Overview: Application Development for IPv6
Application Development – Design Impact
By deployment of Internet Protocol Version 6 (IPv6), the application programmer has to cope with socket connection with multiple address families, i.e. AF_INET and AF_INET6.

How does a programmer handle multiple address families with ease?
- Allows for maximum portability

- All (most) address family dependent stuff is removed from the application by using these functions:
  - `getnameinfo`
  - `getaddrinfo`

- Your code doesn’t need to care if the calling host is IPv4 or IPv6
In order to allow applications to communicate with other IPv6 nodes, the first priority is to convert the applications supporting both IPv4 and IPv6
- IP version-independent structures & APIs
- Must iterate over all addresses returned to choose best address

The applications have to work properly in dual-stacked nodes (where both IPv4 and IPv6 addresses are configured)

The applications have to work properly in IPv4-only nodes (where no IPv6 address is configured)

The applications have to work properly in IPv6-only nodes (where no IPv4 address is configured)
Coding Areas Requiring Changes

- Socket structures
  - Use “generic” socket structure
  - Avoid IP specific structures

- Name to Address Translation functions
New Socket structure

```c
struct sockaddr_storage
{
    sa_family_t ss_family; /*Address family*/
    __ss_aligntype __ss_align; /*Force desired alignment*/
    char __ss_padding[__ss_PADSIZE];
};
```

- Use when you need to allocate space for a socket structure
  - Address family independent: can accommodate IPv4, IPv6, or other AF
### Comparison of Structures

<table>
<thead>
<tr>
<th>IPv4</th>
<th>IPv6</th>
<th>storage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sockaddr_in{}</strong></td>
<td><strong>sockaddr_in6{}</strong></td>
<td><strong>sockaddr_storage{}</strong></td>
</tr>
<tr>
<td>family = AF_INET</td>
<td>family = AF_INET6</td>
<td>family</td>
</tr>
<tr>
<td>16-bit port #</td>
<td>16-bit port #</td>
<td>(opaque)</td>
</tr>
<tr>
<td>32-bit IPv4 addresses</td>
<td>32-bit flow label</td>
<td></td>
</tr>
<tr>
<td>(unused)</td>
<td>128-bit IPv6 address</td>
<td></td>
</tr>
<tr>
<td>Fixed length – 16 bytes</td>
<td></td>
<td>Longest on system</td>
</tr>
<tr>
<td>32-bit scope ID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Protocol Independent Programming
Examples of sockaddr_storage Usage

/* IPv4/IPv6 Support code */
struct sockaddr_storage addr;
socklen_t addrlen;
bind(sockfd, (struct sockaddr *)&addr, addrlen)

/* IPv4/IPv6 Support code */
struct sockaddr_storage addr;
socklen_t addrlen;
accept(sockfd, (struct sockaddr *)&addr, &addrlen)
**Usage example 1**

```c
struct sockaddr_storage ss;
...
if (logging) {
    sval = sizeof(sockaddr_storage);
    if (getpeername(0, (struct sockaddr *)&ss, &sval) < 0)
        err("getpeername: %s", strerror(errno));
}
```

**Usage example 2**

```c
struct sockaddr_storage ss;
struct sockaddr_in6 *sin6;
...
sin6 = (struct sockaddr_in6 *) &ss;
```
Structure Allocation & Casting

Structure Allocation

```c
struct sockaddr_in svrAddr4;    /* IPv4 */
struct sockaddr_in6 svrAddr6;   /* IPv6 */
struct sockaddr_storage svrAddr; /* both */
```

Socket Functions Require (struct sockaddr *)

```c
/* IPv4 */
bind (serverfd, (struct sockaddr *)&svrAddr4, length);
/* IPv6 */
bind (serverfd, (struct sockaddr *)&svrAddr6, length);
/* Both */
bind (serverfd, (struct sockaddr *)&svrAddr, length);
```
Coding Areas Requiring Changes

- Socket structures
- Name to Address Translation functions
<table>
<thead>
<tr>
<th>IPv4 only or Protocol specific</th>
<th>Protocol Independent</th>
</tr>
</thead>
<tbody>
<tr>
<td>gethostbyname()</td>
<td>getaddrinfo()</td>
</tr>
<tr>
<td>getservbyname()</td>
<td></td>
</tr>
<tr>
<td>inet_aton()</td>
<td></td>
</tr>
<tr>
<td>inet_pton()</td>
<td></td>
</tr>
<tr>
<td>gethostbyaddr()</td>
<td></td>
</tr>
<tr>
<td>getservbyport()</td>
<td></td>
</tr>
<tr>
<td>inet_ntoa()</td>
<td></td>
</tr>
<tr>
<td>inet_ntop()</td>
<td></td>
</tr>
<tr>
<td>getnameinfo()</td>
<td></td>
</tr>
</tbody>
</table>
getaddrinfo() returns a linked list of address info structures
  - Returns structure containing a socket structure

