_t len, int family)
{
    /* call DNS functions for PTR query in in-addr.arpa domai
    /* fill in host structure */
    return (&host);
}
```

Reentrant problem

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

Reentrant

Reentrant functions

- 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

Reentrant functions

- 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

Reentrant functions

```
struct hostent *gethostbyname_r(const char *hostname,
                                struct host *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

getservbyname & getservbyport

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;
                char *s_proto;}
```

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

getservbyname & getservbyport

- Typical calls to this function could be as follows:

```
struct servent *sptr;

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

- lines from the /etc/services file are:

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

getservbyname & getservbyport

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

```
struct servent *sptr;

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

getaddrinfo

```
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
- *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

getaddrinfo

```
Struct addrinfo {
    int ai_flags;          hints
    int ai_family;
    int ai_socktype;
    int ai     protocol;
    socklen_t  ai_addrlen;
    char  *ai_canonname;
    sturct sockaddr* ai_addr;
    struct addrinfo * ai_next;
    }
```

- ai_flags
  - AI_PASSIVE, AI_CANONNAME
- ai_family
  - AF_xxx
- ai_socktype
  - SOCK_DGRAM, SOCK_STREAM
- ai_protocol
  - IPROTO_UDP, IPROTO_TCP
- ai_addrlen
  - length of ai_addr
- ai_canonname
- ai_addr
  - socket structure containing information
- ai_next
  - pointer to the next addrinfo structure or NULL

