



Enabling Grids for E-science

IPvX (AF-independent) programming introduction

Basic introduction to C/C++, JAVA, Perl, Python IPv6 programming

High level networking libraries

Rino Nucara (GARR) – rino.nucara@garr.it

Mario Reale (GARR) – mario.reale@garr.it

Etienne Duple (UREC) – etienne.duple@urec.cnrs.fr

v0.6

This document is available at <https://edms.cern.ch/document/976004>

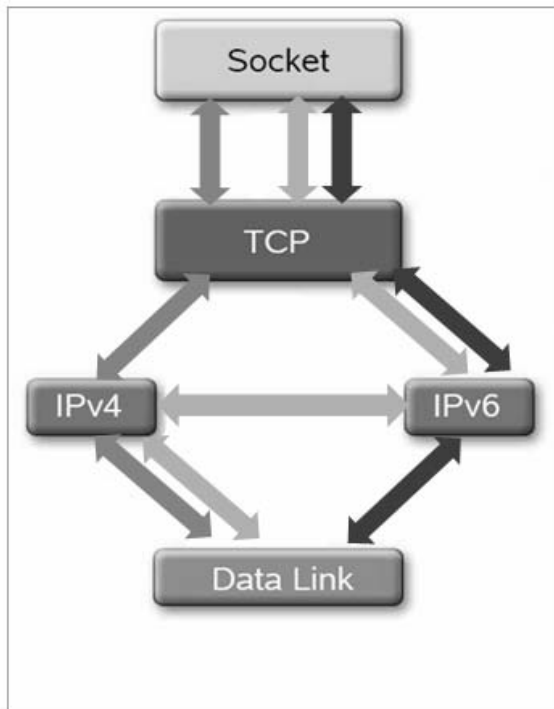
www.eu-egee.org

*IPv6 tutorial @ JRA1/SA3 all hands
Meeting 6 Nov 2008 - Prague*

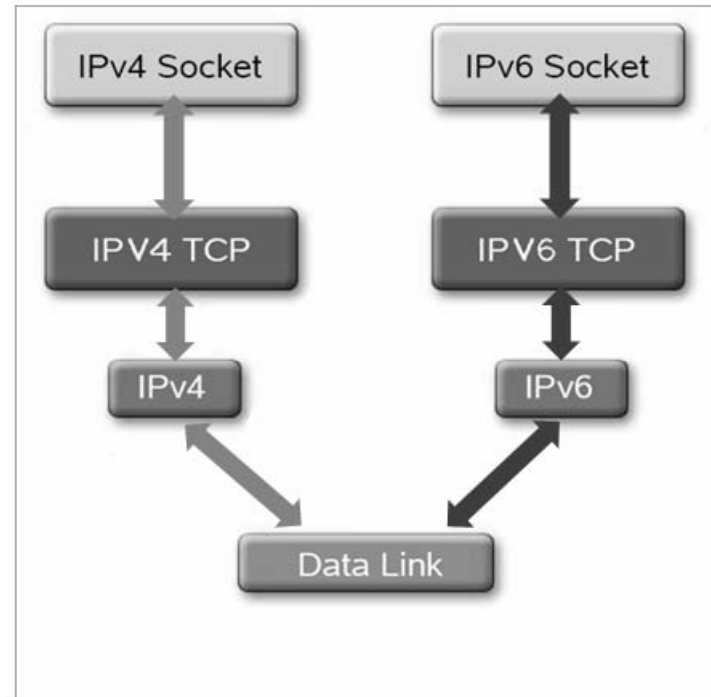


- IPv4 and IPv6 will co-exist for a number of years in future
- IPvX Servers and Clients will need to connect to IPvX Clients and Servers
- To write compatible applications with both IPv4 and IPv6 the following two solutions are available:
 - 1) Use **one IPv6 Socket**, able to handle both IPv4 and IPv6
 - IPv4 can be seen as a *special case* of IPv6 (*IPv4-mapped addresses*)
 - 2) Open **one Socket for IPv4** and **one Socket for IPv6**.
 - Prevent IPv6 Socket from the possibility to deal with IPv4 connections
- To build AF-independent applications one must not a-priori assume a specific version of the IP protocol
 - For doing this, proper functions have been introduced

DUAL STACK OS



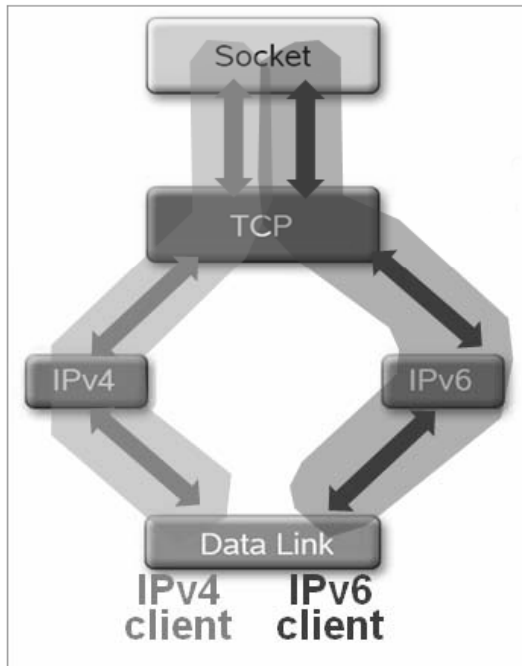
SEPARATED STACKS OS (obsolete)



