Introduction to IPv6 Programming

Rino Nucara
GARR
rino.nucara@garr.it
EuChinaGRID IPv6 Tutorial
Rome, January 16th, 2008
Version 1.3

Content

- The importance of being an IPv6 programmer
- IPv4/IPv6 Interoperability
- Introduction to IPv6 Programming for C
- Pratical exercises
- Introduction to IPv6 Programming for Java
- Pratical exercises
- Bibliography

The importance of being an *IPv6 aware* programmer

- The successful spread of IPv6 needs IPv6-compatible applications ☺
  - Network is ready
    - Most NRENs and Commercial ISPs offer IPv6 connectivity since a long time ago.
    - Already 800+ AS registered under the RIPE and IANA DBs .
  - Generalized lack IPv6 compatible applications in order to boost the migration.
- It is important for programmers to “think IPv6”:
  - To speed up IPv6 adoption
  - Avoid risk of rolling out non compatible IPv6 programs once IPv6 will take place

IPv4/IPv6 Interoperability - Issues

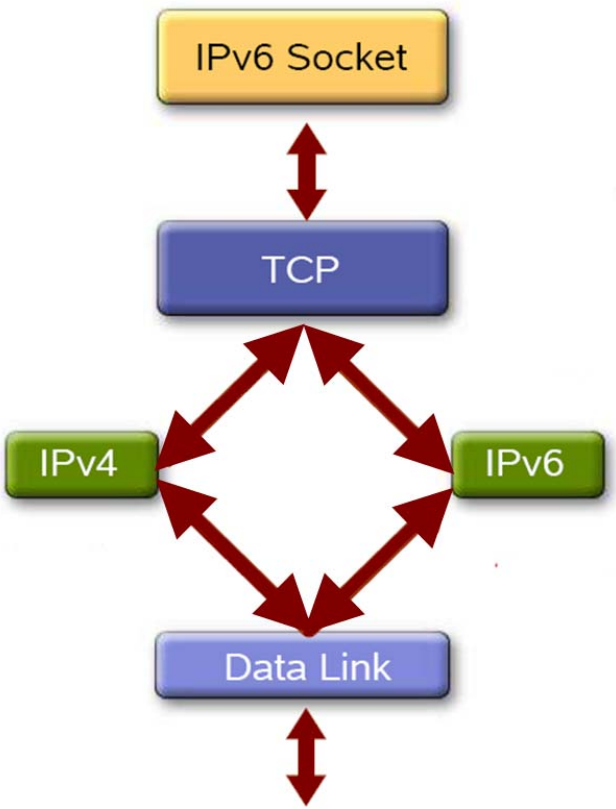
- For some (..2, 5 , 200.. :-)) years we will live in a dual IP protocol world.
- We will see progressive spread of IPv6 deployment and a very relevant residual usage of IPv4 all over the world
- Ways for interoperating between two incompatible protocols need to be identified.

IPv4/IPv6 Interoperability - Solutions

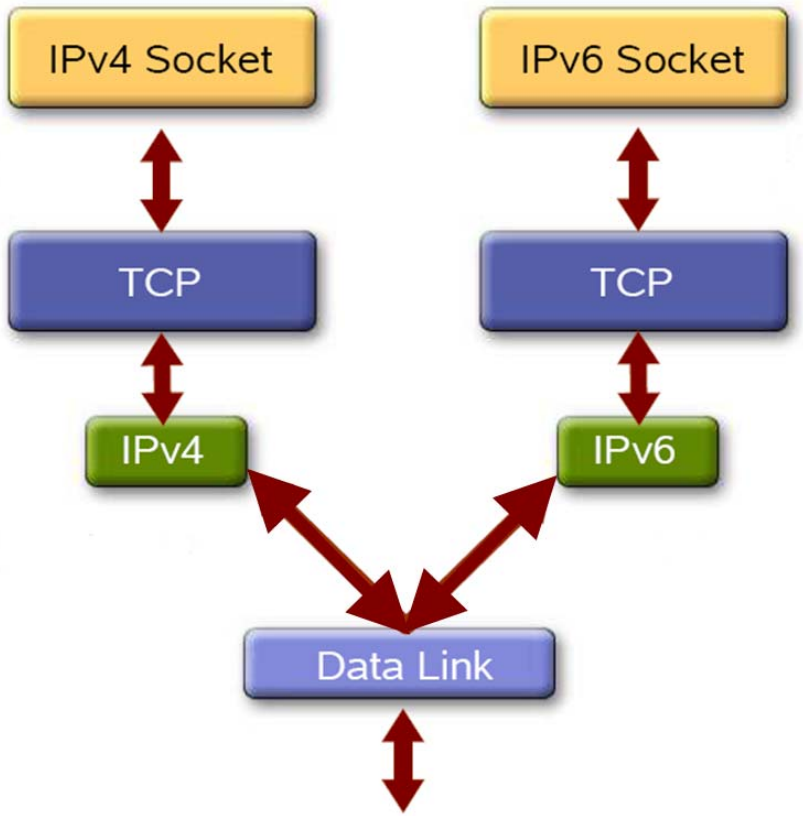
- To gradually introduce IPv6 to the Internet the IETF has carefully designed IPv6 migration.
- Interoperability is achived by the following two key technologies:
  - Dual Stack
  - Tunneling
- In literature (and in implementation) we also find “separated stack”.

Dual stack and separated stack

DUAL STACK

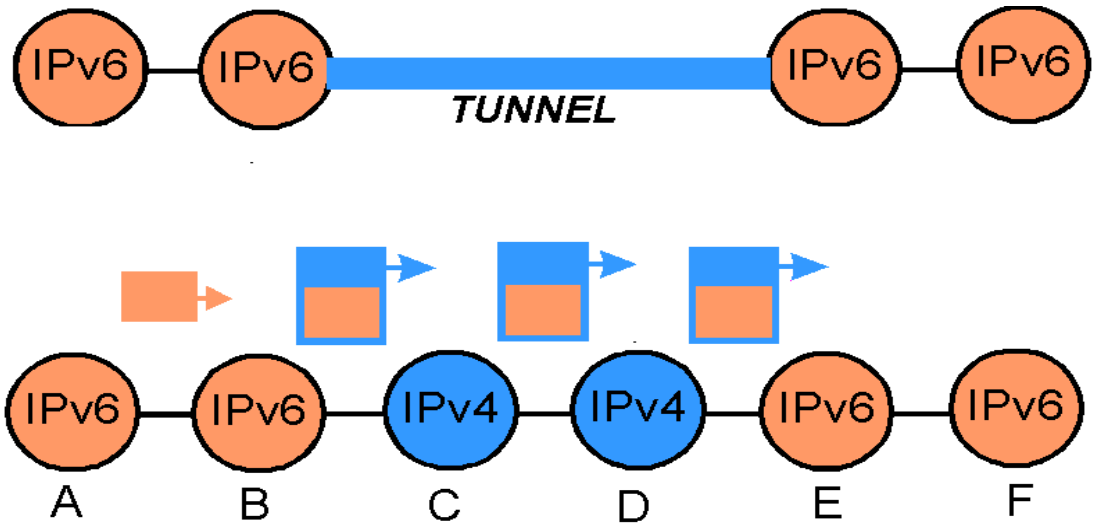


SEPARATED STACK

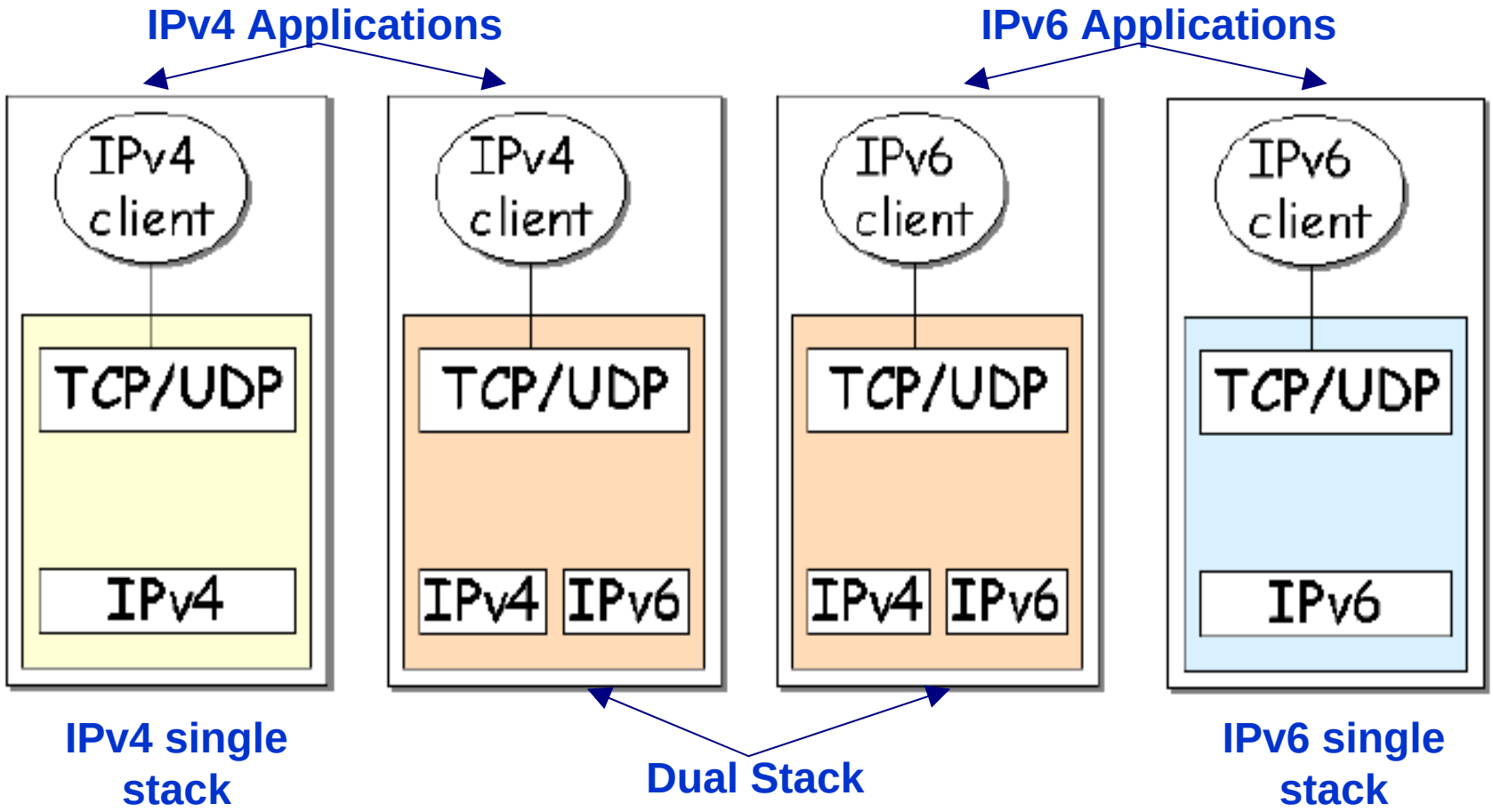


IPv6-over-IPv4 tunneling

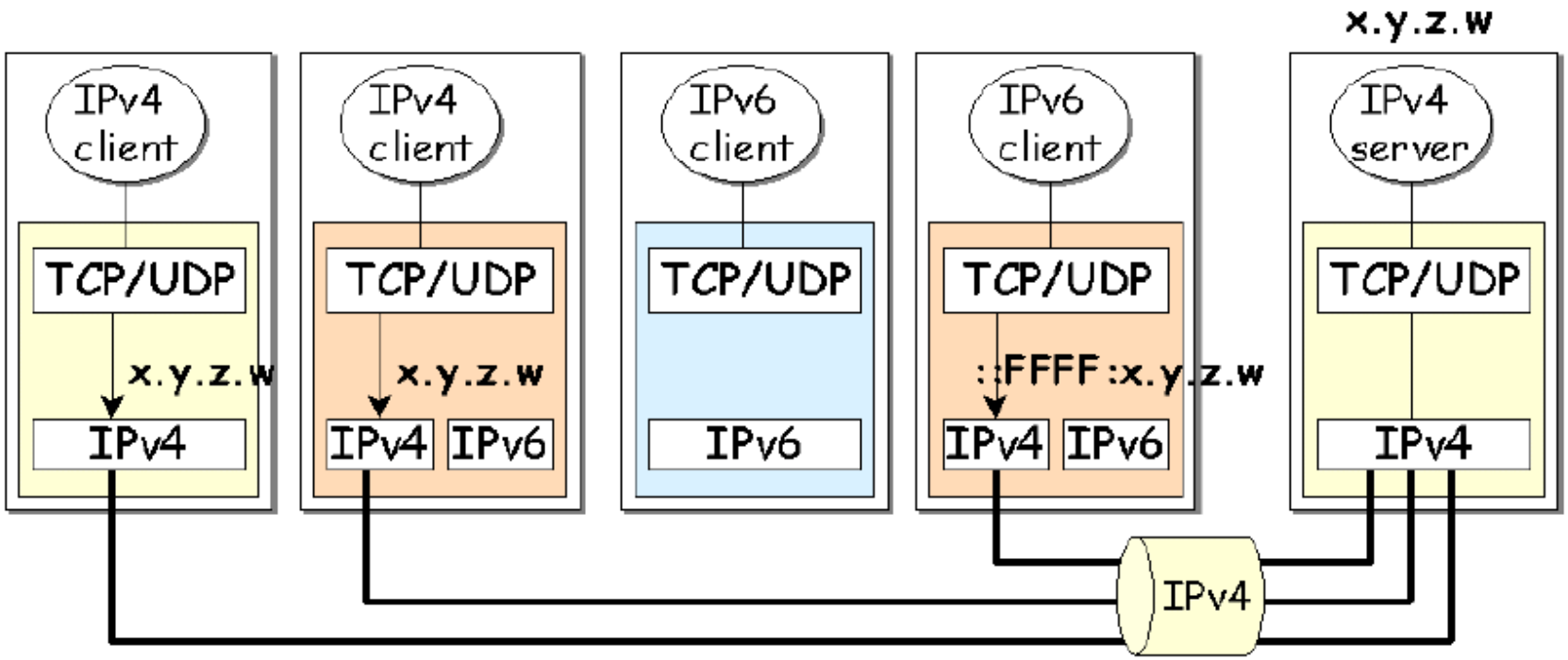
- Using IPv6 between two locations is only possible when all intermediate networks (ISPs) are IPv6 enabled
- Tunnelling is used to transport IPv6 packets through IPv6 incompatible networks.
- To solve this problem **RFC 2893** defines ways to encapsulate an IPv6 packet into an IPv4 packet.