Some hints about the desired kinds of addresses may be given in the hints argument

getaddrinfo dynamically allocates memory unlike its predecessor functions

The corresponding freeaddrinfo() function is called to free the buffers that are allocated by getaddrinfo().
struct addrinfo hints, *res;

memset(0, &hints, sizeof(hints));
hints.ai_flags = AI_PASSIVE;
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM; /* SOCK_DGRAM */
getaddrinfo(NULL, DAYTIME_PORT, &hints, &res);
/* ... */
freeaddrinfo(res);
```c
struct sockaddr_storage clientAddr;
char clientHost[ADDRLEN];
char clientPort[PORTLEN];
/* ... */
connectedfd = accept(serverfd,
    (struct sockaddr *)&clientAddr,
    &alen);
getnameinfo((struct sockaddr *)&clientAddr, addrLen,
    clientHost, sizeof(clientHost),
    clientPort, sizeof(clientPort),
    NI_NUMERICHOST);
printf("Request from host=[%s] port=[%s]\n",
    clienthost, clientservice);
```
Rules of Thumb

- Do not hardcode knowledge about particular AF
- Use getaddrinfo() and getnameinfo() everywhere
- Avoid struct in_addr and struct in6_addr
The construct like the one below should be avoided:

/* BAD EXAMPLE */

switch (sa->sa_family) {
    case AF_INET:
        salen = sizeof(struct sockaddr_in);
        break;
    ...
}

Instead, use res->ai_addrlen returned by getaddrinfo
Example of Protocol Independent Code

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

main() {
    struct addrinfo hints, *res, *res0;
    int error;
    int s;
    const char *cause = NULL;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    error = getaddrinfo("www.company.example", "ftp",
                        &hints, &res0);
    if(error) {
        fprintf(stderr, "%s", gai_strerror(error));
        exit(1);
    }

    s = socket(hints.ai_family, hints.ai_socktype, 0);
    if(s == INVALID_SOCKET) {
        cause = gai_strerror(error);
        fprintf(stderr, "socket error: %s", cause);
        exit(1);
    }