getaddrinfo

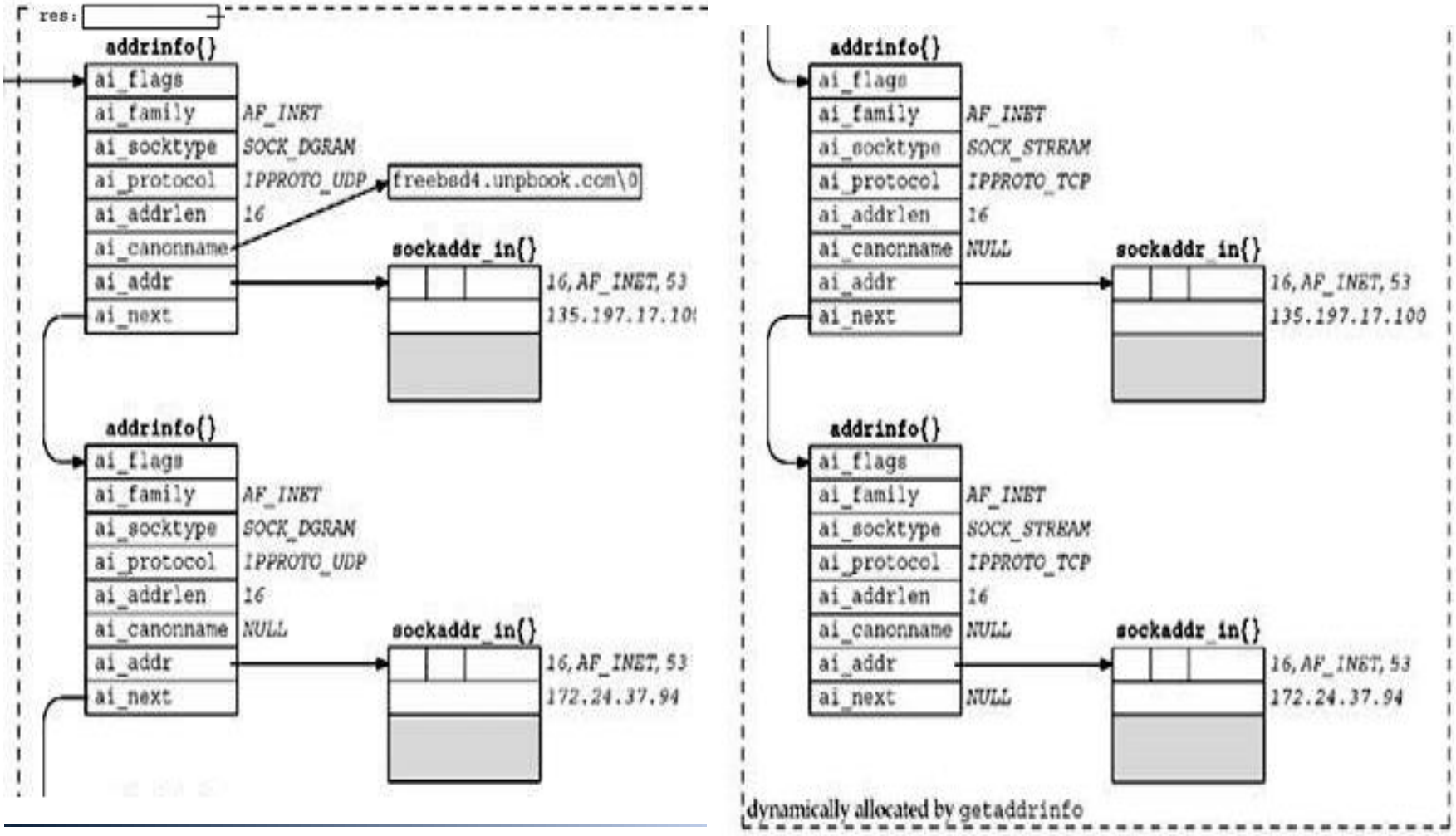
ai_flags elements

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

getaddrinfo

- 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



getaddrinfo

- testga
  - f inet: the address family
  - c : the canonical name
  - h bsdi: the hostname
  - s domain: the service name

```
freebsd % testga -f inet -c -h freebsd4 -s domain

socket (AF_INET, SOCK_DGRAM, 17), ai_canonname = freebsd4.unpbook.com
         address: 135.197.17.100:53
socket (AF_INET, SOCK_DGRAM, 17)
         address: 172.24.37.94:53
socket (AF_INET, SOCK_STREAM, 6), ai_canonname = freebsd4.unpbook.com
         address: 135.197.17.100:53
socket (AF_INET, SOCK_STREAM, 6)
         address: 172.24.37.94:53
```

getaddrinfo

Testga -f inet -t stream -h gateway.tuc.noao.edu -s daytime
(-t socktype, -s service name)

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

socket (AF_INET, SOCK_STREAM, 6)
         address: 140.252.108.1:13
socket (AF_INET, SOCK_STREAM, 6)
         address: 140.252.1.4:13
socket (AF_INET, SOCK_STREAM, 6)
         address: 140.252.104.1:13
```

testga -h alpha -s ftp

```
freebsd % testga -h aix -s ftp -t stream

socket (AF_INET6, SOCK_STREAM, 6)
         address: [3ffe:b80:1f8d:2:204:acff:fe17:bf38]:21
socket (AF_INET, SOCK_STREAM, 6)
         address: 192.168.42.2:21
```

gai_strerror

gai_strerror

const char *gai_strerror(int error)

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

freeaddrinfo

freeaddrinfo

- All storage returned by *getaddrinfo* is dynamically allocated.

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

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

Function based on getaddrinfo

- host_serv (lib/host_serv.c)

  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

- tcp_connect (lib/tcp_connect.c)

  int tcp_connect(const char *_hostname_,
  const char *_service_);

  – Returns an connected socket descriptor

Function based on getaddrinfo

```
 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 }
```

```
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);
```

Function based on getaddrinfo

- tcp_listen (lib/tcp_listen.c)

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

  – Returns a listening socket descriptor

```
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;
```

Function based on getaddrinfo

```
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);
```

Function based on getaddrinfo

- udp_client (lib/udp_client.c)

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

  – Returns an unconnected socket descriptor

```
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;
```

Function based on getaddrinfo

```
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);
```

Function based on getaddrinfo

- udp_connect (lib/udp_connect.c)

  int udp_connect(const char *_hostname_,
  const char *_service_)

  – Returns a connected socket descriptor

```
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);
```

Function based on getaddrinfo

- udp_server (lib/udp_server.c)

  int udp_server(const char *_hostname_,
  const char *_service_,
  socklen_t *_lenptr_)

  – Returns an unconnected socket descriptor

```
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;
```

Function based on getaddrinfo

```
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);
```

getnameinfo

getnameinfo

```
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

getaddrinfo

- 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`.