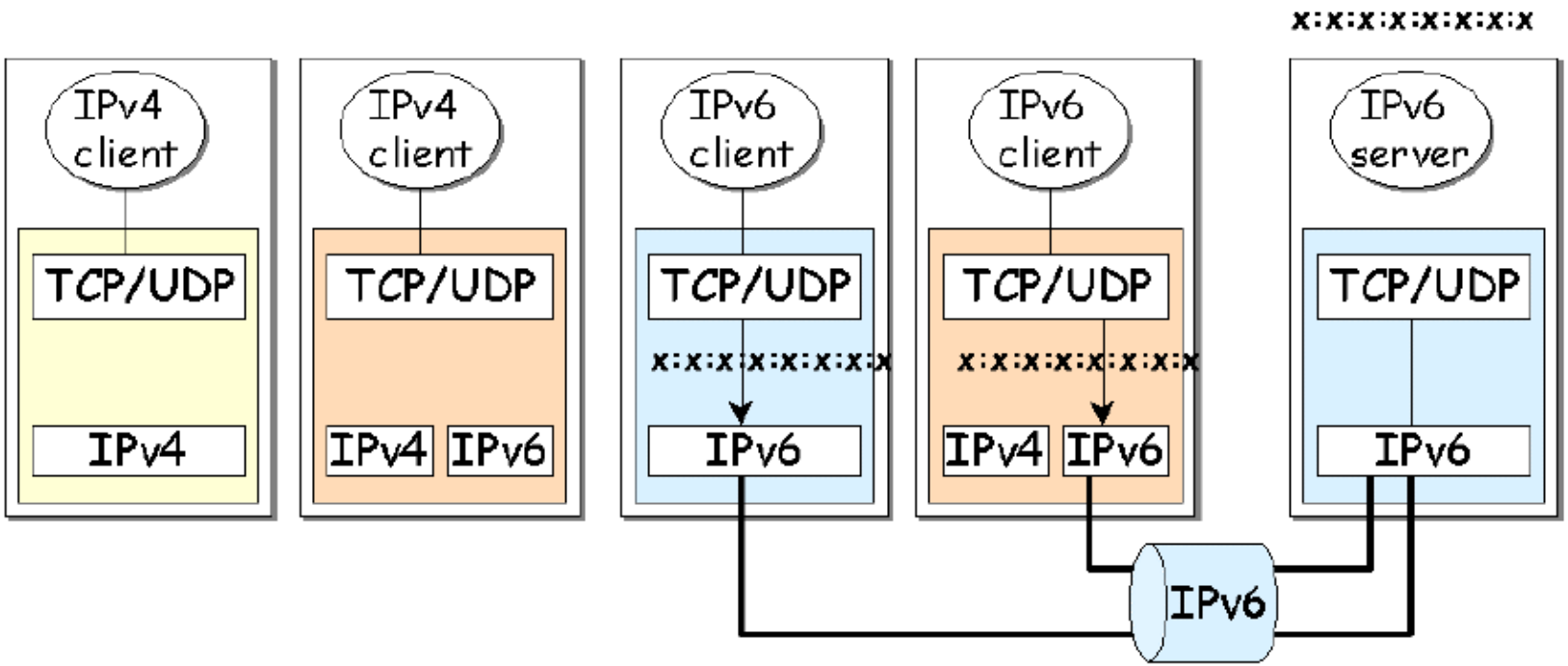
IPv4/IPv6 Interoperability examples: client types



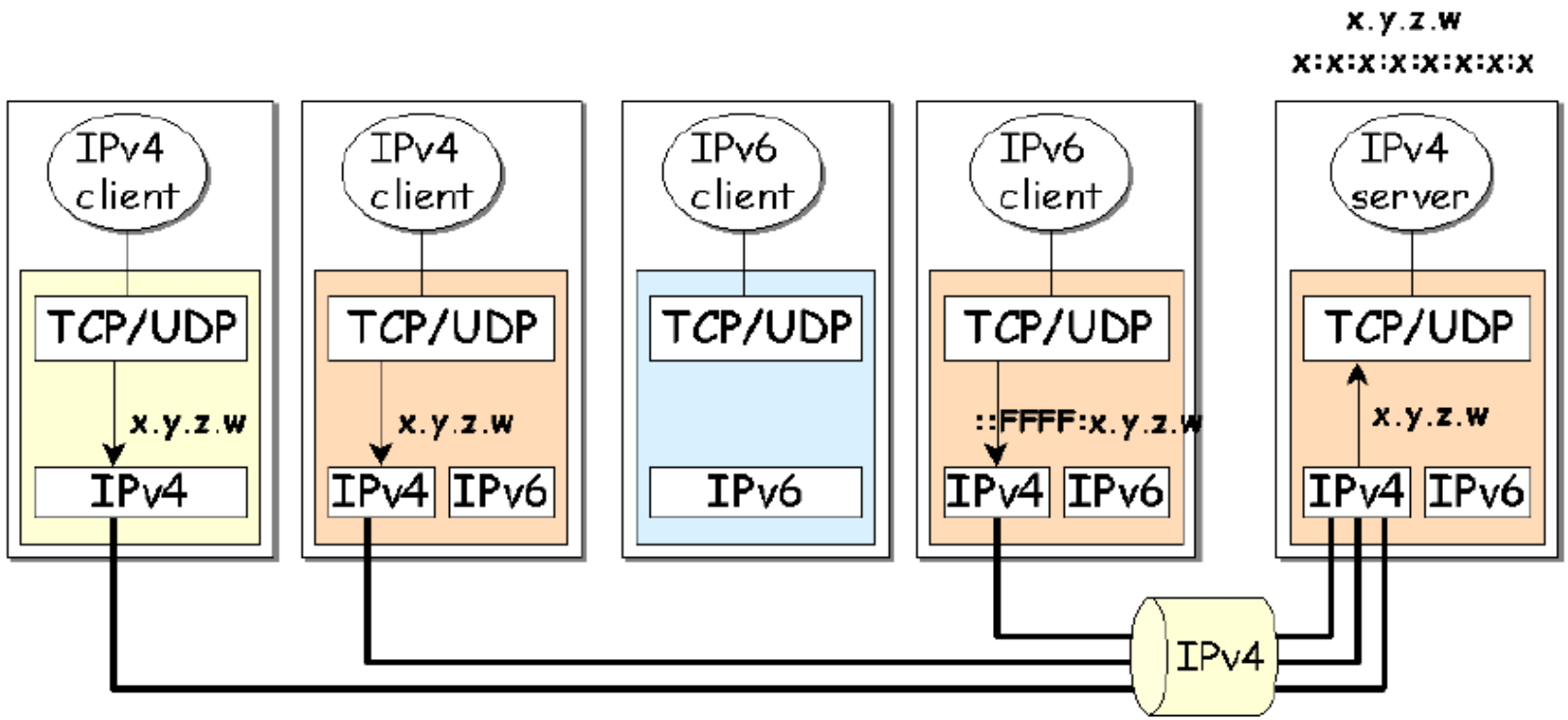
IPv6/IPv4 clients connecting to an IPv4 server at an IPv4-only node



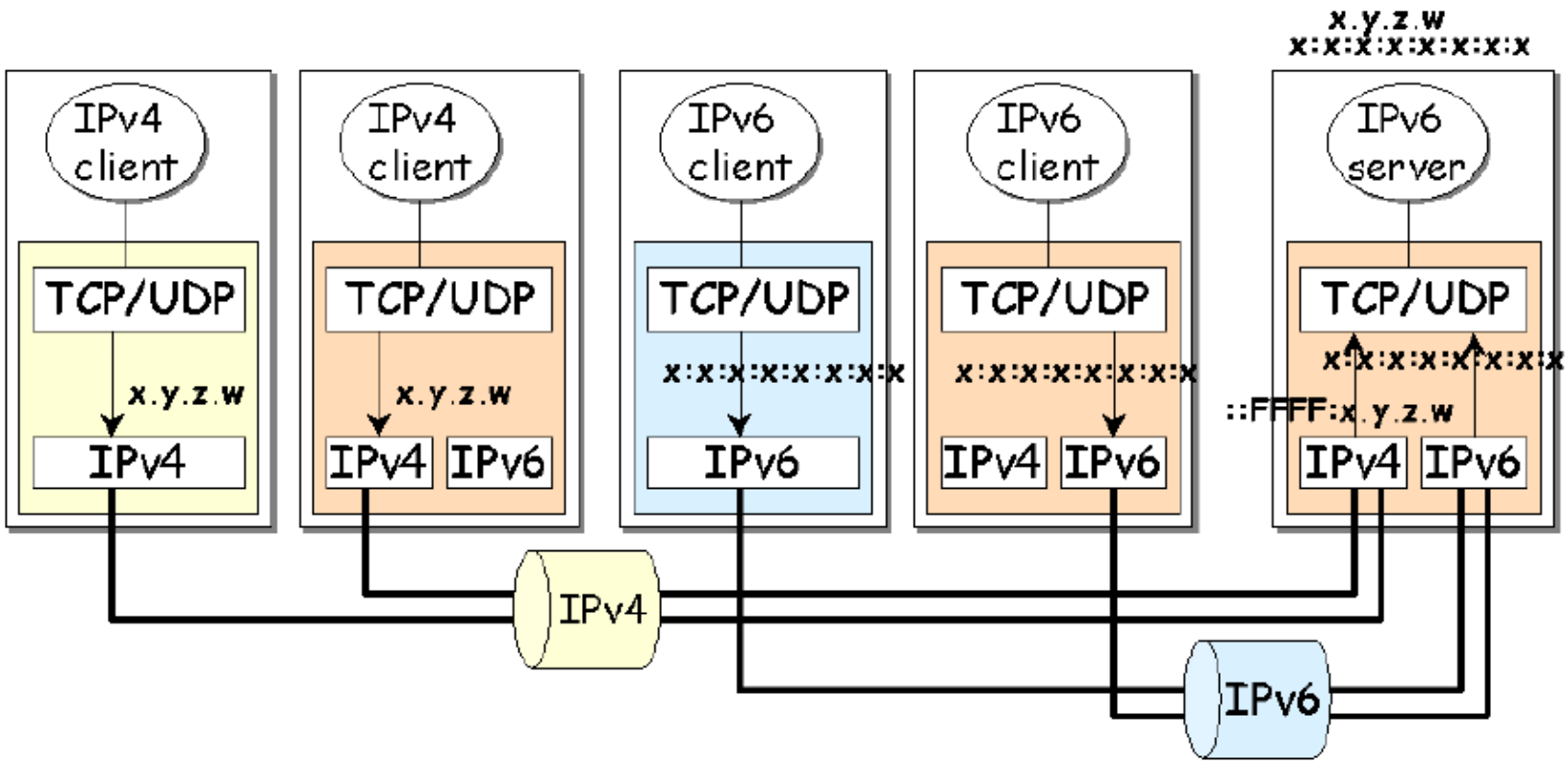
IPv6/IPv4 Clients connecting to an IPv6 server at IPv6-only node



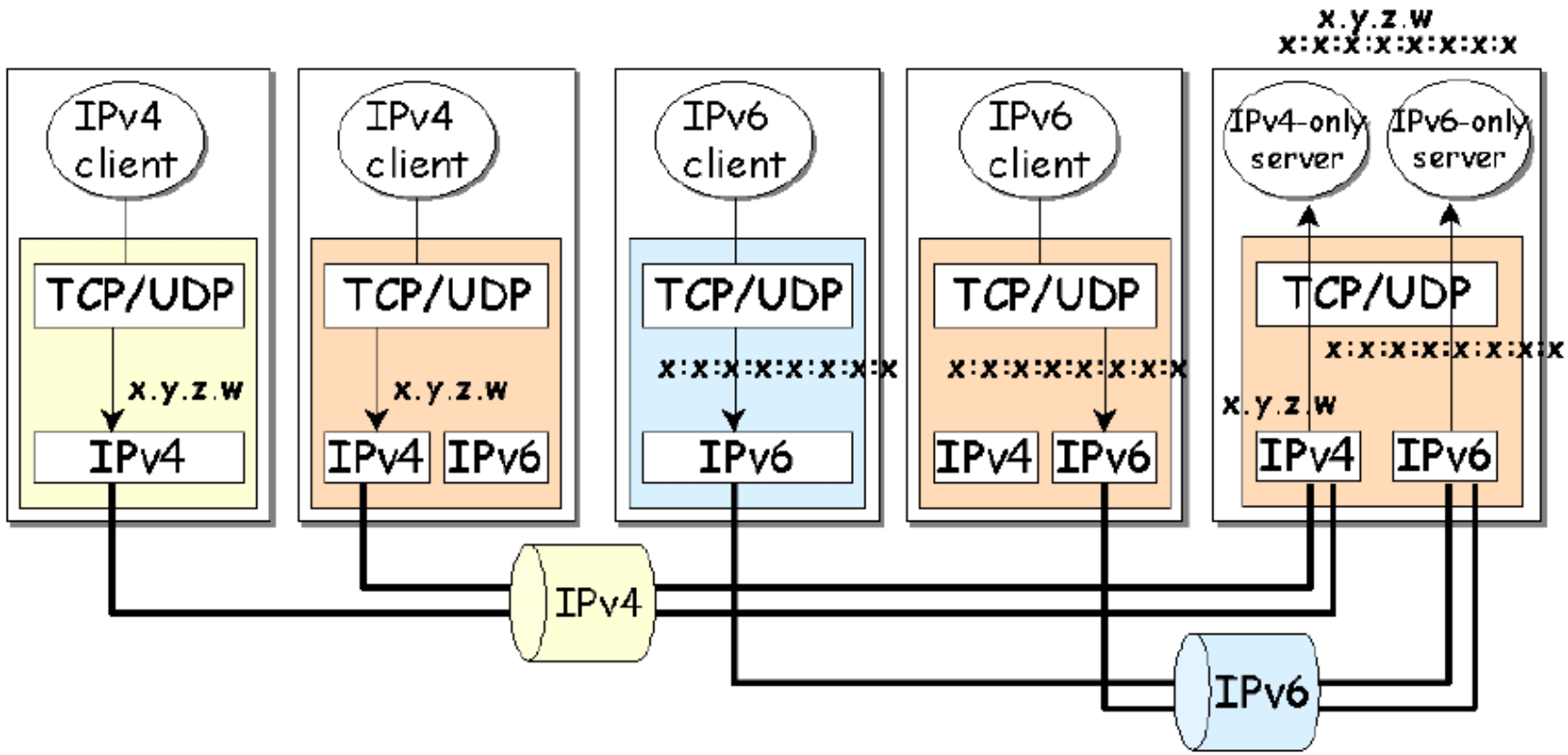
IPv6/IPv4 Clients connecting to an IPv4 server at dual stack node



IPv6/IPv4 Clients connecting to an IPv6 server at dual stack node



IPv6/IPv4 Clients connecting to a separated stack (IPv4-only and IPv6-only) server.



Interoperability Summary

Applications		IPv4 server		IPv6 server	
	Node Stack	IPv4	IPv4/IPv6	IPv6	IPv4/IPv6
IPv4 client	IPv4	IPv4	IPv4	X	IPv4
	IPv4/IPv6	IPv4	IPv4	X	IPv4
IPv6 client	IPv6	X	X	IPv6	IPv6
	IPv4/IPv6	IPv4	IPv4	IPv6	IPv6

Introduction to IPv6 Programming

In C

IPv6 API

- IETF standardized two sets of extensions: RFC 3493 and RFC 3542.
- **RFC 3493 Basic Socket Interface Extensions** for IPv6
  - Is the latest specification (the successor to RFC 2133 and RFC 2553. It is often referred to as “2553bis”)
  - Provides standard definitions for:
    - Core socket functions
    - Address data structures
    - Name-to-Address translation functions
    - Address conversion functions
- **RFC 3542 Advanced Sockets Application Program Interface** (API) for IPv6
  - Is the latest specification and is the successor to RFC2292 (it is often referred to as “2292bis”)
  - Defines interfaces for accessing special IPv6 information:
    - IPv6 header
    - extension headers
    - extend the capability of IPv6 raw socket

Interface Identification

RFC 3493 defines

- two functions mapping an interface name to an index and viceversa,

```
#include <net/if.h>
unsigned int  if_nametoindex(const char *ifname);
```

```
#include <net/if.h>
char  *if_indextoname(unsigned int ifindex, char *ifname);
```

- a third function returning all interface names and indexes,

```
struct if_nameindex {
 unsigned int   if_index;  /* 1, 2, ... */
 char          *if_name;   /* null terminated name: "le0", ... */
};

#include <net/if.h>
struct if_nameindex  *if_nameindex(void);
```

- a fourth function to return the dynamic memory allocated by the previous functions.

```
#include <net/if.h>
void  if_freenameindex(struct if_nameindex *ptr);
```

Example code: listing interfaces

```
$ ip addr
1: lo:[…]
2: eth0: […]
```

```
OUTPUT:
i    index   name
0      1      lo
1      2      eth0
```

```
#include <stdio.h>
#include <net/if.h>

int main(int argc, char *argv[])
{

int i;
struct if_nameindex *ifs = if_nameindex();
if (ifs == NULL) { perror("could not run if_nameindex");return 1;}

for (i=0; (ifs[i].if_index != 0)&&(ifs[i].if_name != NULL); i++)
   {
    printf("%3d  %3d  %s\n", i, ifs[i].if_index,ifs[i].if_name);
   }

if_freenameindex(ifs);

}
```

new address family name

A new address family name, AF_INET6 was defined for IPv6; the related protocol family is PF_INET6, and names belonging to it are defined as follow:

```
#define   AF_INET6   10
#define   PF_INET6   AF_INET6
```

IPv4 source code:

```
socket(PF_INET,SOCK_STREAM,0); /* TCP socket */
socket(PF_INET,SOCK_DGRAM,0); /* UDP socket */
```

IPv6 source code:

```
socket(PF_INET6,SOCK_STREAM,0); /* TCP socket */
socket(PF_INET6,SOCK_DGRAM,0); /* UDP socket */
```

Address structures

- IPv4
  - *struct sockaddr_in*
  - *struct sockaddr*
- IPv6
  - *struct sockaddr_in6*
- IPv4/IPv6/…
  - *struct sockaddr_storage*

Address Data Structure: Struct sockaddr

```
struct sockaddr {
    sa_family_t   sa_family;    // address family, AF_xxx
    char          sa_data[14];  // 14 bytes of protocol address
};
```

Functions provided by socket API use socket address structures to determine the communication service access point.