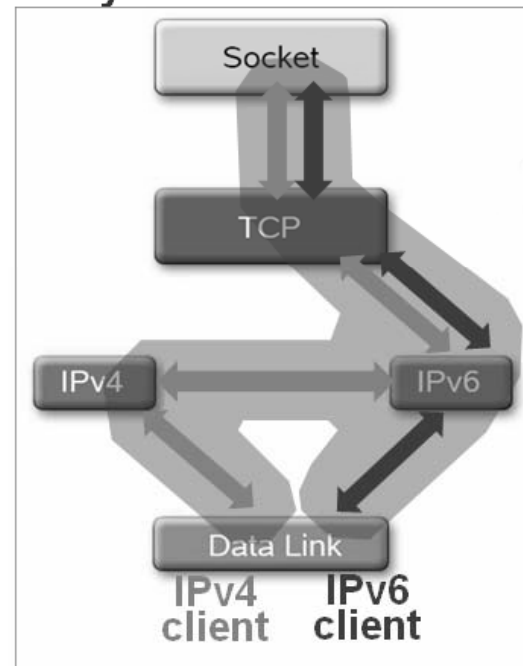
-  **IPv4 traffic**
-  **IPv4 mapped into IPv6 traffic**
-  **IPv6 traffic**

On dual-stack systems, in order to accept both IPv4 and IPv6, socket servers can be designed in two ways:

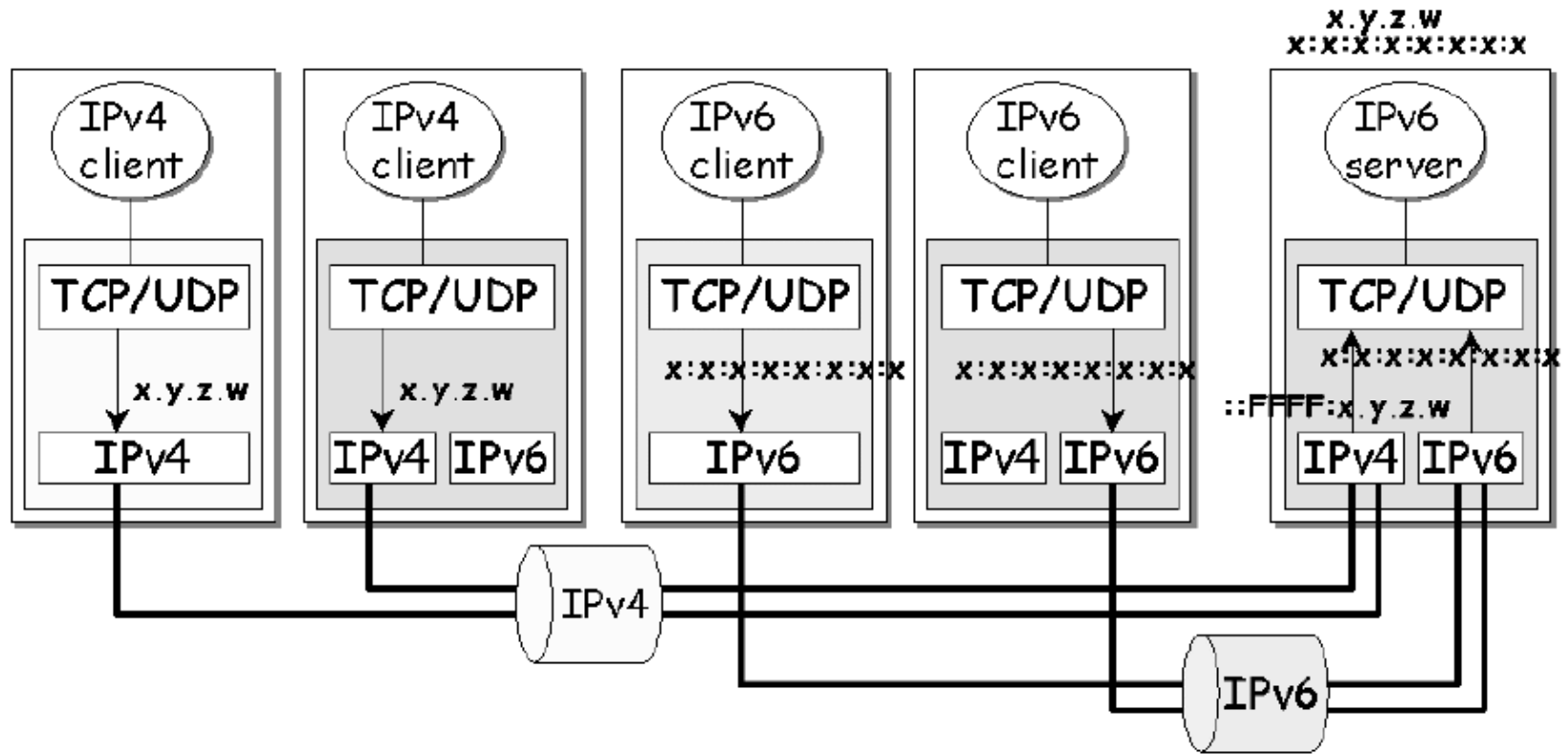
Two sockets:




Only one IPv6 socket:

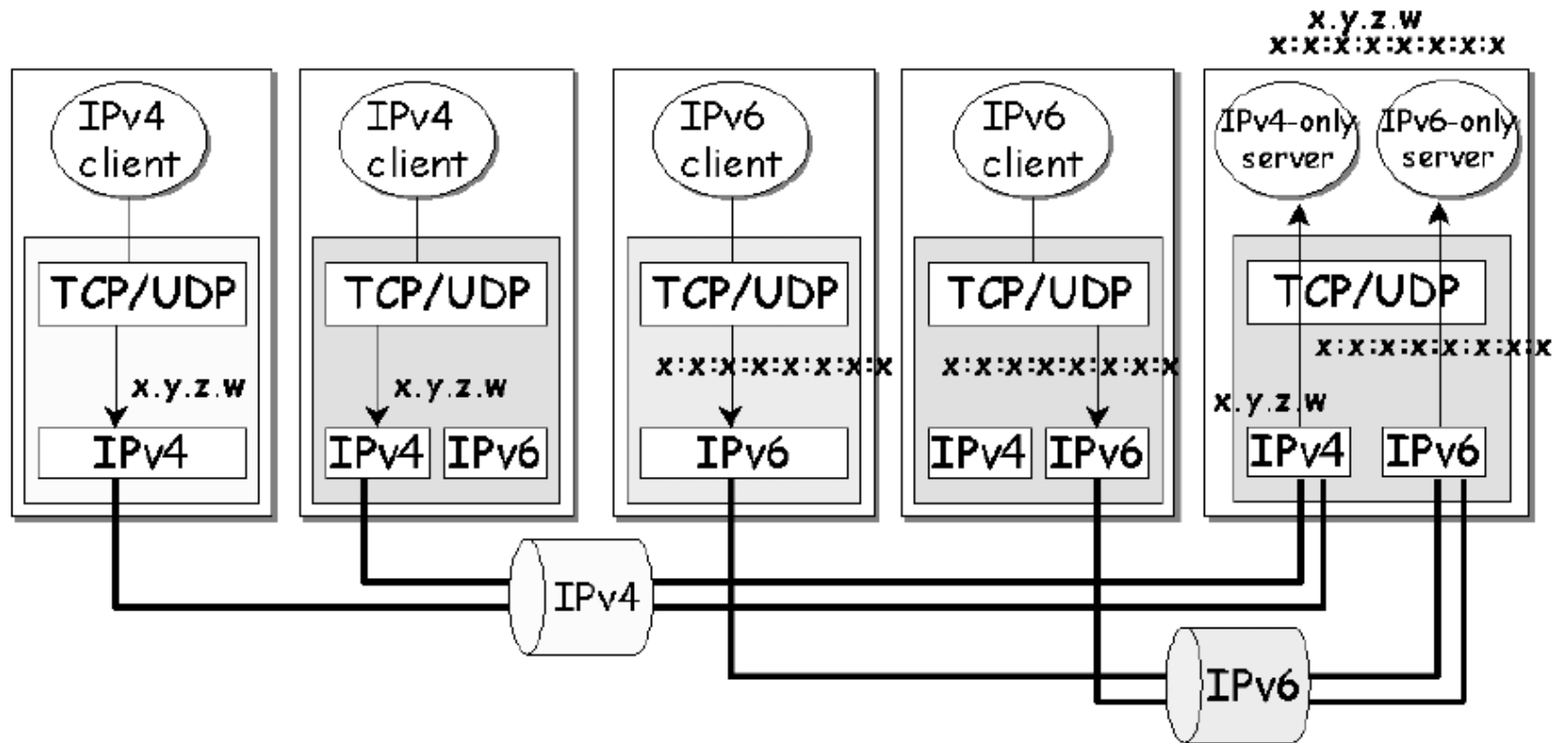



-  Traffic bound to one IPv4 socket
-  Traffic bound to one IPv6 socket




 *Dual Stack*

 *Single IPv4 or IPv6 stacks*



 *Dual Stack or separated stack*

 *Single IPv4 or IPv6 stacks*

Introduction to IPv6 Programming In C

- **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

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 */
```

- **IPv4**
 - *struct sockaddr_in*
 - *struct sockaddr*

- **IPv6**
 - *struct sockaddr_in6*

- **IPv4/IPv6/...**
 - *struct sockaddr_storage*

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

```
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)
}
```

```

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)

```

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)

```

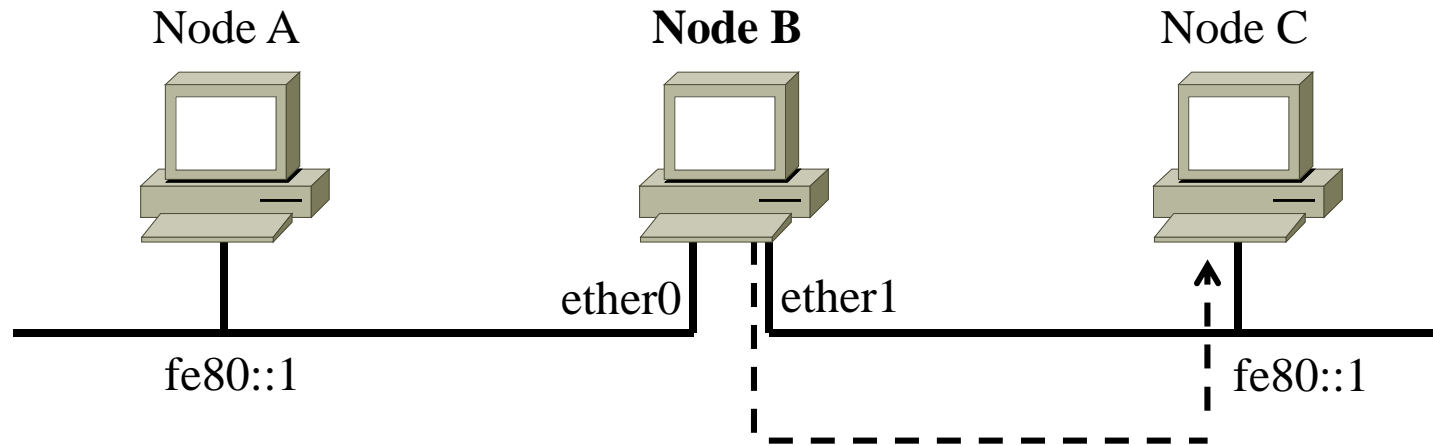
struct in6_addr {
    uint8_t  s6_addr[16]; // 128-bit IPv6 address (N.B.O.)
};
struct sockaddr_in6 {
    sa_family_t    sin6_family; //AF_INET6
    in_port_t      sin6_port;    //transport layer port # (N.B.O.)
    uint32_t       sin6_flowinfo; //IPv6 flow information (N.B.O.)
    struct in6_addr sin6_addr;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