    // Setup the socket
    // Connect to the server
    // Send and receive data
}
```
s = -1;
for (res = res0; res; res = res->ai_next) {
    s = socket(res->ai_family, res->ai_socktype,
               res->ai_protocol);
    if (s < 0) {
        cause = "Error: socket";
        continue;
    }
    if (connect(s, res->ai_addr, res->ai_addrlen) < 0) {
        cause = "Error: connect";
        close(s);
        s = -1;
        continue;
    }
    cause = "Success";
    break; /* okay we got one */
}
int getaddrinfo (const char *hostname, const char *service,
       const struct addrinfo *hints,
       struct addrinfo **result);

struct addrinfo {
    int    ai_flags;       /* AI_PASSIVE, ... */
    int    ai_family;      /* AF_xxx */
    int    ai_socktype;    /* SOCK_xxx */
    int    ai_protocol;    /* 0 or IPPROTO_xxx */
    socklen_t ai_addrlen;  /* length of ai_addr */
    char   *ai_canonname;  /* canonical name */
    struct sockaddr *ai_addr;  /* binary address */
    struct addrinfo *ai_next;  /* next struct in list */
};
int getaddrinfo (const char *hostname, const char *service,
    const struct addrinfo *hints,
    struct addrinfo **result);

struct addrinfo {
    int    ai_flags;        /* AI_PASSIVE, … */
    int    ai_family;       /* AF_xxx */
    int    ai_socktype;     /* SOCK_xxx */
    int    ai_protocol;     /* 0 or IPPROTO_xxx */
    socklen_t  ai_addrlen;      /* length of ai_addr */
    char      *ai_canonname;    /* canonical name */
    struct sockaddr  *ai_addr;  /* binary address */
    struct addrinfo  *ai_next;  /* next struct in list */
};

FLAGS
AI_PASSIVE
AI_CANONNAME
AI_NUMERICHOST
AI_NUMERICSERV
AI_V4MAPPED
AI_ALL
AI_ADDRCONFIG
Hints for getaddrinfo()

- **ai_flags** (zero or more AI_xxx values OR’ed)
  - `AI_PASSIVE`, `AI_CANONNAME`, `AI_NUMERICHOST`, `AI_NUMERICSERV`, `AI_V4MAPPED`, `AI_ALL`, `AI_ADDRCONFIG`

- **ai_family** (an AF_xxx value)
  - `AF_INET`, `AF_INET6`, `AF_UNSPEC`

- **ai_socktype** (a SOCK_xxx value)
  - `SOCK_STREAM`, `SOCK_DGRAM`

- **ai_protocol**
Flags for getaddrinfo

- **AI_PASSIVE** – Used for socket passive open (i.e., server socket)
- **AI_CANONNAME** – Return canonical name of the host
- **AI_NUMERICHOST** – Prevents any kind of name-to-address mapping; the hostname 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
Flags for `getaddrinfo` (2)

- **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.
Multiple addrinfo Returned

- There are two ways that multiple addrinfo 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 (*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
# Error Return for `getaddrinfo`

Nonzero error return constants from `getaddrinfo`

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAI_AGAIN</td>
<td>Temporary failure in name resolution</td>
</tr>
<tr>
<td>EAI_BADFLAGS</td>
<td>Invalid value for ai_flags</td>
</tr>
<tr>
<td>EAI_FAIL</td>
<td>Unrecoverable failure in name resolution</td>
</tr>
<tr>
<td>EAI_FAMILY</td>
<td>ai_family not support</td>
</tr>
<tr>
<td>EAI_MEMORY</td>
<td>Memory Allocation failure</td>
</tr>
<tr>
<td>EAI_NONAME</td>
<td><code>hostname</code> or <code>service</code> not provided, or not known</td>
</tr>
<tr>
<td>EAI_OVERFLOW</td>
<td>User argument buffer overflow</td>
</tr>
<tr>
<td>EAI_SERVICE</td>
<td>Service not supported for ai_socktype</td>
</tr>
<tr>
<td>EAI_SOCKTYPE</td>
<td>ai_socktype not supported</td>
</tr>
<tr>
<td>EAI_SYSTEM</td>
<td>System error returned in errno</td>
</tr>
</tbody>
</table>
Ex 1: `addrinfo` Returned by `getaddrinfo`

**DNS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Host</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsdi</td>
<td>IN A</td>
<td>206.62.226.35</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IN A</td>
<td>206.62.226.66</td>
<td></td>
</tr>
</tbody>
</table>

**/etc/services**

<table>
<thead>
<tr>
<th>Service</th>
<th>Domain</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain</td>
<td>53/tcp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>53/udp</td>
<td></td>
</tr>
</tbody>
</table>

```c
struct addrinfo hints, *res;

memset(0, &hints, sizeof(hints));

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

/* ... */

freeaddrinfo(res);
```
Ex 1: *addrinfo* Returned by *getaddrinfo* (2)

- 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 `SOCKET_STREAM`.
- One for the second IP address and a socket type of `SOCKET_DGRAM`.
Ex 1: addrinfo Returned by getaddrinfo (3)
Ex 1: `addrinfo` Returned by `getaddrinfo` (4)
Ex 1: addrinfo Returned by getaddrinfo (5)
getaddrinfo Ex 2: IPv4 and IPv6 Addresses

DNS

```
bsdi   IN   A       206.62.226.66
        IN   AAAA 2001::1f8d:2
```

```
struct addrinfo hints, *res;

memset(0, &hints, sizeof(hints));

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

/* ... */

freeaddrinfo(res);
```

/etc/services

domain 53/tcp
domain 53/udp
Ex 2: IPv4 and IPv6 Addresses (2)
**DNS**

/etc/services

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bsd</td>
<td>IN</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>206.62.226.66</td>
</tr>
<tr>
<td>IN</td>
<td>A</td>
<td>206.62.226.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>domain 53/tcp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>domain 53/udp</td>
</tr>
</tbody>
</table>

```c
struct addrinfo hints, *res;

memset(0, &hints, sizeof(hints));
hints.ai_flags = AI_V4MAPPED;
hints.ai_family = AF_INET6;
getaddrinfo("bsdi", "domain", &hints, &res);