Since different protocols can handle socket functions, a generic socket address structure called *sockaddr* is used as argument to these functions

From an application programmer's point of view, **the only use** of these generic socket address structures is **to cast pointers to protocol-specific structures**.

sockaddr structure (2+14 bytes) holds socket address information for many types of sockets.
sa_family represents address family.
sa_data contains data about address.

Address Data Structure: Struct sockaddr_in (1/2)

```
struct in_addr {
    uint32_t s_addr; // 32-bit IPv4 address (4 bytes)
                     // network byte ordered
};

struct sockaddr_in {
    sa_family_t    sin_family;  // Address family (2 bytes)
    in_port_t      sin_port;    // Port number (2 bytes)
    struct in_addr sin_addr;    // Internet address (4 bytes)
    char           sin_zero[8]; // Empty (for padding) (8 bytes)
}
```

sockaddr_in is a parallel structure to deal with struct *sockaddr* for IPv4 addresses.

sin_port contains the port number and must be in Network Byte Order.

sin_family corresponds to sa_family (in a sockaddr structure) and contains the type of address family (AF_INET for IPv4). As sin_port also sin_family must be in Network Byte Order.

sin_addr represents Internet address (for IPv4).

Address Data Structure: Struct sockaddr_in (2/2)

```
struct in_addr {
    uint32_t s_addr; // 32-bit IPv4 address (4 bytes)
                     // network byte ordered
};

struct sockaddr_in {
    sa_family_t    sin_family;  // Address family (2 bytes)
    in_port_t      sin_port;    // Port number (2 bytes)
    struct in_addr sin_addr;    // Internet address (4 bytes)
    char           sin_zero[8]; // Empty (for padding) (8 bytes)
}
```

sin_zero is included to pad the structure to the length of a struct sockaddr and should be set to all zero using the *bzero()* or *memset()* functions.

Casting

```
struct sockaddr_in addrIPv4;

/* fill in addrIPv4{} */

bind (sockfd, (struct sockaddr *) & addrIPv4, sizeof(addrIPv4) );
```

A pointer to a struct sockaddr_in (16 bytes) can be casted to a pointer to a sockaddr struct (16 bytes) and vice versa. So even though socket() wants a struct sockaddr * it is necessary to cast sockaddr while passing to socket function.

Functions provided by socket API use socket address structures to determine the communication service access point.

As stated before, the generic socket address structure *sockaddr* is used as argument to these functions (for any of the supported communication protocol families).

Address Data Structure: Sockaddr_in6 (1/3)

```
struct in6_addr {
          uint8_t  s6_addr[16];  // 128-bit IPv6 address (N.B.O.)
};
struct sockaddr_in6 {
  sa_family_t     sin6_family;   //AF_INET6
  in_port_t       sin6_port;     //transport layer port # (N.B.O.)
  uint32_t        sin6_flowinfo; //IPv6 flow information (N.B.O.)
  struct in6_addr sin6_addr;    // IPv6 address
  uint32_t        sin6_scope_id; //set of interfaces for a scope
}
```

sockaddr_in6 structure holds IPv6 addresses and is defined as a result of including the <**netinet/in.h**> header.

sin6_family overlays the sa_family field when the buffer is cast to a sockaddr data structure. The value of this field must be AF_INET6. (2Byte)

sin6_port contains the 16-bit UDP or TCP port number. This field is used in the same way as the sin_port field of the sockaddr_in structure. The port number is stored in **network byte order.** (2Byte)

Address Data Structure: Sockaddr_in6 (2/3)

```
struct in6_addr {
          uint8_t  s6_addr[16];  // 128-bit IPv6 address (N.B.O.)
};
struct sockaddr_in6 {
  sa_family_t     sin6_family;   //AF_INET6
  in_port_t       sin6_port;     //transport layer port # (N.B.O.)
  uint32_t        sin6_flowinfo; //IPv6 flow information (N.B.O.)
  struct in6_addr sin6_addr;    // IPv6 address
  uint32_t        sin6_scope_id; //set of interfaces for a scope
}
```

sin6_flowinfo is a 32-bit field intended to contain flow-related information. The exact way this field is mapped to or from a packet is not currently specified. Until its exact use will be specified, applications should set this field to zero when constructing a sockaddr_in6, and ignore this field in a sockaddr_in6 structure constructed by the system. (4Byte)

sin6_addr is a single in6_addr structure. This field holds one 128-bit IPv6 address. The address is stored in **network byte order**. (16Byte)

Address Data Structure: Sockaddr_in6 (3/3)

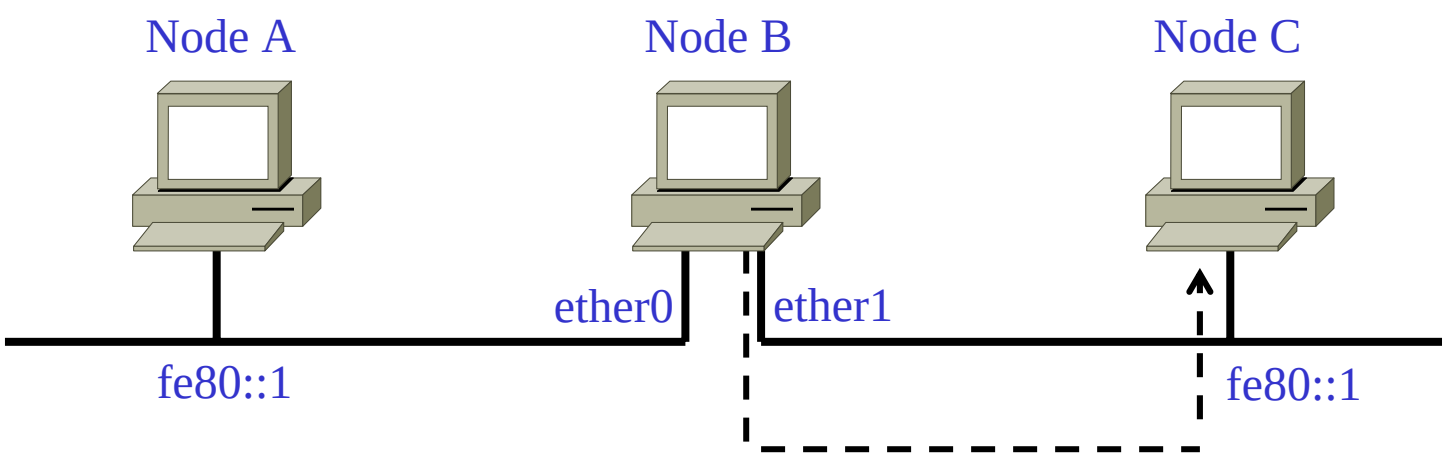
```
struct in6_addr {
          uint8_t  s6_addr[16];  // 128-bit IPv6 address (N.B.O.)
};
struct sockaddr_in6 {
  sa_family_t     sin6_family;   //AF_INET6
  in_port_t       sin6_port;     //transport layer port # (N.B.O.)
  uint32_t        sin6_flowinfo; //IPv6 flow information (N.B.O.)
  struct in6_addr sin6_addr;    // IPv6 address
  uint32_t        sin6_scope_id; //set of interfaces for a scope
}
```

sin6_scope_id is a 32-bit integer that identifies a set of interfaces as appropriate for the scope of the address carried in the sin6_addr field. The mapping of sin6_scope_id to an interface or set of interfaces is left to implementation and future specifications on the subject of scoped addresses. (4Byte)

RFC 3493 did not define the usage of the sin6_scope_id field because at the time there was some debate about how to use that field.
The intent was to publish a separate specification to define its usage, but that has not happened.

Address Data Structure:Sockaddr_in6 (sin6_scope_id)

To communicate with node A or node C, node B has to disambiguate between them with a link-local address you need to specify the scope identification.



String representation of a scoped IPv6 address is augmented with scope identifier after % sign (es. Fe::1%ether1).
NOTE! Scope identification string is implementation-dependent.

Address Data Structure: Sockaddr_in6 in BSD

```
struct in6_addr {
        uint8_t  s6_addr[16];  // 128-bit IPv6 address (N.B.O.)
};
struct sockaddr_in6 {
  unit8_t       sin6_len;     //length of this struct
  sa_family_t   sin6_family;   //AF_INET6 (8bit)
  in_port_t     sin6_port;     //transport layer port # (N.B.O.)
  uint32_t      sin6_flowinfo; //IPv6 flow information (N.B.O.)
  struct in6_addr sin6_addr;    // IPv6 address
  uint32_t      sin6_scope_id; //set of interfaces for a scope
}
```

The 4.4BSD release includes a small, but incompatible change to the socket interface.

The "sa_family" field of the sockaddr data structure was changed **from a 16-bit value to an 8-bit** value, and the **space saved used to hold a length field, named "sa_len"**.

The sockaddr_in6 data structure given in the previous section cannot be correctly casted into the newer sockaddr data structure.

For this reason, the following alternative IPv6 address data structure is provided to be used on systems based on 4.4BSD. It is defined as a result of including the <netinet/in.h> header.

Address Data Structure: Sockaddr_in6 in BSD

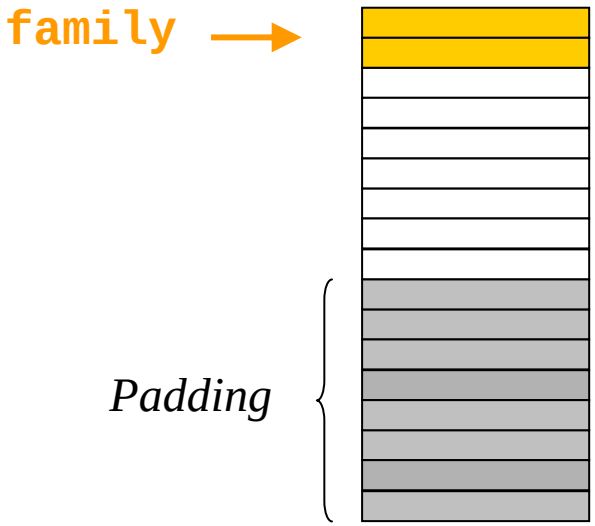
For dealing this incompatible difference should be userfull to use Preprocessor Directive:

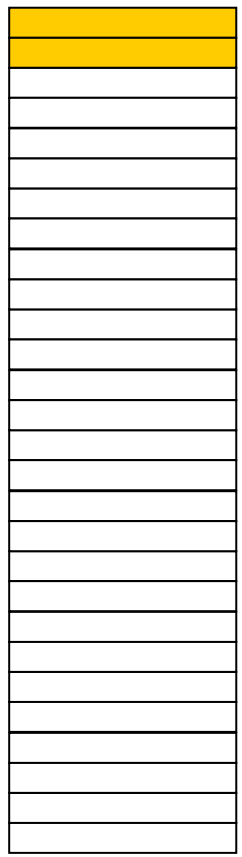
```
[…]
struct sockaddr_in6 sin6;

memset(&sin6, 0, sizeof(sin6));

#ifdef __BSD__
sin6.sin6_len = sizeof(sin6);
#endif

sin6.sin6_family = AF_INET6;
sin6.sin6_port = htons(5002);

[…]
```

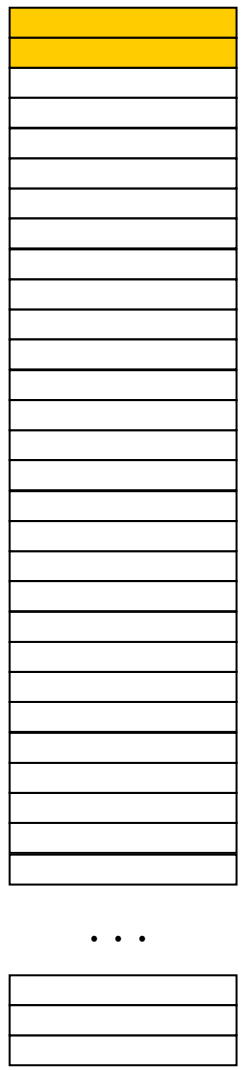
16 byte
sockaddr_in

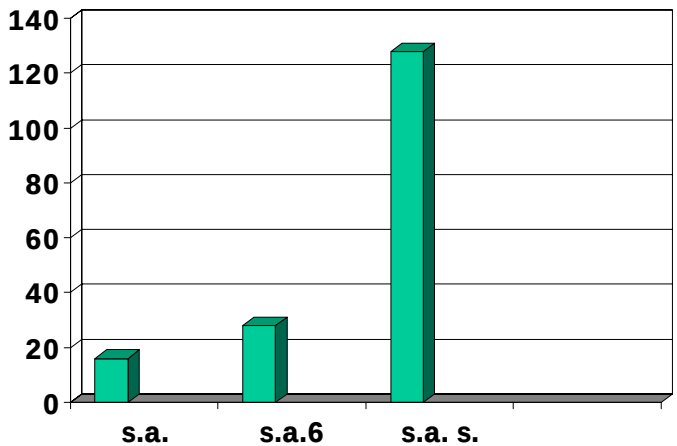
28 byte
sockaddr_in6

128 byte
sockaddr_storage









Problem code example

```
Struct sockaddr sa;
Struct sockaddr_in6 *sin6=(struct sockaddr *) &sa;
Memeset (sin6, 0, sizeof(*sin6));
```

In this code, the *memset()* operation will overwrite the memory region immediately following the space that was allocated for the *sockaddr{}* structure.

It is therefore important to use a **new address structure: *sockaddr_storage{}*** as a socket address placeholder throughout the code in order to avoid introducing this kind of programming bug.

Address Data Structure:sockaddr_storage (1/2)

In order to write portable and multiprotocol applications, another data structure is defined: the new ***sockadd_storage***.

This function is designed to store all protocol specific address structures with the right dimension and alignment.

Hence, portable applications should use the sockaddr_storage structure to store their addresses, both IPv4 or IPv6 ones.