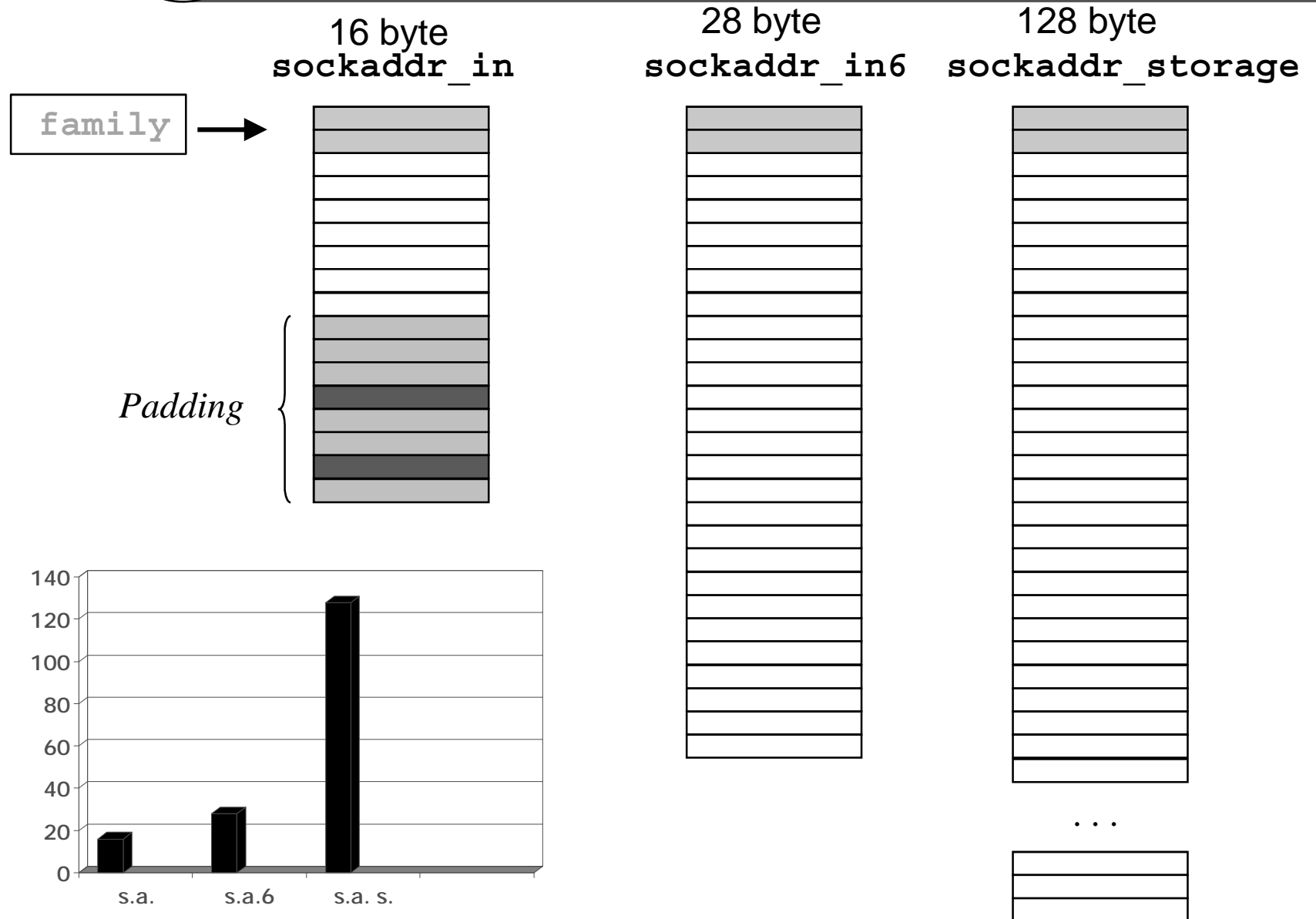
To communicate with node A or node C, **node B** has to disambiguate between them with a link-local address you need to specify the scope identification.



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.

Sockaddr_storage



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
```

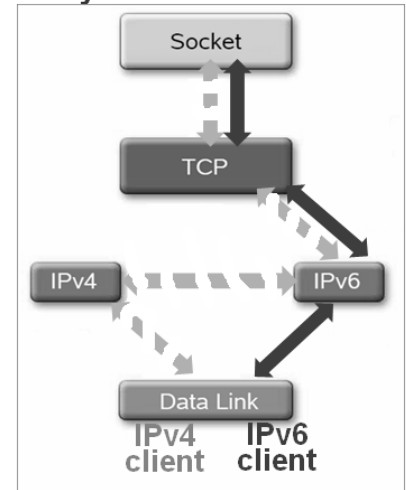
All of these new options are at the IPPROTO_IPV6 level (specifies the code in the system to interpret the option).

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

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

- the socket can be used to send and receive IPv6 packets only.
- by default is turned off.

Only one IPv6 socket:



```
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 usage 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, off; 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));
#else
    off=0;
    setsockopt(s0, IPPROTO_IPV6, IPV6_V6ONLY, &off, sizeof(off));
#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);
}
```

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
```

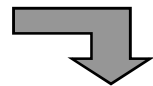
Using **USE_IPV6_V6ONLY**

```
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
```