/* ... */

freeaddrinfo(res);
```
Ex 3: IPv4-mapped (2)
DNS

/etc/services

bsdi IN A 206.62.226.35
IN AAAA 2001::1f8d:2 domain 53/tcp
domain 53/udp

struct addrinfo hints, *res;
memset(0, &hints, sizeof(hints);
hints.ai_flags = AI_ALL | AI_V4MAPPED;
hints.ai_family = AF_INET6;
getaddrinfo("bsdi", "domain", &hints, &res);

/* ... */

freeaddrinfo(res);
Ex 4: IPv4-mapped with IPv6 Addresses (2)

```c
addrinfo {}
ai_flags
AF_INET6
SOCK_DGRAM
0
16
NULL
ai_addr
ai_next

addrinfo {}
ai_flags
AF_INET6
SOCK_STREAM
0
28
ai_canonname
bsdi.kohala.com
ai_addr
ai_next

sockaddr_in6{
28AF_INET6 53
2001::1f8d:2
}

sockaddr_in6{
28AF_INET6 53
::FFFF:206.62.226.35
}

sockaddr_in6{
28AF_INET6 53
::FFFF:206.62.226.35
}
```
DNS

/getetc/services

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Type</th>
<th>Address</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsdI</td>
<td>A</td>
<td>206.62.226.35</td>
<td>53/tcp</td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td>2001::1f8d:2</td>
<td>53/udp</td>
</tr>
</tbody>
</table>

Client making getaddrinfo() call has no interface configured with IPv6

```c
struct addrinfo hints, *res;

memset(0, &hints, sizeof(hints));
hints.ai_flags = AI_ADDRCONFIG;
hints.ai_family = AF_INET6;
getaddrinfo("bsdI", "domain", &hints, &res);

/* ... */

freeaddrinfo(res);
```
If the node has no IPv6 source addresses configured, and `af` “hint” equals AF_INET6, and the node name being looked up has both AAAA and A records, then

- if only the `AI_ADDRCONFIG` “hint” is specified, the function returns a null pointer

```
res
    null
```
### DNS

<table>
<thead>
<tr>
<th>DNS</th>
<th>/etc/services</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsdi IN A</td>
<td>206.62.226.35</td>
</tr>
<tr>
<td>IN AAAA</td>
<td>2001::1f8d:2</td>
</tr>
</tbody>
</table>

Client making `getaddrinfo()` call has no interface configured with IPv6

```c
struct addrinfo hints, *res;

memset(0, &hints, sizeof(hints);
    hints.ai_flags = 0;
    hints.ai_family = AF_INET6;
    getaddrinfo("bsdi", "domain", &hints, &res);

    freeaddrinfo(res);
```
Ex 6: `addrinfo` Returned by `getaddrinfo` (2)

```
res

addrinfo {}
ai_flags
AF_INET6
SOCK_STREAM
0
28
ai_canonname
ai_addr
ai_next

addrinfo {}
ai_flags
AF_INET6
SOCK_DGRAM
0
28
ai_canonname
ai_addr
ai_next

sockaddr_in6 {}
28AF_INET6 53
2001::1f8d:2

sockaddr_in6 {}
28AF_INET6 53
2001::1f8d:2
```
s = -1;
for (res = res0; res; res = res->ai_next) {
    s = socket(res->ai_family, res->ai_socktype,
              res->ai_protocol);
    if (s < 0) {
        cause = "Error: socket";
        continue;
    }
    if (connect(s, res->ai_addr, res->ai_addrlen) < 0) {
        cause = "Error: connect";
        close(s);
        s = -1;
        continue;
    }
    cause = "Success";
    break; /* okay we got one */
...
s = -1;
for (res = res0; res; res = res->ai_next) {
    s = socket(res->ai_family, res->ai_socktype, 
                res->ai_protocol);
    if (s < 0) {
        cause = "Error: socket";
        continue;
    }
    if (connect(s, res->ai_addr, res->ai_addrlen) < 0) {
        cause = "Error: connect";
        close(s);
        s = -1;
        continue;
    }
    cause = "Success";
    break; /* okay we got one */
...
% nslookup
> set q=AAAA
name: bsdi.kohala.com
address: 2001::1f8d:2
Protocol Independent Programming

getaddrinfo Ex 7: Server Socket

/etc/services
  myservice 76/tcp
  myservice 76/udp