This new structure **hides the specific socket address structure that the application is using.**

Address Data Structure:sockaddr_storage (2/2)

```
/*Desired design of maximum size and alignment */
#define _SS_MAXSIZE    128  /* Implementation specific max size */
#define _SS_ALIGNSIZE  (sizeof (int64_t)) /* Implementation specific
desired alignment */

/*Definitions used for sockaddr_storage structure paddings design.*/
#define _SS_PAD1SIZE   (_SS_ALIGNSIZE - sizeof (sa_family_t))
#define _SS_PAD2SIZE   (_SS_MAXSIZE - (sizeof (sa_family_t) +
                              _SS_PAD1SIZE + _SS_ALIGNSIZE))
struct sockaddr_storage {
    sa_family_t  ss_family;     /* address family */
    /* Following fields are implementation specific */
    char      __ss_pad1[_SS_PAD1SIZE];
              /* 6 byte pad, this is to make implementation
              /* specific pad up to alignment field that */
              /* follows explicit in the data structure */
    int64_t   __ss_align;     /* field to force desired structure */
               /* storage alignment */
    char      __ss_pad2[_SS_PAD2SIZE];
              /* 112 byte pad to achieve desired size, */
              /* _SS_MAXSIZE value minus size of ss_family */
              /* __ss_pad1, __ss_align fields is 112 */
};
```

Pass addresses

Socketcalls where a socket address structure is provided from an application to the kernel

- IPv4:

```
struct sockaddr_in addr;
socklen_t addrlen = sizeof(addr);
   // fill addr structure using an IPv4 address
   // before calling socket funtion
bind(sockfd,(struct sockaddr *)&addr, addrlen);
```

- IPv6:

```
struct sockaddr_in6 addr;
socklen_t addrlen = sizeof(addr);
  //fill addr structure using an IPv6 address
  //before calling socket function
bind(sockfd,(struct sockaddr *)&addr, addrlen);
```

- IPv4 and IPv6:

```
struct sockaddr_storage addr;
socklen_t addrlen=sizeof(addr);
   // fill addr structure using an IPv4/IPv6 address
   // and fill addrlen before calling socket function
bind(sockfd,(struct sockaddr *)&addr, addrlen);a
```

Get adresses

Socket calls where a socket address structure is provided from the kernel to an application

- IPv4:

```
struct sockaddr_in addr;
socklen_t addrlen = sizeof(addr);
accept(sockfd,(struct sockaddr *)&addr, &addrlen);
// addr structure contains an IPv4 address
```

- IPv6:

```
struct sockaddr_in6 addr;
socklen_t addrlen = sizeof(addr);
accept(sockfd,(struct sockaddr *)&addr, &addrlen);
// addr structure contains an IPv6 address
```

- IPv4 and IPv6:

```
struct sockaddr_storage addr;
socklen_t addrlen = sizeof(addr);
accept(sockfd,(struct sockaddr *)&addr, &addrlen);
// addr structure contains an IPv4/IPv6 address
// addrlen contains the size of the addr structure returned
```

IPv6 Wildcard Address

1
```
#include <netinet/in.h>:
extern const struct in6_addr in6addr_any;
```

2
```
#include <netinet/in.h>.
#define IN6ADDR_ANY_INIT {{{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 }}}
```

IPv6 Loopback Address

1
```
<netinet/in.h>
extern const struct in6_addr in6addr_loopback;
```

2
```
<netinet/in.h>
#define IN6ADDR_LOOPBACK_INIT {{{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1}}}
```

IPv6 Wildcard Address (1/2)

**the IPv6 Wildcard Address is provided in two forms:
a global variable and a symbolic constant.**

1

```
#include <netinet/in.h>:
extern const struct in6_addr in6addr_any;
```

For example, to bind a socket to port number 23, but let the system select the source address, an application could use the following code:

```
struct sockaddr_in6 sin6;
. . .
sin6.sin6_family = AF_INET6;
sin6.sin6_flowinfo = 0;
sin6.sin6_port = htons(23);
sin6.sin6_addr = in6addr_any;  /* structure assignment */
. . .
if (bind(s, (struct sockaddr *) &sin6, sizeof(sin6)) == -1)
. . .
```

IPv6 Wildcard Address (2/2)

Another option is a symbolic constant named IN6ADDR_ANY_INIT.

2

```
#include <netinet/in.h>.
#define IN6ADDR_ANY_INIT {{{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 }}}
```

This constant can be used to initialize an in6_addr structure:

```
struct in6_addr anyaddr = IN6ADDR_ANY_INIT;
```

Note that this constant can be used **ONLY at declaration time**. It can not be used to assign a previously declared in6_addr structure. For example, the following code will not work:

```
/* This is the WRONG way to assign an unspecified address */
      struct sockaddr_in6 sin6;
        . . .
      sin6.sin6_addr = IN6ADDR_ANY_INIT; /* will NOT compile */
```

IPv6 Loopback Address (1/2)

the IPv6 loopback address is provided in two forms -- a global variable and a symbolic constant.

1

```
<netinet/in.h>
extern const struct in6_addr in6addr_loopback;
```

Applications use in6addr_loopback as they would use INADDR_LOOPBACK in IPv4 applications. For example, to open a TCP connection to the local telnet server, an application could use the following code:

```
struct sockaddr_in6 sin6;
 . . .
 sin6.sin6_family = AF_INET6;
 sin6.sin6_flowinfo = 0;
 sin6.sin6_port = htons(23);
 sin6.sin6_addr = in6addr_loopback;  /* structure assignment */
 . . .
 if (connect(s, (struct sockaddr *) &sin6, sizeof(sin6)) == -1)
 . . .
```

IPv6 Loopback Address (2/2)

Second way is a symbolic constant named IN6ADDR_LOOPBACK_INIT:

2
```
<netinet/in.h>
#define IN6ADDR_LOOPBACK_INIT {{{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1}}}
```

It can be used at **declaration time ONLY**.
for example:

```
struct in6_addr loopbackaddr = IN6ADDR_LOOPBACK_INIT;
```

Like IN6ADDR_ANY_INIT, this constant cannot be used in an assignment to a previously declared IPv6 address variable.

Socket Options

A number of new socket options are defined for IPv6:

```
IPV6_UNICAST_HOPS
IPV6_MULTICAST_IF
IPV6_MULTICAST_HOPS
IPV6_MULTICAST_LOOP
IPV6_JOIN_GROUP
IPV6_LEAVE_GROUP
IPV6_V6ONLY
```

All of these new options are at the IPPROTO_IPV6 level (specifies the code in the system to interpret the option).

The declaration for IPPROTO_IPV6 is obtained by including the header <netinet/in.h>.

Socket Options: Unicast Hop Limit

IPV6_UNICAST_HOPS option controls the hop limit used in outgoing unicast IPv6 packets.

```
int  hoplimit = 10;
if (
   setsockopt(s, IPPROTO_IPV6, IPV6_UNICAST_HOPS,
   (char *) &hoplimit, sizeof(hoplimit))
    == -1) perror("setsockopt IPV6_UNICAST_HOPS");
```

```
int  hoplimit;
socklen_t  len = sizeof(hoplimit);

if (
    getsockopt(s, IPPROTO_IPV6, IPV6_UNICAST_HOPS,
    (char *) &hoplimit, &len)
     == -1) perror("getsockopt IPV6_UNICAST_HOPS");
else
   printf("Using %d for hop limit.\n", hoplimit);
```

IPV6_V6ONLY

AF_INET6 sockets may be used for both IPv4 and IPv6 comunication.

- the socket can be used to send and receive IPv6 packets only.

- takes an int value (but is a boolean option).

-by default is turned off.

```
int on = 1;

if(setsockopt(s,IPPROTO_IPV6,IPV6_V6ONLY,(char *)&on,sizeof(on))==-1)
      perror("setsockopt IPV6_V6ONLY");
    else
      printf("IPV6_V6ONLY set\n");
```

An example use of this option is to allow two versions of the same server process to run on the same port, one providing service over IPv6, the other providing the same service over IPv4 (separated Stack).

IPV6_V6ONLY EXAMPLE

```
struct sockaddr_in6 sin6, sin6_accept;
socklen_t sin6_len; int s0, s;  int on; char hbuf[NI_MAXHOST];

memset(&sin6,0,sizeof(sin6));
sin6.sin6_family=AF_INET6; sin6.sin6_len=sizeof(sin6);
sin6.sin6_port=htons(5001);

s0=socket(AF_INET6,SOCK_STREAM,IPPROTO_TCP);
on=1; setsockopt=(s0,SOL_SOCKET, SO_REUSEADDR, &on,sizeof(on));

#ifdef USE_IPV6_V6ONLY
  on=1;
  setsockopt(s0,IPPROTO_IPV6, IPV6_V6ONLY,&on,sizeof(on));
#endif

bind(s0,(const struct sockaddr *)&sin6, sizeof(sin6));
listen(s0,1);
while(1){
   sin6_len=sizeof(sin6_accept);
   s=accept(s0,(struct sockaddr *)&sin6_accept, &sin6_len);
   getnameinfo((struct sockaddr *)&sin6_accept, sin6_len, hbuf,
       sizeof(hbuf), NULL, 0, NI_NUMERICHOST);
   printf("accept a connection from %s\n",hbuf);
   close(s);
}
```

Running example code

Without using **USE_IPV6_V6ONLY**:

```
telnet ::1 5001
```
→ Accept a connection from ::1

```
telnet 127.0.0.1 5001
```
→ Accept a connection from ::ffff:127.0.0.1

Using **USE_IPV6_V6ONLY**

```
telnet ::1 5001
```
→ Accept a connection from ::1

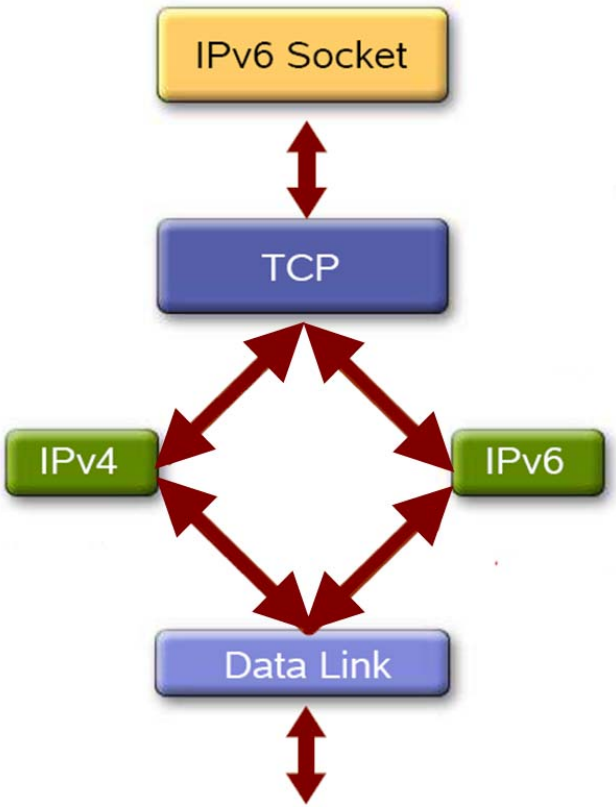
```
telnet 127.0.0.1 5001
```
→

```
Trying 127.0.0.1 …
telnet: connection to address 127.0.0.1: Connection refused
```

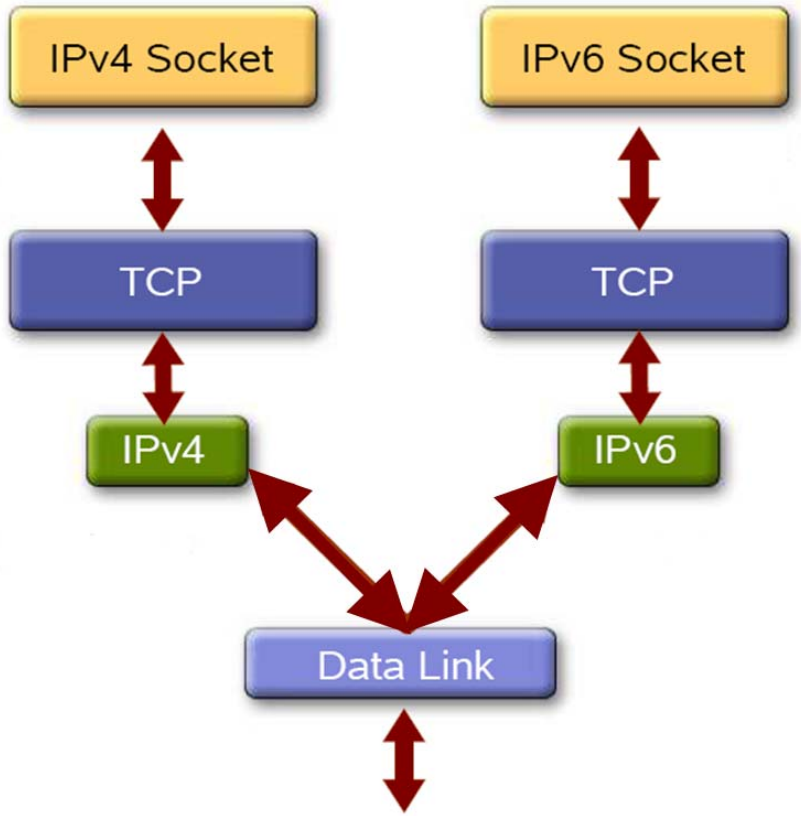
Dual stack and separated stack implementation

Following there are implementation examples of dual stack and separated stack in C

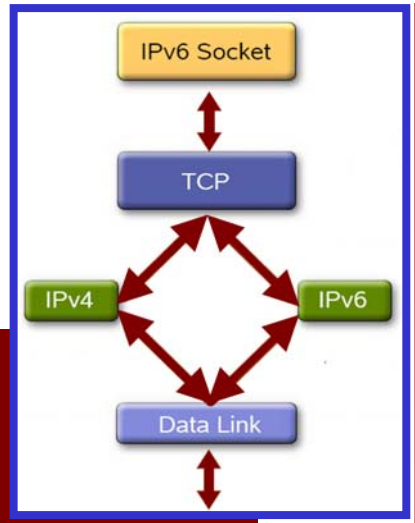
DUAL STACK



SEPARATED STACK



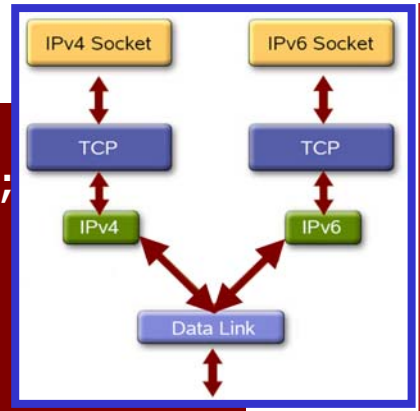
Server (Dual Stacks)




```
int ServSock, csock;
struct sockaddr addr, from;
...
ServSock = socket(AF_INET6, SOCK_STREAM,PF_INET6);

bind(ServSock, &addr, sizeof(addr));
do {
        csock = accept(ServSocket, &from,sizeof(from));
        doClientStuff(csock);
} while (!finished);
```

```
$ netstat -nap |grep 5002
tcp6   0     0 :::5002         :::*          LISTEN     3720/a.out
```

Server (Separate Stacks) 1/4

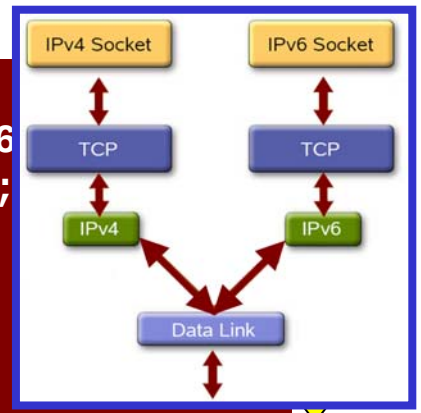



```
ADDRINFO AI0, AI1;
ServSock[0] = socket(AF_INET6, SOCK_STREAM, PF_INET6);
ServSock[1] = socket(AF_INET, SOCK_STREAM, PF_INET);
...
bind(ServSock[0], AI0->ai_addr, AI0->ai_addrlen);
bind(ServSock[1], AI1->ai_addr, AI1->ai_addrlen);
...
select(2, &SockSet, 0, 0, 0);
if (FD_ISSET(ServSocket[0], &SockSet)) {
    // IPv6 connection
    csock = accept(ServSocket[0], (LPSOCKADDR)&From,FromLen);
    ...
}
if (FD_ISSET(ServSocket[1], &SockSet)) {
    // IPv4 connection
    csock = accept(ServSocket[1], (LPSOCKADDR)&From, FromLen);
    ...
}
```

```
$ netstat -nap |grep 5002

tcp    0     0 0.0.0.0:5002   0.0.0.0:*    LISTEN     3720/a.out
tcp6   0     0 :::5002        :::*         LISTEN     3720/a.out
```

Server (Separate Stacks) 1/4

```
ADDRINFO AI0, AI1;
ServSock[0] = socket(AF_INET6, SOCK_STREAM, PF_INET6
ServSock[1] = socket(AF_INET, SOCK_STREAM, PF_INET);
...
bind(ServSock[0], AI0->ai_addr, AI0->ai_addrlen);
bind(ServSock[1], AI1->ai_addr, AI1->ai_addrlen);
...
select(2, &SockSet, 0, 0, 0);
if (FD_ISSET(ServSocket[0], &SockSet)) {
    // IPv6 connection
    csock = accept(ServSocket[0], (LPSOCKADDR)&Fro
    ...
}
if (FD_ISSET(ServSocket[1], &SockSet)) {
    // IPv4 connection
    csock = accept(ServSocket[1], (LPSOCKADDR)&Fro
    ...
}
```



IPV6_V6ONLY option allows two versions of the same server process to run on the same port, one providing service over IPv6, the other providing the same service over IPv4.

```
$ netstat -nap |grep 5002

tcp    0      0 0.0.0.0:5002    0.0.0.0:*    LISTEN      3720/a.out
tcp6   0      0 :::5002         :::*         LISTEN      3720/a.out
```

Server (Separate Stacks) 3/4

Previous Example Code Output:

CLIENT **SERVER**

```
$ telnet ::1 5002
```

```
IPv6 connection
accept a connection from ::1
```

```
$ telnet 127.0.0.1 5002
```

```
IPv4 connection
accept a connection from 127.0.0.1
```

The two sockets are listening on the server :

```
$ netstat -nap |grep 5002

tcp    0      0 0.0.0.0:5002    0.0.0.0:*    LISTEN      3720/a.out
tcp6   0      0 :::5002         :::*         LISTEN      3720/a.out
```

Dual / Separated Stack Output

DUAL STACK

CLIENT

```
telnet 127.0.0.1 5001
```

SERVER

```
Accept a connection from ::ffff:127.0.0.1
```

```
$ netstat -nap |grep 5001
tcp6   0      0 :::5001         :::*         LISTEN      3735/a.out
```

SEPARATED STACK

CLIENT

```
$ telnet 127.0.0.1 5002
```

SERVER

```
IPv4 connection
accept a connection from 127.0.0.1
```

```
$ netstat -nap |grep 5002
tcp    0      0 0.0.0.0:5002    0.0.0.0:*    LISTEN      3720/a.out
tcp6   0      0 :::5002         :::*         LISTEN      3720/a.out
```

Server (Separate Stacks) 4/4

For the most curious, I've put a full code example here:

http://www.nucara.it/ipv6/

... BREAK ! ...