- **1) If IPV6_V6ONLY = 1**
 - a)The socket will only accept IPv6 connections
 - b)You can create another IPv4 socket on the same port

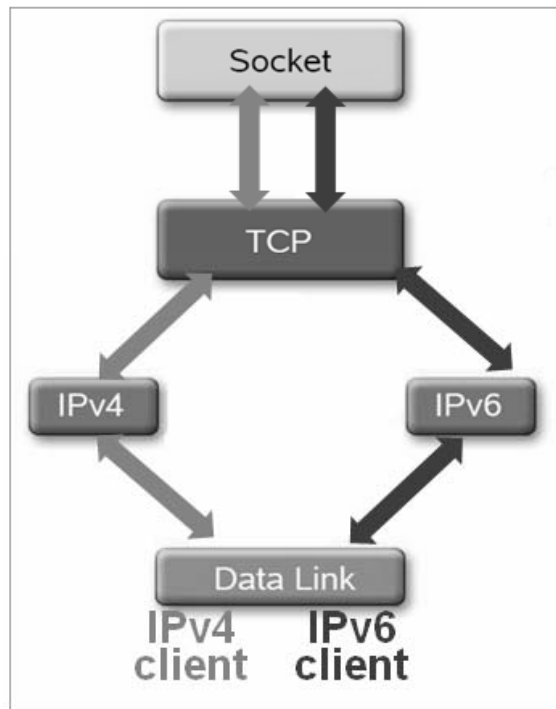
- **2) If IPV6_V6ONLY = 0**
 - a)The socket will accept both IPv4 and IPv6 connections
 - b)You cannot create another IPv4 socket on the same port (it would fail with error EADDRINUSE)

If you don't set or unset IPV6_V6ONLY in setsockopt in your program its value will be the one at /proc/sys/net/ipv6/bindv6only (unpredictable for applications) so you risk 1a or 2b

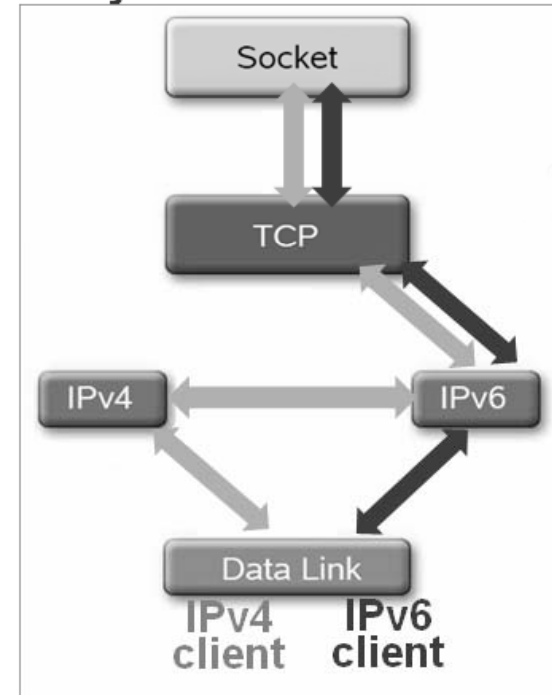
“Two sockets” and “Only one IPv6 socket”

In the following slides implementation examples of the “Two sockets” and “One IPv6 socket” scenarios are presented

Two sockets:



Only one IPv6 socket:

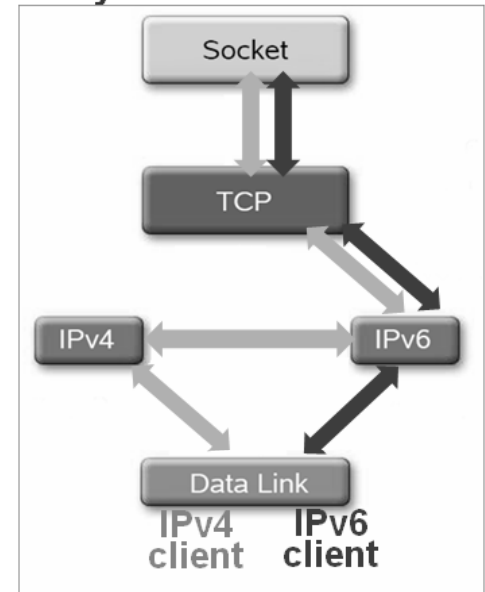


Only one IPv6 socket

```
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);
```

Only one IPv6 socket:



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

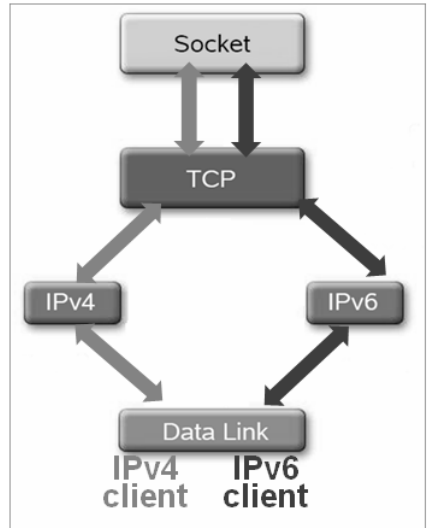
Two Sockets (1/3)

```

...
ServSock[0] = socket(AF_INET6, SOCK_STREAM, PF_INET6);
ServSock[1] = socket(AF_INET, SOCK_STREAM, PF_INET);
...
setsockopt(s[0], IPPROTO_IPV6, IPV6_V6ONLY, &on, sizeof(on))
...
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);
    ...
}