```c
struct addrinfo hints;
struct addrinfo *res;
char *myservice;
...
memset(&hints, 0, sizeof(hints)); /* set-up hints structure */
hints.ai_family = AF_UNSPEC;
hints.ai_flags = AI_PASSIVE;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo(NULL, myservice, &hints, &res);
```
Ex 7: Server Socket (2)
**getnameinfo()**

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

- **Input parameters**
  - sockaddr, addrlen
  - hostlen, servlen
  - flags

- **Output**
  - host, serv
  - error code
**getnameinfo()**

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

**FLAGS**
- NI_NOFQDN
- NI_NUMERICHOST
- NI_NAMEREQD
- NI_NUMERICSERV
- AI_DGRAM
<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NI_DGRAM</td>
<td>Datagram service</td>
</tr>
<tr>
<td>NI_NAMEREQD</td>
<td>Return an error if name cannot be resolved from addr</td>
</tr>
<tr>
<td>NI_NOFQDN</td>
<td>Return only hostname portion of FQDN</td>
</tr>
<tr>
<td>NI_NUMERICHOST</td>
<td>Return numeric string for hostname</td>
</tr>
<tr>
<td>NI_NUMERICSERV</td>
<td>Return numeric string for service name</td>
</tr>
</tbody>
</table>

Flags for `getnameinfo()`
Getting IP address as string for the accepted connection:

```c
struct sockaddr_storage clientAddr;
char   clientHost[NI_MAXHOST];
char   clientService[NI_MAXSERV];
size_t alen = sizeof(struct sockaddr_storage);

/*... */
connectedfd = accept(serverfd,
            (struct sockaddr *)&clientAddr,
            &alen);

err = getnameinfo((struct sockaddr *)&clientAddr, alen,
            clientHost, sizeof(clientHost),
            clientService, sizeof(clientService),
            NI_NUMERICHOST);
```

**INPUT ARGUMENTS**
Getting IP address as string for the accepted connection:

```c
struct sockaddr_storage clientAddr;
char clientHost[NI_MAXHOST];
char clientService[NI_MAXSERV];
size_t alen = sizeof(struct sockaddr_storage);

/*... */

connectedfd = accept(serverfd, 
    (struct sockaddr *)&clientAddr, 
    &alen);

err = getnameinfo( 
    (struct sockaddr *)&clientAddr, alen, 
    clientHost, sizeof(clientHost), 
    clientService, sizeof(clientService), 
    NI_NUMERICHOST);
```
Getting IP address as string for the accepted connection:

```c
struct sockaddr_storage clientAddr;
char clientHost[NI_MAXHOST];
char clientService[NI_MAXSERV];
size_t alen = sizeof(struct sockaddr_storage);
/*... */
connectedfd = accept(serverfd, (struct sockaddr *)&clientAddr, &alen);

err = getnameinfo((struct sockaddr *)&clientAddr, alen, clientHost, sizeof(clientHost), clientService, sizeof(clientService), NI_NUMERICHOST);
```

**ERROR**

**INPUT ARGUMENTS**

**OUTPUT**
Use of getnameinfo() for Numeric Hostname

The following code returns the numeric hostname, and service name, for given socket address. Observe that there is no hardcoded reference to particular address family.

```c
struct sockaddr *cliaddr;
/* input */
char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];
...
if (getnameinfo(cliaddr, cliaddr->sa_len, hbuf, sizeof(hbuf), sbuf,
               sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV) != 0) {
    error(1, "could not get numeric hostname");
    /*NOTREACHED*/
}
printf("Connection From host=%s, serv=%s\n", hbuf, sbuf);
```
The following code looks up the hostname corresponding to the socket address

```c
struct sockaddr *sa;
/* input */
char hbuf[NI_MAXHOST];
...
if (getnameinfo(res->ai_addr, res->ai_addrlen, hbuf,
    sizeof(hbuf), NULL, 0, NI_NAMEREQD) != 0)
{
    error(1, "could not resolve hostname");
    /*NOTREACHED*/
}
printf("host=%s\n", hbuf);
```
Presentation Format <-> Address Structure

- Change inet_aton()/inet_addr() by:
  - `inet_ntop()` (protocol dependent function)
  - `getaddrinfo()` (protocol independent function), remember to use freeaddrinfo

Address Structure -> Presentation Format

- Change inet_ntoa() by:
  - `inet_ntop()` (protocol dependent function)
  - `getnameinfo()` (protocol independent function) with flag NI_NUMERICHOST
Summary

- Use IP version independent structures:
  - sockaddr_storage

- Use IP version independent functions:
  - getaddrinfo()/getnameinfo()

- Not use inet_n top()/inet_pton()

- Not use gethostbyname()/gethostbyaddr()

- Iterated jobs for finding the working address:
  - Server:
    - listening packets addressed to a specific port.
  - Clients:
    - connecting to one of the server addresses.
References

Questions???

timothy.s.morizot@irs.gov