We will see at 2 pm

Address conversion functions (1/3)

```
#include <netinet/in.h>

unsigneg long int htonl (unsigned long int hostlong)
unsigned short int htons (unsigned short int hostshort)
unsigned long int ntohl (unsigned long int netlong)
unsigned short int ntohs (unsigned short int netshort)
```

DEPRECATED

Old address conversion functions (**working only with IPv4**) have been replaced by new IPv6 compatible ones:

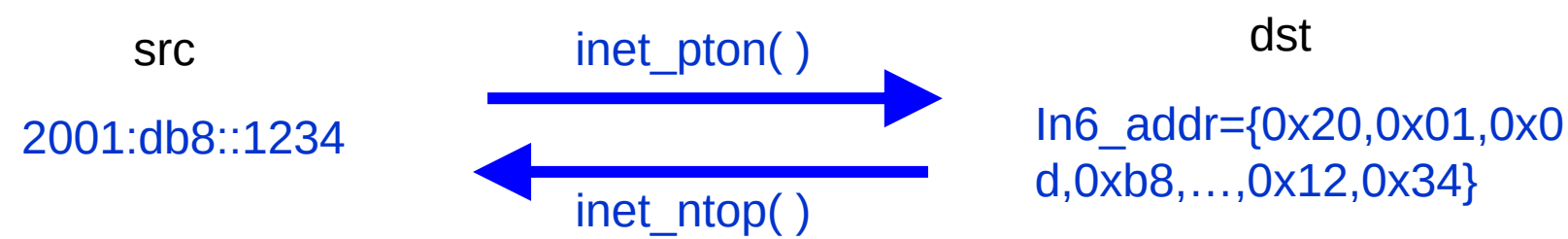
NEW

```
#include <arpa/inet.h>
int         inet_pton(int family, const char *src, void *dst);
const char *inet_ntop(int family, const void *src, char *dst,
size_t cnt);
```

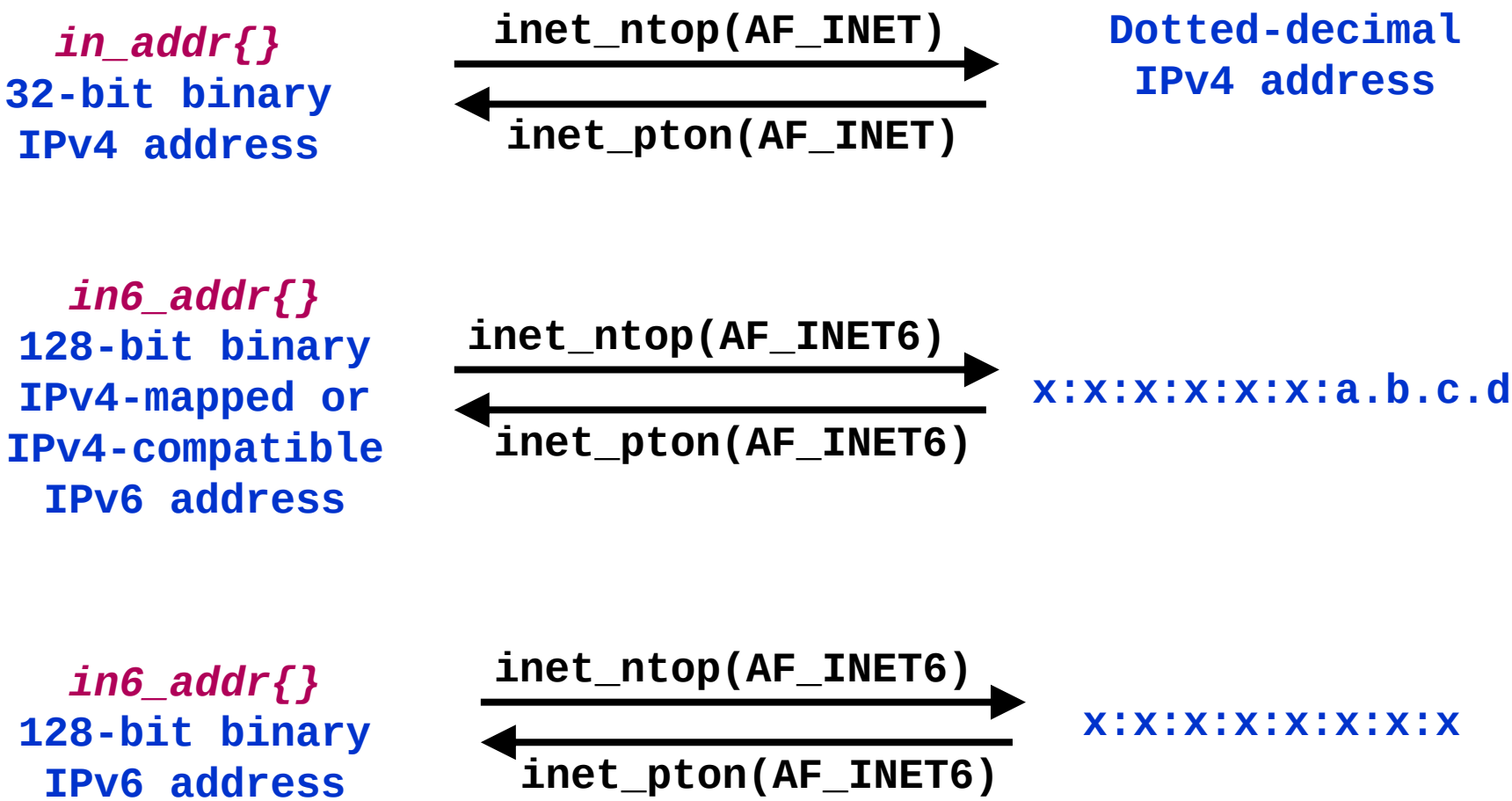
Address conversion functions (2/3)

Address conversion functions (**working with both IPv4 and IPv6 addresses**) to be used to switch between a binary representation and a human friendly presentation

```
#include <arpa/inet.h>
// From presentation to IPv4/IPv6 binary representation
int inet_pton(int family, const char *src, void *dst);

// From IPv4/IPv6 binary to presentation
const char *inet_ntop(int family, const void *src, char *dst,
size_t cnt);
```



Address Convertion Functions (3/3)



Address Convertion example

```
struct sockaddr_in6 addr;
char straddr[INET6_ADDRSTRLEN];

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

addr.sin6_family = AF_INET6; /* family */
addr.sin6_port = htons(MYPORT); /* port, networt byte order */

  //from presentation to binary representation
inet_pton(AF_INET6, “2001:720:1500:1::a100",&(addr.sin6_addr));

  //from binary representation to presentation
Straddr=inet_ntop(AF_INET6, &addr.sin6_addr, straddr,sizeof(straddr));
```

Moving towards Network Transparent Programming

- New functions have been defined to support both protocol versions (IPv4 and IPv6).
- A new way of programming and managing the socket has been introduced: network trasparent programming.
- According to this new approach the following functions have been defined:
  - getaddrinfo
  - getnameinfo

Going to Network Transparent Programming

- For Network Transparent Programming it is important to pay attention to:
  - **Use of name instead of address** in applications is advisable; in fact, usually the hostname remains the same, while the address may change more easily. From application point of view the name resolution is a system-independent process.
  - **Avoid the use of hard-coded numerical addresses** and binary representation of addresses.
  - **Use *getaddrinfo* and *getnameinfo* functions**.

Name to Address Translation Function

The **gethostbyname()** for IPv4 and ***gethostnyname2()*** function created for IPv6 **was deprecated** in RFC 2553 and was replaced by **getaddrinfo()** function.

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

DEPRECATED

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

DEPRECATED

```
#include <netdb.h>
#include <sys/socket.h>
int getaddrinfo(const char *nodename, const char *servname,
        const struct addrinfo *hints, struct addrinfo **res);
```

Nodename and Service Name Translation

Nodename-to-address translation is done in a protocol-independent way using the ***getaddrinfo()*** function.

int getaddrinfo(...)

Function returns:
0 for success
not 0 for error
(see *gai_strerror*)

const char *nodename

Host name or Address string

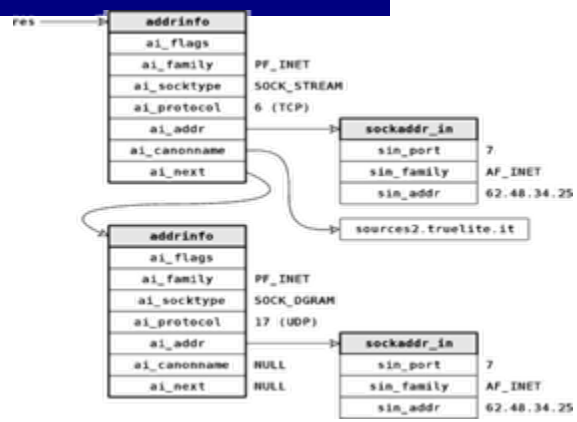
const char *servname

Servicename or decimal port
("http" or 80)

const struct addrinfo *hints

Options
Es. nodename is a numeric host addressing

struct addrinfo **res



Nodename and Service Name Translation

getaddrinfo() takes as input a service name like “http” or a numeric port number like “80” as well as an FQDN and returns a list of addresses along with the corresponding port number.

The *getaddrinfo* function is very flexible and has several modes of operation. It **returns a dynamically allocated linked list** of addrinfo structures containing useful information (for example, sockaddr structure ready for use).

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

int getaddrinfo(const char *nodename,
const char *servname, const struct addrinfo *hints,
struct addrinfo **res);
```

getaddrinfo input arguments

```
int getaddrinfo(const char *hostname, const char *servname,
        const struct addrinfo *hints, struct addrinfo **res);
```

hostname is either a hostname or an address string.

servname is either a service name or 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 he wants to be returned.
(see next slide)

getaddrinfo input arguments

The caller can set only these values in *hints* structure:

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

ai_family: The protocol family to return (es. AF_INET, AF_INET6, AF_UNSPEC). When *ai_family* is set to AF_UNSPEC, it means the caller will accept any protocol family supported by the operating system.
ai_socktype: Denotes the type of socket that is wanted: SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW. When *ai_socktype* is zero the caller will accept any socket type.
ai_protocol: Indicates which transport protocol is desired, IPPROTO_UDP or IPPROTO_TCP. If *ai_protocol* is zero the caller will accept any protocol.

getaddrinfo input arguments: ai_flag (1/3)

```
struct addrinfo {
  int      ai_flags;     // AI_PASSIVE, AI_CANONNAME, ..
[…]
};
```

ai_flags shall be set to zero or be the bitwise-inclusive OR of one or more of the values:

AI_PASSIVE
if it is specified the caller requires addresses that are suitable for accepting incoming connections. When this flag is specified, *nodename* is usually *NULL*, and address field of the *ai_addr* member is filled with the "any“ address (e.g. INADDR_ANY for an IPv4 or IN6ADDR_ANY_INIT for an IPv6).

AI_CANONNAME
the function shall attempt to determine the canonical name corresponding to nodename (The first element of the returned list has the ai_canonname filled in with the official name of the machine).

getaddrinfo input arguments: ai_flag (2/3)

```
struct addrinfo {
  int      ai_flags;     // AI_PASSIVE, AI_CANONNAME, ..
[…]
};
```

AI_NUMERICHOST
specifies that nodename is a numeric host address string. Otherwise, an [EAI_NONAME] error is returned. This flag shall prevent any type of name resolution service (for example, the DNS) from being invoked.

AI_NUMERICSERV
specifies that servname is a numeric port string. Otherwise, an [EAI_NONAME] error shall be returned. This flag shall prevent any type of name resolution service (for example, NIS+) from being invoked.

AI_V4MAPPED
if no IPv6 addresses are matched, IPv4-mapped IPv6 addresses for IPv4 addresses that match *nodename* shall be returned. This flag is applicable only when *ai_family* is AF_INET6 in the hints structure.

getaddrinfo input arguments: ai_flag (3/3)