```

Two sockets:



```
$ 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

```

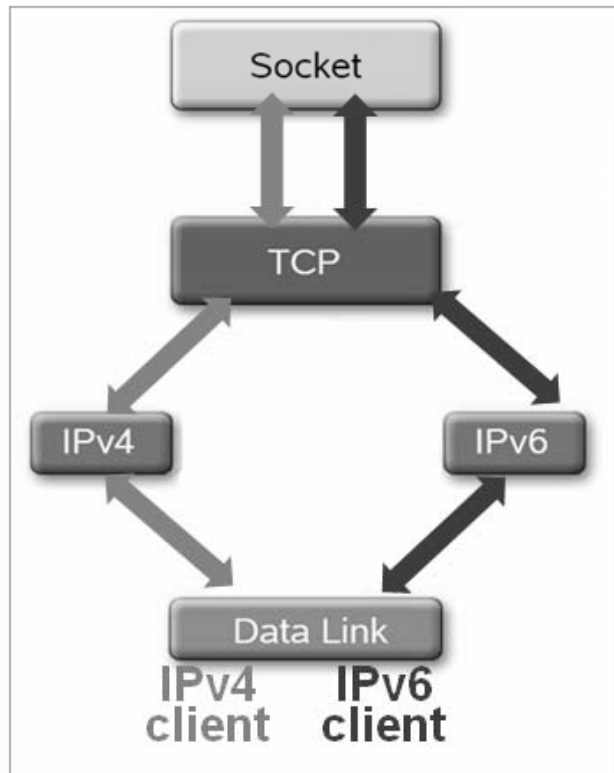

Two Sockets (2/3)

```

...
ServSock[0] = socket(AF_INET6, SOCK_STREAM, PF_INET6);
ServSock[1] = socket(AF_INET, SOCK_STREAM, PF_INET);
...
setsockopt(s[0], IPPROTO_IPV6, IPV6_V6ONLY, &on, sizeof(on));
...
bind(ServSock[0], AI0->ai_addr, AI0->ai_addrlen);
      bind(ServSock[1], AI1->ai_addr, AI1->ai_addrlen);

```

Two sockets:



```

...
    );
    , &SockSet)) {
    ket [0], (LPSOCKADDR)&Fron
    , &SockSet)) {
    ket [1], (LPSOCKADDR)&Fron

```

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.

```

...
02
5002 0.0.0.0:* LISTEN 3720/a.out
:::* LISTEN 3720/a.out

```

Previous Example Code Output:

CLIENT

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



SERVER

```
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
```

Only one IPv6 Socket

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
```

Two Socket

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
```

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

```
unsigned 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:

```
#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);
```



NEW

- **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 transparent programming.**
- **Network programmers should write code without a-priori assuming a specific IP version (IPv4 or IPv6)**
- **According to this new approach the following functions have been defined:**
 - `getaddrinfo`
 - `getnameinfo`
- **For Network Transparent Programming it is important to pay attention to:**
 - **Use of name instead of address**
 - **Avoid the use of hard-coded numerical addresses**
 - **Use `getaddrinfo` and `getnameinfo` functions.**

The **gethostbyname()** for IPv4 and **gethostbyname2()** 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

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);
```

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

int getaddrinfo(...)

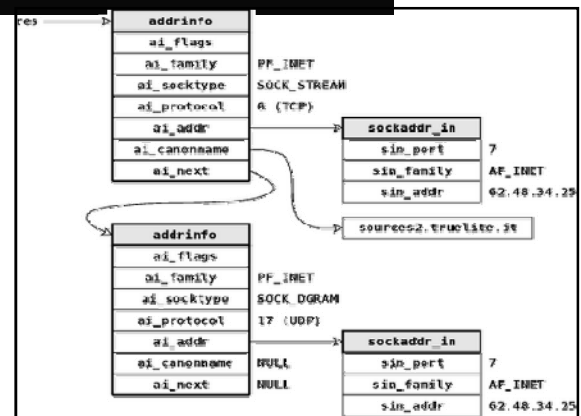
Function returns:
0 for success
not 0 for error
(see *gai_strerror*)

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



The caller can set only these values in the *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.


```

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).

```
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.

```

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.

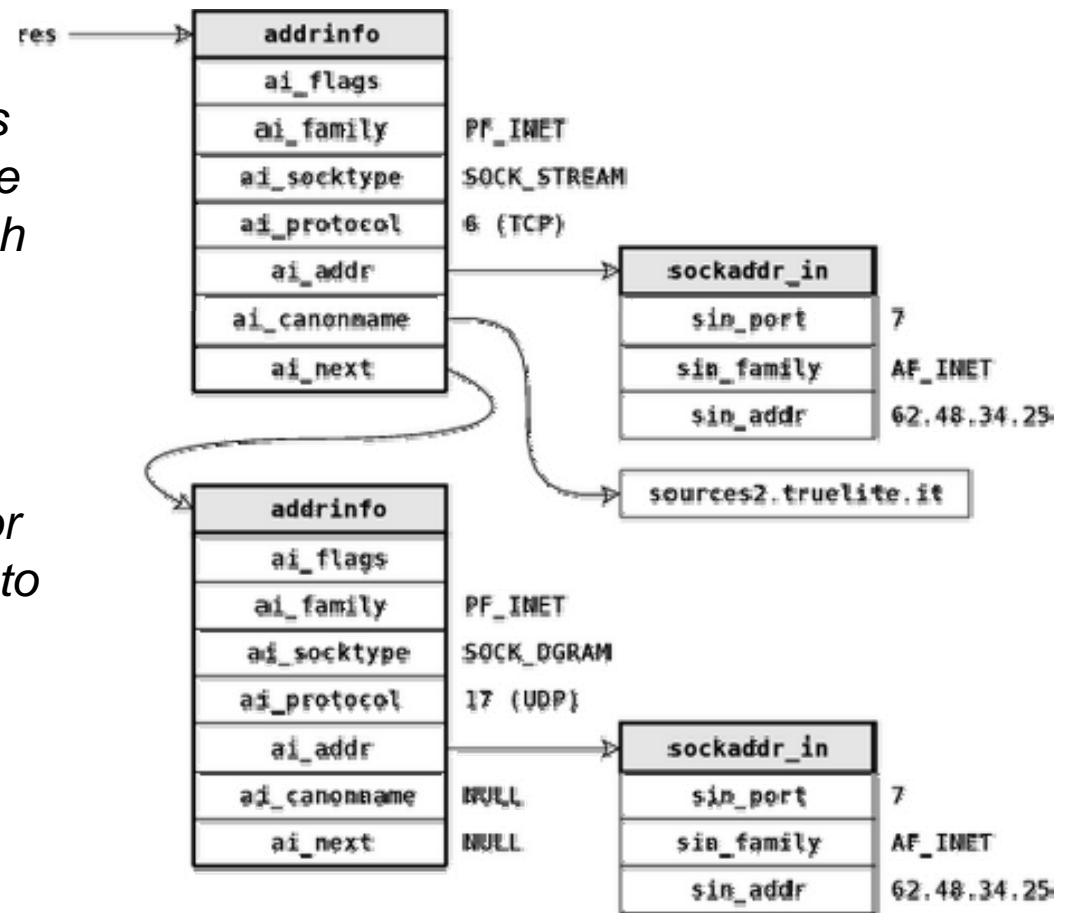
AI_ADDRCONFIG

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.

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`).



```

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.

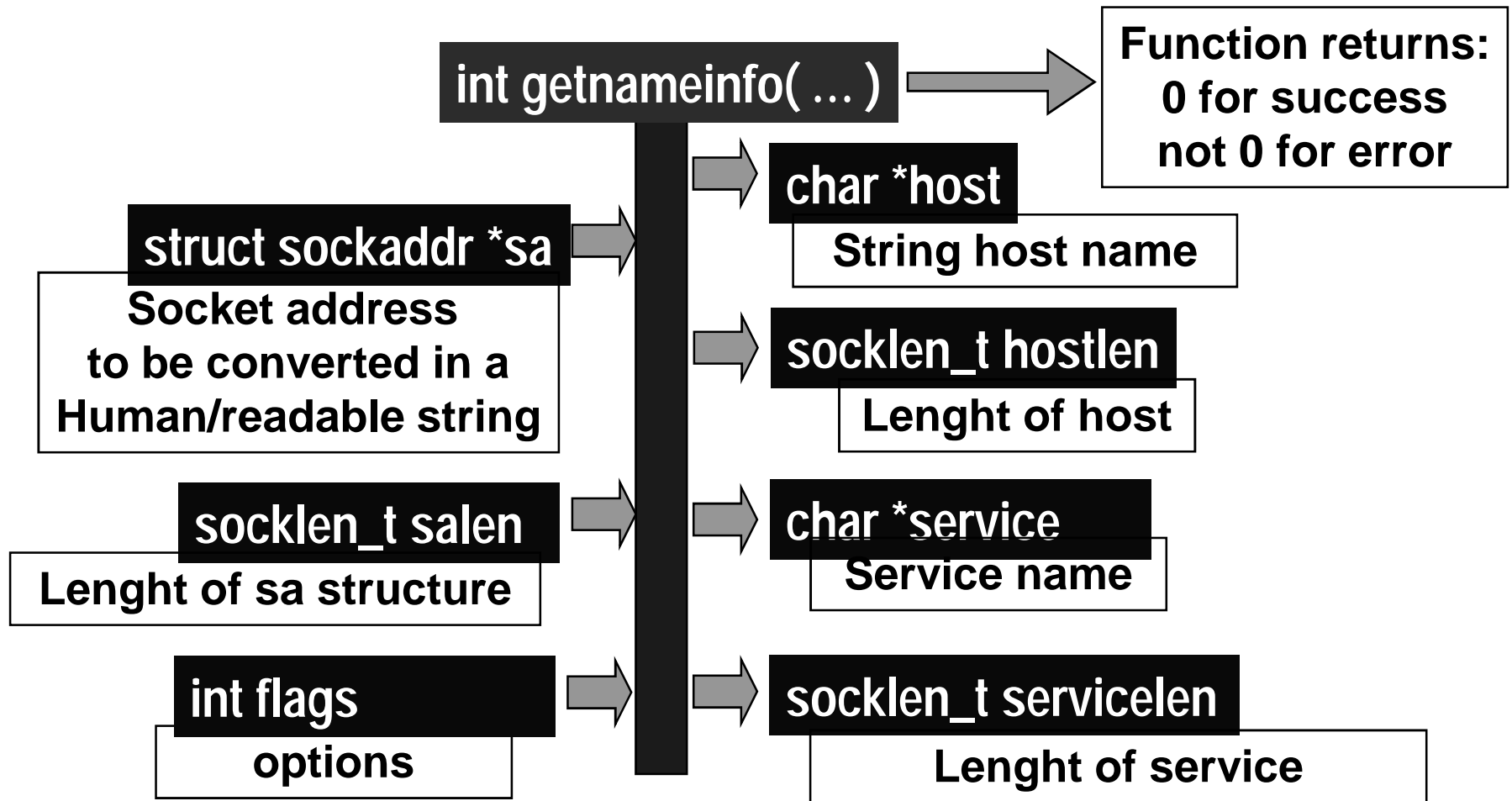
ai_addr is a pointer to a socket address structure.

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).

Nodename and Service Name Translation

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



```
#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.
- [...]

- **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.