```
struct addrinfo {
  int      ai_flags;     // AI_PASSIVE, AI_CANONNAME, ..
[…]
};
```

AI_ALL

If this flag is set along with AI_V4MAPPED when looking up IPv6 addresses the function will return all IPv6 addresses as well as all IPv4 addresses. The latter mapped to IPv6 format.

AI_ADDRCONFIG

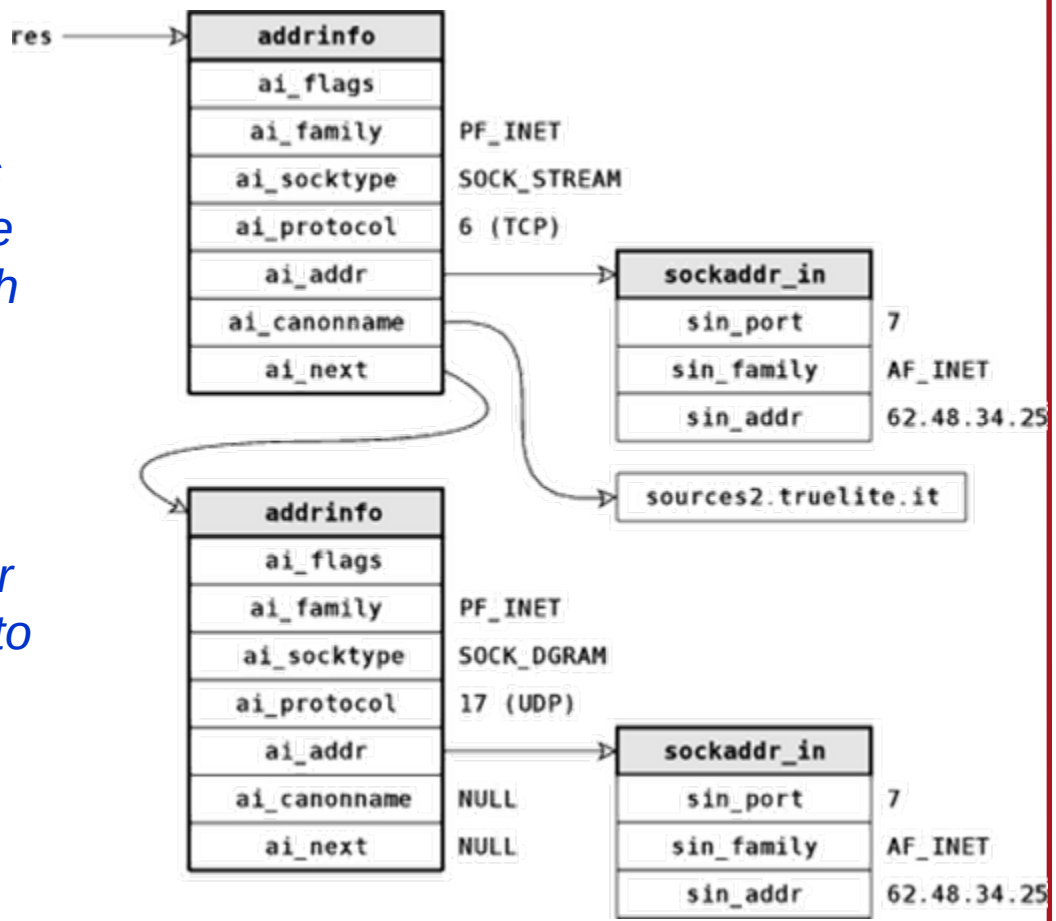
Only addresses whose family is supported by the system will be returned: IPv4 addresses shall be returned only if an IPv4 address is configured on the local system, and IPv6 addresses shall be returned only if an IPv6 address is configured on the local system. The loopback address is not considered for this case as valid as a configured address.

getaddrinfo output

If getaddrinfo returns 0 (success) ***res*** argument is filled in with a pointer to a linked list of addrinfo structures (linked through the *ai_next pointer.*

In case of multiple addresses associated with the hostname one struct is returned for each address (usable with hint.ai_family, if specified).

One struct is returned also for each socket type (according to hint.ai_socktype).



getaddrinfo output

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

The information returned in the *addrinfo* structures is ready for socket calls and ready to use in the *connect, sendto* (for client) or *bind* (for server) function.

ai_addr is a pointer to a socket address structure.
ai_addrlen is the length of this socket address structure.
ai_canonname member of the first returned structure points to the canonical name of the host (if AI_CANONNAME flag is set in hints structure).

gai_strerror (error handling)

```
#include <netdb.h>
char *gai_strerror(int error);
```

The nonzero error return values from *getaddrinfo* can be traslated by the *gai_strerror* function into a human readable string.

```
EAI_ADDRFAMILY - address family for hostname not supported: the
                 specified network host does not have any network
                 addresses in the requested address family.
EAI_AGAIN      - The  name  server  returned a temporary failure
                 indication.  Try again later.
EAI_BADFLAGS   - ai_flags contains invalid flags.
EAI_FAIL       - unrecoverable failure in name resolution
EAI_FAMILY     - The requested address family is not supported
                 at all.
EAI_MEMORY     - Out of memory.
EAI_NODATA     - The specified network host exists, but does not
                 have any network addresses defined.
EAI_NONAME     - hostname or service not provided or known
EAI_SERVICE    - service not supported for ai_socktype
EAI_SOCKTYPE   - The requested socket type is not supported at
                 all.
EAI_SYSTEM     - Other system error, check errno for details.
```

gai_strerror example

```
error=getaddrinfo("www.kame.net","http",&hints,&res0);

if(error)
{
  fprintf(stderr,"error: %s\n",gai_strerror(error));
  exit(1);
}
```

freeaddrinfo (memory release)

```
#include <netdb.h>
void freeaddrinfo(struct addrinfo *ai);
```

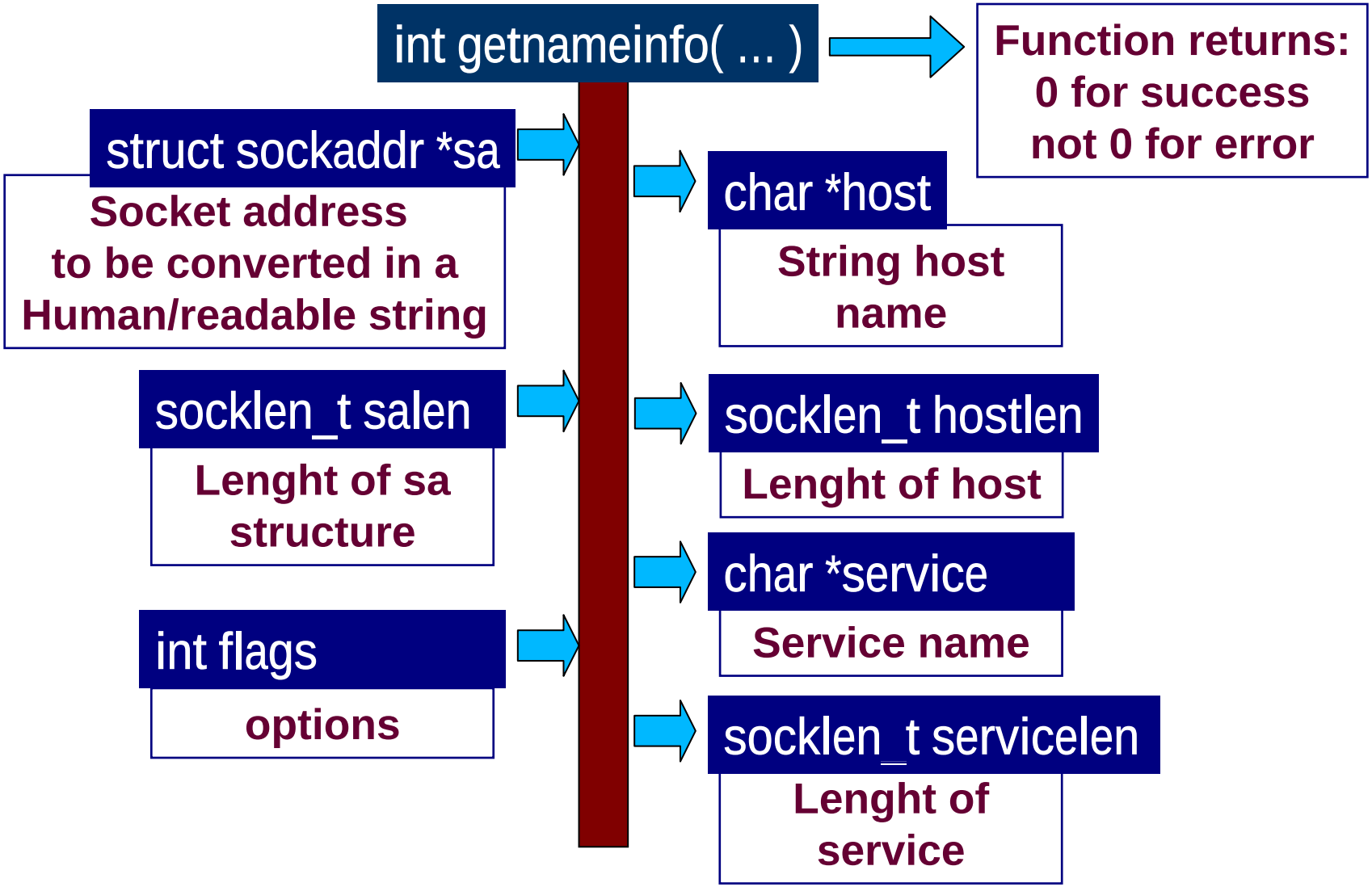
The *freeaddrinfo()* function frees addrinfo structure returned by getaddrinfo(), along with any additional storage associated with those structures (for example, storage pointed to by the ai_canonname and ai_addr fields).

freeaddrinfo example

```
n = getaddrinfo(hostname, service, &hints, &res);
//Try open socket with each address getaddrinfo returned, until
getting a valid socket.

resave = res;

while (res) {
  sockfd = socket(res->ai_family,
                  res->ai_socktype,
                  res->ai_protocol);
  if (!(sockfd < 0))
      break;
  res = res->ai_next;
}

freeaddrinfo(ressave);
```

Nodename and Service Name Translation

getnameinfo is the complementary function to getaddrinfo.



getnameinfo (1/4)

It takes a socket address and returns a character string describing the host and another character string describing the service.

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

int getnameinfo (const struct sockaddr *sa,
  socklen_t salen,char *host, socklen_t hostlen,
  char *service, socklen_t servicelen,
  int flags);
```

getnameinfo input and output (2/4)

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

int getnameinfo (const struct sockaddr *sa, socklen_t salen,
                 char *host, socklen_t hostlen,
                 char *service, socklen_t servicelen,
                   int flags);
```

sa (input) points to the socket address structure containing the protocol address to be converted in to a human-readable string. **salen** (input) is the length of this structure.

host (output) points to a buffer able to contain up to **hostlen** (output) characters containing the host name as a null-terminated string.

service (output) points to a buffer able to contain up to **servicelen** bytes that receives the service name as a null-terminated string. If the service's name cannot be located, the numeric form of the service address (for example, its port number) shall be returned instead of its name.

getnameinfo input and output (3/4)

To help allocate arrays to hold string pointed by host and service, two constants are defined : NI_MAXHOST and NIMAXSERV.

```
#include <netdb.h>
NI_MAXHOST //=1025 maximum size of returned host string
NI_MAXSERV //=32 maximum size of returned service string
```

getnameinfo (4/4)

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

int getnameinfo (const struct sockaddr *sa, socklen_t salen,
                 char *host, socklen_t hostlen,
                 char *service, socklen_t servicelen,
                   int flags);
```

flags changes the default actions of the function.

By default the fully-qualified domain name (FQDN) for the host shall be returned, but:

- If the flag bit NI_NOFQDN is set, **only the node name portion of the FQDN** shall be returned for local hosts.
- If the flag bit NI_NUMERICHOST is set, **the numeric form of the host's address** shall be returned instead of its name.

getnameinfo (4/4)

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

int getnameinfo (const struct sockaddr *sa, socklen_t salen,
                 char *host, socklen_t hostlen,
                 char *service, socklen_t servicelen,
                   int flags);
```

[…]

- If the flag bit NI_NAMEREQD is set, an error shall be returned if the host's name cannot be located.
- the flag bit NI_NUMERICSERV is set, **the numeric form of the service address** shall be returned (for example, its port number) instead of its name, under all circumstances.
- If the flag bit NI_DGRAM is set, this indicates that **the service is a datagram service** (SOCK_DGRAM). The default behavior shall assume that the service is a stream service (SOCK_STREAM).

Error Return Values

The getnameinfo() function shall fail and return the corresponding value if:

```
[EAI_AGAIN]    The name could not be resolved at this time.
                  Future attempts may succeed.

[EAI_BADFLAGS] The flags had an invalid value.

[EAI_FAIL]     A non-recoverable error occurred.

[EAI_FAMILY]   The address family was not recognized or the address
                  length was invalid for the specified family.

[EAI_MEMORY]   There was a memory allocation failure.

[EAI_NONAME]   The name does not resolve for the supplied
                  parameters.
                  NI_NAMEREQD is set and the host's name cannot be
                  located, or both nodename and servname were null.

[EAI_OVERFLOW] An argument buffer overflowed.

[EAI_SYSTEM]   A system error occurred.  The error code can be
                  found in errno.
```

getnameinfo examples

- Two examples will now follow, illustrating the **usage of *getaddrinfo*()**:
  - First illustrates the usage of **the result** from *getaddrinfo*() for **subsequent calls to socket() and to connect**().
  - Second **passively opens listening sockets** to accept incoming HTTP.

getnameinfo examples 1

```
struct addrinfo hints,*res,*res0; int error; int s;

memset(&hints,0,sizeof(hints));
hints.ai_family=AF_UNSPEC;
hints.ai_socktype=SOCK_STREAM;
error=getaddrinfo("www.kame.net","http",&hints,&res0);
[…]

s=-1;
for(res=res0; res; res=res->ai_next){
   s=socket(res->ai_family, res->ai_socktype,res->ai_protocol);
   if(s<0) continue;
   if(connect(s,res->ai_addr,res->ai_addrlen)<0){
        close(s); s=-1; continue;}
        break; // we got one!
   }
   if(s<0){fprintf(stderr,"No addresses are reachable");exit(1);}
   freeaddrinfo(res0);
}
```

getnameinfo examples 2

```
    struct addrinfo hints, *res, *res0;
    int error; int s[MAXSOCK]; int nsock; const char *cause=NULL;
memset (&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.aisocktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
error=getaddrinfo(NULL,"http", &hints, &res0);
 nsock=0;
for(res=res0; res && nsock<MAXSOCK; res=res->ai_next)
{
 s[nsock]=socket(res->ai_family, res->ai_socktype, res->ai_protocol);
 if(s[nsock]<0) continue;
#ifdef IPV6_V6ONLY
  if(res->ai_family == AF_INET6){int on=1;
  if(setsockopt(s[nsock],IPPROTO_IPV6,IPV6_V6ONLY,&on,sizeof(on)))
  {close(s[nsock]);continue;}}
#endif
 if(bind(s[nsock], res->ai_addr, res->ai_addrlen)<0)
  {close(s[nsock]);continue;}
  if(listen(s[nsock],SOMAXCONN)<0){close(s[nsock]);continue;}
        nsock++;
}
if(nsock==0){ /*no listening socket is available*/}
freeaddrinfo(res0);
}
```

Address Testing Macros

These are Macros described in RFC …

```
#include <netinet/in.h>

   int  IN6_IS_ADDR_UNSPECIFIED (const struct in6_addr *);
   int  IN6_IS_ADDR_LOOPBACK    (const struct in6_addr *);
   int  IN6_IS_ADDR_MULTICAST   (const struct in6_addr *);
   int  IN6_IS_ADDR_LINKLOCAL   (const struct in6_addr *);
   int  IN6_IS_ADDR_SITELOCAL   (const struct in6_addr *);
   int  IN6_IS_ADDR_V4MAPPED    (const struct in6_addr *);
   int  IN6_IS_ADDR_V4COMPAT    (const struct in6_addr *);

   int  IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr *);
   int  IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr *);
   int  IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr *);
   int  IN6_IS_ADDR_MC_ORGLOCAL (const struct in6_addr *);
   int  IN6_IS_ADDR_MC_GLOBAL   (const struct in6_addr *);
```

Porting application to IPv6 (1/2)

- To port IPv4 applications in a multi-protocol environment, developers should look out for these basic points
  - Use DNS names instead of numerical addresses
  - Replace incidental hard-coded addresses with other kinds
  - Sequences of zeros can be replaced by double colons sequence :: only one time per address, e.g. The previous address can be rewritten as 2001:760:40ec::12:3a
  - In the IPv6 RFCs and documentation, the minimum subnet mask is shown as /64, but in some cases, like point-to-point connections, a smaller subnet (such as /126) can be used.
  - In numerical addressing, RFC3986 specifies that squared brackets delimit IPv6 address to avoid mismatches with the port separator such as http://[2001:760:40ec::12:3a]:8000

Porting application to IPv6 (2/2)

- Applications in a dual-stack host prefer to use IPv6 address instead of IPv4
- In IPv6, it is normal to have multiple addresses associated to an interface. In IPv4, no address is associated to a network interface, while at least one (link local address) is in IPv6.
- All functions provided by broadcast in IPv4 are implemented on multicast in IPv6.
- The two protocols can not communicate directly, even in dual-stack hosts.

Rewriting applications

- To rewrite an application with IPv6 compliant code, the first step is to find all IPv4-dependent functions.
- A simple way is to check the source and header file with UNIX grep utility.

```
$ grep sockaddr_in *c *.h
$ grep in_addr *.c *.h
$ grep inet_aton *.c *.h
$ grep gethostbyname *.c *.h
```

Rewriting applications

- Developers should pay attention to hard-coded numerical address, host names, and binary representation of addresses.
- It is recommended to put all network functions in a single file.
- It is also suggested to replace all *gethostbyname* with the *getaddrinfo* function, a simple switch can be used to implement protocol dependent part of the code.
- Server applications must be developed to handle multiple listen sockets, one per address family, using the *select* call.

... BREAK ! ...

We will see at 15:45

Introduction to IPv6 Programming

In JAVA

IPv6 and Java (1/2)

- Java APIs are already IPv4/IPv6 compliant. IPv6 support in Java is available since **1.4.0 in Solaris and Linux** machines and since **1.5.0 for Windows XP and 2003 server**.
- IPv6 support in Java is automatic and transparent. Indeed no source code change and no bytecode changes are necessary. Every Java application is already IPv6 enabled if:
  - It does not use hard-coded addresses (no direct references to IPv4 literal addresses);
  - All the address or socket information is encapsulated in the Java Networking API;
  - Through setting system properties, address type and/or socket type preferences can be set;
  - It does not use non-specific functions in the address translation.

IPv6 and Java (2/2)

- IPv4-mapped address has significance only at the implementation of a dual-protocol stack and it is never returned.
- For new applications IPv6-specific new classes and APIs can be used.

Java code example (server)

```
import java.io.*;
import java.net.*;

ServerSocket serverSock = null;
Socket cs = null;

try {
 serverSock = new ServerSocket(5000);
 cs = serverSock.accept();
 BufferedOutputStream b = new
             BufferedOutputStream(cs.getOutputStream());
 PrintStream os = new PrintStream(b,false);
 os.println(“hallo!”); os.println("Stop");

 cs.close();
 os.close();
}catch (Exception e) {[...]}
```

Java code example (client)

```
import java.io.*;
import java.net.*;

Socket s = null; DataInputStream is = null;

try {
  s = new Socket("localhost", 5000);
  is = new DataInputStream(s.getInputStream());
  String line;
  while( (line=is.readLine())!=null ) {
    System.out.println("received: " + line);
    if (line.equals("Stop")) break;
  }
  is.close();
  s.close();
}catch (IOException e) { […] }
```

Class InetAddress

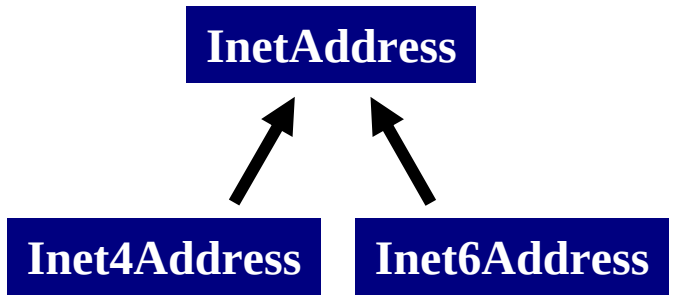
- This class represents an IP address.
  It provides:
  - address storage.
  - name-address translation methods.
  - address conversion methods.
  - address testing methods.

Inet4Address and Inet6Address

In J2SE 1.4, InetAddress is extended to support both IPv4 and IPv6 addresses.

Utility methods are added to check address types and scopes.

```
public final class Inet4Address extends InetAddress
public final class Inet6Address extends InetAddress
```



Inet4Address and Inet6Address

- The two types of addresses, IPv4 and IPv6, can be distinguished by the Java type Inet4Address and Inet6Address.
- V4 and V6 specific state and behaviors are implemented in these two subclasses.
- Due to Java's object-oriented nature, an application normally only needs to deal with InetAddress class—through polymorphism it will get the correct behavior.
- Only when it needs to access protocol-family-specific behaviors, such as in calling an IPv6-only method, or when it cares to know the class types of the IP address, will it ever become aware of Inet4Address and Inet6Address.

Class InetAddress

```
public static InetAddress getLocalHost()
                                    throws UnknownHostException
```

Returns the local host.

```
public static InetAddress getByName(String host)
                                    throws UnknownHostException
```

Determines the IP address of a host, given the host's name.

Class InetAddress

```
public byte[] getAddress()
```

Returns the raw IP address of this InetAddress object. The result is in network byte order: the highest order byte of the address is in getAddress()[0].

```
InetAddress addr=InetAddress.getLocalHost();
byte[] b=addr.getAddress();
for(int i: b){System.out.print(i+" ");}
```

```
Output:
127 0 0 1
```

```
public static InetAddress getByAddress(byte[] addr)
                                    throws UnknownHostException
```

Returns an InetAddress object given the raw IP address . The argument is in network byte order: the highest order byte of the address is in getAddress()[0].

This method doesn't block, i.e. no reverse name service lookup is performed.

IPv4 address byte array must be 4 bytes long and IPv6 byte array must be 16 bytes long

Class InetAddress

```
public static InetAddress[] getAllByName(String host)
                                    throws UnknownHostException
```

Given the name of a host, returns an array of its IP addresses, based on the configured name service on the system.

```
for (InetAddress ia : InetAddress.getAllByName("www.kame.net")) {
            System.out.println(ia);
}
```

```
output:
  www.kame.net/203.178.141.194
  www.kame.net/2001:200:0:8002:203:47ff:fea5:3085
```

Class InetAddress

```
public String getCanonicalHostName()
```

Gets the fully qualified domain name for this IP address.
Best effort method, meaning we may not be able to return the FQDN depending on the underlying system configuration.

```
System.out.println(
   InetAddress.getByName("www.garr.it").getCanonicalHostName()
);
```

```
output:
  lx1.dir.garr.it
```

```
public String getHostAddress()
```

Returns the IP address string in textual presentation.

```
addr = InetAddress.getByName("www.garr.it");
System.out.println(addr.getHostAddress());
```

```
output:
   193.206.158.2
```

Class InetAddress

```
public String getHostName()
```

Gets the host name for this IP address.

If this InetAddress was created with a host name, this host name will be remembered and returned; otherwise, a reverse name lookup will be performed and the result will be returned based on the system configured name lookup service.

```
System.out.print(
      InetAddress.getByName("www.garr.it").getHostName()
);
```

```
output:
   www.garr.it
```

```
System.out.print(
    InetAddress.getByName("193.206.158.2").getHostName()
);
```

```
output:
   lx1.dir.garr.it
```

Class InetAddress

```
public boolean isReachable(int timeout)
                                        throws IOException
```

```
public boolean isReachable(NetworkInterface netif,int ttl,int
timeout)
                                        throws IOException
```

Test whether that address is reachable. A typical implementation will use ICMP ECHO REQUESTs if the privilege can be obtained, otherwise it will try to establish a TCP connection on port 7 (Echo) of the destination host.

netif - the NetworkInterface through which the test will be done, or null for any interface
ttl - the maximum numbers of hops to try or 0 for the default
timeout - the time, in milliseconds, before the call aborts

A negative value for the ttl will result in an IllegalArgumentException being thrown. The timeout value, in milliseconds, indicates the maximum amount of time the try should take. If the operation times out before getting an answer, the host is deemed unreachable. A negative value will result in an IllegalArgumentException being thrown.

InetAddress Example

```
InetAddress ia=InetAddress.getByName("www.garr.it");
//or
InetAddress ia=InetAddress.getByName(“[::1]"); //or "::1"

String host_name = ia.getHostName();
System.out.println( host_name ); // ip6-localhost

String addr=ia.getHostAddress();
System.out.println(addr); //print IP ADDRESS
```

```
InetAddress[ ] alladr=ia.getAllByName("www.kame.net");
for(int i=0;i<alladr.length;i++) {
                    System.out.println(  alladr[i]  );  }
Output:
  www.kame.net/203.178.141.194
  www.kame.net/2001:200:0:8002:203:47ff:fea5:3085
```

New metods

In InetAddress class new metods have been added:

```
InetAddress.isAnyLocalAddress()
InetAddress.isLoopbackAddress()
InetAddress.isLinkLocalAddress()
InetAddress.isSiteLocalAddress()
InetAddress.isMCGlobal()
InetAddress.isMCNodeLocal()
InetAddress.isMCLinkLocal()
InetAddress.isMCSiteLocal()
InetAddress.isMCOrgLocal()
InetAddress.getCanonicalHostName()
InetAddress.getByAddr()
```

Inet6Address have a metod more than Inet4Address:

```
Inet6Address.isIPv4CompatibleAddress()
```

Code Example

```
Enumeration netInter = NetworkInterface.getNetworkInterfaces();
 while ( netInter.hasMoreElements() )
 {
   NetworkInterface ni = (NetworkInterface)netInter.nextElement();
   System.out.println( "Net. Int. : "+ ni.getDisplayName() );
   Enumeration addrs = ni.getInetAddresses();
     while ( addrs.hasMoreElements() )
     {
       Object o = addrs.nextElement();
       if ( o.getClass() == InetAddress.class ||
            o.getClass() == Inet4Address.class ||
            o.getClass() == Inet6Address.class )
       {
         InetAddress iaddr = (InetAddress) o;
         System.out.println( iaddr.getCanonicalHostName() );
         System.out.print("addr type: ");
         if(o.getClass() == Inet4Address.class) {…println("IPv4");}
         if(o.getClass() == Inet6Address.class){…println( "IPv6");}
         System.out.println( "IP: " + iaddr.getHostAddress() );
         System.out.println("Loopback? "+iaddr.isLoopbackAddress());
         System.out.println("SiteLocal?"+iaddr.isSiteLocalAddress());
         System.out.println("LinkLocal?"+iaddr.isLinkLocalAddress());
       }
     }
 }
```

Output Example

```
Net. Int. : eth0
------------------------
CanonicalHostName: fe80:0:0:0:212:79ff:fe67:683d%2
addr type: IPv6   IP: fe80:0:0:0:212:79ff:fe67:683d%2
Loopback? False  SiteLocal? False   LinkLocal? true
------------------------
CanonicalHostName: 2001:760:40ec:0:212:79ff:fe67:683d%2
addr type: IPv6   IP: 2001:760:40ec:0:212:79ff:fe67:683d%2
Loopback? False   SiteLocal? False   LinkLocal? false
------------------------
CanonicalHostName: pcgarr20.dir.garr.it
addr type: IPv4   IP: 193.206.158.140
Loopback? False   SiteLocal? False  LinkLocal? false

Net. Int. : lo
------------------------
CanonicalHostName: ip6-localhost
addr type: IPv6   IP: 0:0:0:0:0:0:0:1%1
Loopback? True   SiteLocal? False   LinkLocal? false
------------------------
CanonicalHostName: localhost
addr type: IPv4   IP: 127.0.0.1
Loopback? True   SiteLocal? False   LinkLocal? false
```

IPv6 Networking Properties: preferIPv4Stack

`java.net.preferIPv4Stack (default: false)`

If IPv6 is available on the operating system, the underlying native socket will be an IPv6 socket. This allows Java(tm) applications to connect too, and accept connections from, both IPv4 andIPv6 hosts.

If an **application has a preference to only use IPv4 sockets**, then this property can be set to true. **The implication is that the application will not be able to communicate with IPv6 hosts.**

java.net.preferIPv4Stack Example (1/3)

```
$ java networkInt

Net. Int. : eth0
-------------------------
CanonicalHostName: fe80:0:0:0:212:79ff:fe67:683d%2
IP: fe80:0:0:0:212:79ff:fe67:683d%2
-------------------------
CanonicalHostName: 2001:760:40ec:0:212:79ff:fe67:683d%2
IP: 2001:760:40ec:0:212:79ff:fe67:683d%2
-------------------------
CanonicalHostName: pcgarr20.dir.garr.it
IP: 193.206.158.140

Net. Int. : lo
-------------------------
CanonicalHostName: ip6-localhost
IP: 0:0:0:0:0:0:0:1%1
-------------------------
CanonicalHostName: localhost
IP: 127.0.0.1
```

java.net.preferIPv4Stack Example (2/3)

```
$ java -Djava.net.preferIPv4Stack=true networkInt

Net. Int. : eth0
-------------------------
CanonicalHostName: pcgarr20.dir.garr.it
IP: 193.206.158.140

Net. Int. : lo
-------------------------
CanonicalHostName: localhost
IP: 127.0.0.1
```

java.net.preferIPv4Stack Example (2/3)

To configure **java.net.preferIPv4Stack** it is possible to use the "-D" option while launching the application

```
$ java -Djava.net.preferIPv4Stack=true networkInt
```

… or configure this property directly in the code:

```
System.setProperty("java.net.preferIPv4Stack","true");
```

```
Properties p = new Properties(System.getProperties());
p.setProperty("java.net.preferIPv6Addresses", "true");
System.setProperties(p);
```

IPv6 Networking Properties: preferIPv6Addresses

`java.net.preferIPv6Addresses (default: false)`

If IPv6 is available on the operating system, the default preference is to prefer an IPv4-mapped address over an IPv6 address. This is for backward compatibility reasons—for example, applications that depend on access to an IPv4-only service, or applications that depend on the %d.%d.%d.%d representation of an IP address.

This property can be set to try to change the preferences to use IPv6 addresses over IPv4 addresses. This allows applications to be tested and deployed in environments where the application is expected to connect to IPv6 services.

java.net.preferIPv6Addresses Example

```
//System.setProperty("java.net.preferIPv6Addresses",“false");
InetAddress ia=InetAddress.getByName("www.kame.net");
String ss=ia.getHostAddress();
System.out.println(ss); //print 203.178.141.194


System.setProperty("java.net.preferIPv6Addresses","true");
InetAddress ia=InetAddress.getByName("www.kame.net");
String ss=ia.getHostAddress();
System.out.println(ss);//print 2001:200:0:8002:203:47ff:fea5:3085
```

```
$java -Djava.net.preferIPv6Addresses=true -jar test.jar
2001:200:0:8002:203:47ff:fea5:3085

$java -Djava.net.preferIPv6Addresses=false -jar test.jar
203.178.141.194
$java -jar test.jar
203.178.141.194
```

setPerformancePreferences

```
java.net.SocketImpl
public void setPerformancePreferences(
        int connectionTime, int latency, int bandwidth)
```

This method allows the application to express its own preferences to tune the performance characteristics of this socket. Performance preferences are described by three integers whose values indicate the relative importance of short connection time, low latency, and high bandwidth. With this method, the network oriented notion of Quality of Service (QoS) is introduced. Any application can set its preferences to adapt its network traffic and provide the best QoS.

connectionTime: An int expressing the relative importance of a short connection time.
latency: An int expressing the relative importance of low latency.
bandwidth: An int expressing the relative importance of high bandwidth.

If the application prefers short connection time over both low latency and high bandwidth, for example, then it could invoke this method with the values (1, 0, 0).
If the application prefers high bandwidth above low latency, and low latency above short connection time, then it could invoke this method with the values (0, 1, 2).

Bibliography

Bibliography (from Web)

- Programming guidelines on transition to IPv6
  Tomás P. de Miguel and Eva M. Castro
  http://long.ccaba.upc.es/long/045Guidelines/LONG_Trans_app_IPv6.pdf
- Networking IPv6 User Guide for JDK/JRE 5.0
  http://java.sun.com/j2se/1.5.0/docs/guide/net/ipv6_guide/index.html
- Simone Piccardi, Guida alla Programmazione in Linux
  http://www.lilik.it/~mirko/gapil/gapil.html
- Java and IPv6 (slides)
  Jean Christophe Collet
  http://72.34.43.90/IPV6/North_American_IPv6_Summit_2004/Wednesday/PDFs/Jean_Christophe_Collet.pdf

Bibliography (Books)

- Qing Li – Tatuya Jinmei – Keiichi Shima
  IPv6 Core Protocols Implementation
  MORGAN KAUFMAN
- W. Richard Stevens
  Unix Network Programming (second edition)
  Prentice Hall
- Jun-ichiro – itojun Hagino
  IPv6 Network Programming
  ELSEVIER DIGITAL PRESS
- Elliotte Rusty Harold
  Java Network Programming
  O'REILLY
- Merlin Hughes – Michael Shoffner – Derek Hammer
  Java Network Programming 2 edition
  MANNING

END

Introduction to IPv6 Programming

In PERL

Perl for IPv6

An IPv6 function-set for the perl language is provided by the **Socket6 module**. Like the Socket core module for IPv4, this module provides a C-Style set of functions to open and manipulate sockets in IPv6. The general structure of the module and the address data structures are similar to the C programming interface.

Developers should take care of the same general concepts described in section about introduction programming IPv6 in C.

The module is available on the CPAN web site. To work properly, the module must be included in the code in conjunction to the core module Socket.

```
use Socket
Use Socket6
```

Function list (1/6)

```
BINARY_ADDRESS = inet_pton (FAMILY, TEXT_ADDRESS)
```

This function **converts** string format IPv4/IPv6 addresses to binary format, the FAMILY argument specify the type of address (AF_INET or AF_INET6).

```
TEXT_ADDRESS = inet_ntop (FAMILY, BINARY_ADDRESS)
```

This function **converts** an address in binary format to string format; like for the previous function (inet_pton), the FAMILY argument must be used to specify the family type of the address.

example

```
$a=inet_ntop(AF_INET6,inet_pton(AF_INET6,"::1"));
print $a; //print ::1
```

Function list (2/6)

```
STRUCT_ADDR = pack_sockaddr_in6 (PORT, ADDRESS)
```

This function returns a sockaddr_in6 structure filled with PORT and ADDRESS arguments in the correct fields. The ADDRESS argument is a 16-byte structure (as returned by inet_pton). The other fields of the structure are not set.

```
(PORT,STRUCT_ADDR) = unpack_sockaddr_in6 (ADDR)
```

This function unpacks a sockaddr_in6 structure to an array of two elements, where the first element is the port number and the second element is the address included in the structure.

example

```
$lh6=inet_pton(AF_INET6,"::1");
$p_saddr6=pack_sockaddr_in6(80,$lh6);
($port,$host) = unpack_sockaddr_in6($p_saddr6);
print inet_ntop(AF_INET6,$host); //print ::1
print $port; //print 80
```

Function list (3/6)

pack_sockaddr_in6_all (PORT, FLOWINFO, ADDRESS, SCOPEID)

This function returns a sockaddr_in6 structure filled with the four specified arguments.

unpack_sockaddr_in6_all (NAME)

This function unpacks a sockaddr_in6 structure to an array of four element:

- The port number
- Flow informations
- IPv6 network address (16-byte format)
- The scope of the address

Function list (4/6)

```
getaddrinfo(NODENAME,SERVICENAME,[FAMILY,SOCKTYPE,PROTOCOL,FLAGS])
```

This function converts node names to addresses and service names to port numbers. At least one of NODENAME and SERVICENAME must have a true value. If the lookup is successful this function **returns an array of information** blocks. Each information block consists of five elements: **address family, socket type, protocol, address and canonical name** if specified.
The arguments in squared brackets are optional.

```
getnameinfo (NAME, [FLAGS])
```

This function returns a node or a service name. The optional attribute FLAGS controls what kind of lookup is performed.

Example

```
use Socket;
use Socket6;
@res = getaddrinfo('hishost.com', 'daytime', AF_UNSPEC,SOCK_STREAM);
 $family = -1;
 while (scalar(@res) >= 5) {
  ($family, $socktype, $proto, $saddr, $canonname, @res)=@res;
  ($host, $port) =getnameinfo($saddr,NI_NUMERICHOST|NI_NUMERICSERV);
   print STDERR "Trying to connect to $host port $port...\n";

   socket(Socket_Handle, $family, $socktype, $proto) || next;
   connect(Socket_Handle, $saddr) && last;
   close(Socket_Handle);
   $family = -1;
 }

 if ($family != -1) {
   print STDERR "connected to $host port port $port\n";
 } else {
   die "connect attempt failed\n";
 }
```

Example

```
use Socket; use Socket6;      use IO::Handle;      $family = -1;

@res = getaddrinfo('www.kame.net', 'http', AF_UNSPEC, SOCK_STREAM);
while (scalar(@res) >= 5) {
  ($family, $socktype, $proto, $saddr, $canonname, @res) = @res;
  ($host,$port) = getnameinfo($saddr,NI_NUMERICHOST|NI_NUMERICSERV);
  print STDERR "Trying to connect to $host port $port...\n";

  socket(Socket_Handle, $family, $socktype, $proto) || next;

  connect(Socket_Handle, $saddr) && last;
  close(Socket_Handle); $family = -1;
}

if ($family != -1) { Socket_Handle->autoflush();
    print Socket_Handle "GET\n";
    print STDERR "connected to $host port port $port\n";
    while($str=<Socket_Handle>){print STDERR $str;}
}else {die "connect attempt failed\n";}
```

Example output

```
Trying to connect to 2001:200:0:8002:203:47ff:fea5:3085 port 80 …
connected to 2001:200:0:8002:203:47ff:fea5:3085 port 80

[…]<title>The KAME project</title>[…] The KAME project […]
<img src="/img/kame-anime-small.gif" alt="Dancing kame" /> […]
```

Example output

```
(…)=getnameinfo($saddr,NI_NUMERICHOST|NI_NUMERICSERV);
print STDERR "Trying to connect to $host port $port…";
```

OUTPUT:

```
Trying to connect to 2001:200:0:8002:203:47ff:fea5:3085 port 80 …
```

```
(…)=getnameinfo($saddr,0);
print STDERR "Trying to connect to $host port $port…";
```

OUTPUT:

```
Trying to connect to orange.kame.net port www …
```

Function list (5/6)

gethostbyname2 (HOSTNAME, FAMILY)

This function is the multiprotocol implementation of gethostbyname; the address family is selected by the FAMILY attribute. This function converts node names to addresses.

gai_strerror (ERROR_NUMBER)

This function returns a string corresponding to the error number passed in as an argument.

in6addr_any

This function returns the 16-octet wildcard address.

in6add_loopback

This function returns the 16-octet loopback address.

Function list (6/6)

`getipnodebyname (HOST, [FAMILY, FLAGS])`

This function takes either a node name or an IP address string and performs a lookup on that name (or conversion of the string). It returns a list of five elements: the canonical host name, the address family, the length in octets of the IP addresses returned, a reference to a list of IP address structures, and a reference to a list of aliases for the host name. This function was **deprecated** in RFC3493. The getnameinfo function should be used instead.

`getipnodebyaddr (FAMILY, ADDRESS)`

This function takes an IP address family and an IP address structure and performs a reverse lookup on that address. This function was **deprecate**d in RFC3493: the getaddrinfo function should be used instead.

Introduction to IPv6 Programming

In PHP

PHP and IPv6

- IPv6 is supported by default in PHP-4.3.4 and PHP-5.2.3 versions.
- Few functions have been defined to support IPv6.
- sparse documentation is provided for IPv6 programming

inet_pton

```
string inet_pton ( string $address )
```

This function converts a human readable IPv4 or IPv6 address (if PHP was built with IPv6 support enabled) into an address family appropriate 32bit or 128bit binary structure.

Note: This function is not implemented on Windows platforms.

```
$in_addr = inet_pton('127.0.0.1');
$in6_addr = inet_pton('::1');
```

inet_ntop

```
string inet_ntop ( string $in_addr )
```

This function converts a 32bit IPv4, or 128bit IPv6 address (if PHP was built with IPv6 support enabled) into an address family appropriate string representation. Returns FALSE on failure.

Note: This function is not implemented on Windows platforms.

```
$packed = chr(127) . chr(0) . chr(0) . chr(1);
$expanded = inet_ntop($packed);

echo $expanded; /* Outputs: 127.0.0.1 */


$packed = str_repeat(chr(0), 15) . chr(1);
$expanded = inet_ntop($packed);

echo $expanded; /* Outputs: ::1 */
```

fsockopen

```
resource fsockopen ( string $target [, int $port [, int &$errno
[, string &$errstr [, float $timeout]]]] )
```

Initiates a socket connection to the resource specified by target.

fsockopen() returns a file pointer which may be used together with the other file functions (such as fgets(), fgetss(), fwrite(), fclose(), and feof()).

Depending on the environment, the Unix domain or the optional connect timeout may not be available.

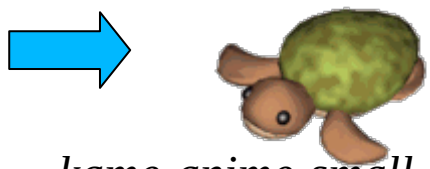
example code

```
$ip=…
$fp = fsockopen($ip, 80, $errno, $errstr);
if (!$fp) {
    echo "ERROR: $errno - $errstr<br />\n";
} else {
    //$status = socket_get_status($fp);
    fwrite($fp, "GET\n");
    $txt=fread($fp, 1000);
    echo $txt;
    fclose($fp);
}
```

*kame-**no**-anime-small.gif*

```
//www.kame.net
$ip="203.178.141.194“;
```

```
//www.kame.net
$ip='[2001:200::8002:203:47ff:fea5:3085]';
```



kame-anime-small.gif

gethostbyname

```
string gethostbyname ( string $hostname )
```

Returns the IP address of the Internet host specified by hostname or a string containing the unmodified hostname on failure.

gethostbyname and gethostbynamel do not ask for AAAA records.

BUT we can write code for asking AAAA records using **dns_get_record()** function

dns_get_record

```
array dns_get_record (string $hostname [,int $type])
```

This function fetch DNS Resource Records associated with a hostname.

dns_get_record returns an array of associative arrays. Each associative array contains at minimum the following keys:

host: the record in the DNS namespace to which the rest of the associated data refers.
class: dns_get_record() only returns Internet class records and as such this parameter will always return IN.
type: string containing the record type. Additional attributes will also be contained in the resulting array dependant on the value of type.
ttl:Time To Live remaining for this record.

$type limits the query (es. DNS_A,DNS_AAAA).

Note: This function is not implemented on Windows platforms, nor does it (currently) work on *BSD systems. Try the PEAR class Net_DNS.

checkdnsrr

```
int checkdnsrr ( string $host [, string $type] )
```

Check DNS records corresponding to a given Internet host name or IP address.
Returns TRUE if any records are found; returns FALSE if no records were found or if an error occurred.

type may be any one of: A, MX, NS, SOA, PTR, CNAME, AAAA, A6, SRV, NAPTR or ANY. The default is MX.

host may either be the IP address in dotted-quad notation or the host name.

Note: AAAA type added with PHP 5.0.0

Note: This function is not implemented on Windows platforms.
(use PEAR >> Net_DNS)

```
echo checkdnsrr ('www.kame.net','AAAA'); // return 1.
```

PEAR Net_IPv6 (1/7)

```
require_once 'Net/IPv6.php';
```

Pear Net_IPv6 Provides function to work with the 'Internet Protocol v6'.

Net_IPv6::checkIPv6() -- Validation of IPv6 addresses
Net_IPv6::compress() -- compress an IPv6 address
Net_IPv6::uncompress() -- Uncompresses an IPv6 address
Net_IPv6::getAddressType() -- Returns the type of an IP address
Net_IPv6::getNetmask() -- Calculates the network prefix
Net_IPv6::isInNetmask() -- Checks if an IP is in a specific address space
Net_IPv6::removeNetmaskSpec() -- Removes the Netmask length specification
Net_IPv6::splitV64() -- splits an IPv6 address in it IPv6 and IPv4 part

```
boolean Net_IPv6::checkIPv6 (string $ip)
```

Checks an IP for IPv6 compatibility. $ip is the ip to check.

PEAR Net_IPv6 (2/7)

`string Net_IPv6::compress (string $ip)`

Compresses an IPv6 address. RFC 2373 allows you to compress zeros in an address to '::'. This function expects an valid IPv6 address and compresses successive zeros to '::'

Es. FF01:0:0:0:0:0:0:101 -> FF01::101
0:0:0:0:0:0:0:1 -> ::1

`string Net_IPv6::uncompress (string $ip)`

Uncompresses an IPv6 address. RFC 4291 allows you to compress zeros in an address to '::'. This function expects an valid IPv6 address and expands the '::' to the required zeros.

Es. FF01::101 -> FF01:0:0:0:0:0:0:101

::1 -> 0:0:0:0:0:0:0:1

PEAR Net_IPv6 (3/7)

```
int Net_IPv6::getAddressType (string $ip)
```

Returns the type of an IP address. RFC 2460, Section 2.3 describes several types of addresses in the IPv6 addresse space. This methods tries to find the type of address for the given IP.

Return the addresstype (int). the type can be one of this constants:

- NET_IPV6_MULTICAST
- NET_IPV6_UNICAST_PROVIDER
- NET_IPV6_LOCAL_LINK
- NET_IPV6_LOCAL_SITE
- NET_IPV6_UNKNOWN_TYPE
- NET_IPV6_RESERVED_UNICAST_GEOGRAPHIC
- NET_IPV6_RESERVED_IPX
- NET_IPV6_RESERVED
- NET_IPV6_RESERVED_NSAP
- NET_IPV6_UNASSIGNED

PEAR Net_IPv6 (4/7)

```
string Net_IPv6::getNetmask (string $ip [, int $bits = null])
```

Calculates the network prefix based on the netmask bits.

string $ip : the IP address in Hex format, compressed IPs are allowed

int $bits : if the number of netmask bits is not part of the IP, you must provide the mumber of bits

Return value: string - the network prefix

note:This function can be called statically.

PEAR Net_IPv6 (5/7)

```
boolean Net_IPv6::isInNetmask (string $ip, string $netmask [, int $bits = null])
```

Checks if an (compressed) IP is in a specific address space. If the IP does not contain the number of netmask bits (for example: F8000::FFFF/16), then you have to use the $bits parameter.

string $ip - the IP address in Hex format, compressed IPs are allowed
string $netmask - the netmask (for example: F800::)
int $bits - if the number of netmask bits is not part of the IP, you must provide the number of bits

Return value

boolean - TRUE, if the IP is in the address space.

This function can be called statically.

PEAR Net_IPv6 (6/7)

```
string Net_IPv6::removeNetmaskSpec (string $ip)
```

Removes a possible existing netmask length specification in an IP address.

string $ip - the IP address in Hex format, compressed IPs are allowed

Return value

string - the IP without netmask length specification.

This function can be called statically.

PEAR Net_IPv6 (7/7)

`array Net_IPv6::splitV64 (string $ip)`

Splits an IPv6 address into the IPv6 and a possible IPv4-formated part. RFC 2373 allows you to note the last two parts of an IPv6 address in the IPv4 address format.

Parameter

string $ip - the IP address to split

Return value

array - key [0] contains the IPv6 part
key [1] the IPv4 formated part

This function can be called statically.

Bibliography

Bibliography (from Web)

- Programming guidelines on transition to IPv6
  Tomás P. de Miguel and Eva M. Castro
  http://long.ccaba.upc.es/long/045Guidelines/LONG_Trans_app_IPv6.pdf
- Networking IPv6 User Guide for JDK/JRE 5.0
  http://java.sun.com/j2se/1.5.0/docs/guide/net/ipv6_guide/index.html
- Simone Piccardi, Guida alla Programmazione in Linux
  http://www.lilik.it/~mirko/gapil/gapil.html
- www.PHP.net
- Java and IPv6 (slides)
  Jean Christophe Collet
  http://72.34.43.90/IPV6/North_American_IPv6_Summit_2004/Wednesday/PDFs/Jean_Christophe_Collet.pdf

Bibliography (Books)

- Qing Li – Tatuya Jinmei – Keiichi Shima
  IPv6 Core Protocols Implementation
  MORGAN KAUFMAN
- W. Richard Stevens
  Unix Network Programming (second edition)
  Prentice Hall
- Jun-ichiro – itojun Hagino
  IPv6 Network Programming
  ELSEVIER DIGITAL PRESS
- Elliotte Rusty Harold
  Java Network Programming
  O'REILLY
- Merlin Hughes – Michael Shoffner – Derek Hammer
  Java Network Programming 2 edition
  MANNING