```

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);
}

```

```

    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.ai_socktype = 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);
}
    
```

Introduction to IPv6 Programming In Perl

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
```

```
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
```

```
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
```

```
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

```
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.


```

use Socket;
use Socket6;
@res = getaddrinfo('hishost.com', 'daytime', AF_UNSPEC, SOCK_STREAM);
$family = -1;
while (scalar(@res) >= 5) {
    ($family, $socktype, $proto, $saddr, $scanonname, @res)=@res;
    ($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";
}

```

```

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, $canonicalname, @res) = @res;
    ($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";}

```

```
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 [...]
 [...]
```

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 ...
```

```
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.

```
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 **deprecated** in RFC3493: the getaddrinfo function should be used instead.

IPv6 Programming in Python in 2 slides: just a few hints

- **Python supports IPv6 since v2.3 on Linux/Solaris**
- **Key IPv6 functions are similar to C:**
 - **getnameinfo, getaddrinfo**
- **Drawbacks of using Python for IPv6:**
 - Most of any existing networking code needs modifications in order to be IPv6 compliant. However these modifications are easy.
 - There is no IPV6_V6ONLY symbol defined in Python (yet). You have to define it manually.

Purpose of the function call	function
Get the list of addresses to listen on	getaddrinfo()
Create the server socket	socket()
Bind the server socket	bind()
Listen on the server socket	listen()
Choose if the IPv6 socket should accept IPv4 connections or not	setsockopt(..., IPV6_V6ONLY, ...)
Accept a client connection	accept()

Purpose of the function call	function
Return the list of addresses to connect to the server host	getaddrinfo()
Create the client socket	socket()
Get a character string representing an IP address	getnameinfo (... , NI_NUMERICHOST)
Connect to the server	connect()

Introduction to IPv6 Programming In Java

- **Java APIs are already IPv4/IPv6 compliant.**
- **IPv6 support in Java is available since**
 - 1.4.0 in Solaris and Linux machines
 - 1.5.0 for Windows XP and 2003 server.
- **IPv6 support in Java is implicit and transparent.**
- **Indeed no source (nor bytecode!) code change 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.

- **IPv4-mapped address has a meaning only at implementation of a dual-protocol stack and it is never returned**
 - The corresponding plain IPv4 address is returned instead
- **For new applications IPv6-specific new classes and APIs can be used.**

- Advantage: Most of any existing networking code is already IPv6 compliant.
- Advantage: The sample Java code is shorter because some important functionality is handled by the Java environment itself in a transparent way.
- Drawback: The parameter **`/proc/sys/net/ipv6/bindv6only`** **must be set to 0** on the OS where the programs are run. Otherwise:
 - A server will not accept IPv4 clients
 - A client will fail to connect to an IPv4 server (it will say “Network is unreachable”!)
 - This is reported in the Java bug database.

- Based on the **ServerSocket** class

```
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) { [...]
```

- Based on the **Socket** class

```
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) { [...] }
```

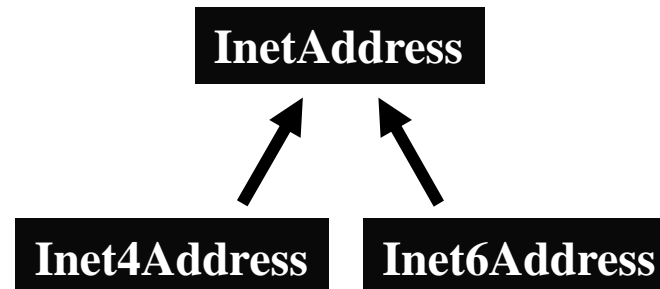
- **This class represents an IP address.**
It provides:
 - address storage.
 - name-address translation methods.
 - address conversion methods.
 - address testing methods.

Inet4Address and Inet6Address

In J2SE 1.4, the InetAddress has been 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
```



- The two types of addresses, IPv4 and IPv6, can be distinguished by the Java class 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 the **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, it has to become aware of the **Inet4Address** and **Inet6Address** classes.

```
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.

```
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

```
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
```

```
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
```

```
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
```

```
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 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
```


To the **InetAddress** class new methods have been added:

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

Inet6Address has one further method w.r.t. **Inet4Address**:

```
Inet6Address.isIPv4CompatibleAddress ()
```

```

Enumeration netInter = NetworkInterface.getNetworkInterfaces();
while ( netInter.hasMoreElements() )
{
    NetworkInterface ni = (NetworkInterface)netInter.nextElement();
    System.out.println( "Net. Int. : "+ ni.getDisplayName() );
    Enumeration addr = ni.getInetAddresses();
    while ( addr.hasMoreElements() )
    {
        Object o = addr.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());
        }
    }
}
    
```

```

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

```

```
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** applications to connect to, and accept connections from, **both IPv4 and IPv6 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 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 -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
```

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);
```

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

If IPv6 is available on the OS, 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 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.


```
//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
```

High level programming libraries

- **Goal**
- **High-level Python**
- **High-level Perl**
- **High-level C/C++**
- **Other High-level techniques**

- **High-level code is shorter**
- **High-level code is easier to read**
- **High-level code is not affected (or little) by changes in the low-level API, like the changes required now to support IPv6.**

- **High-level networking libraries exist in Python, but they have several drawbacks.**
- **An example: the class ThreadingTCPServer**

– Simple usage:

```
...
ThreadingTCPServer.address_family = socket.AF_INET6
server = ThreadingTCPServer("", port), <handler>)
server.serve_forever()
```

– We see that:

- This class is IPv4-only by default, so we have to indicate the IPv6 address family.
- Therefore our code is not address-family independent (and will fail if IPv6 is not available on the operating system).
- The class does not cover all IPv6 aspects: the IPV6_V6ONLY option should be unset in order to allow IPv4 connection to the IPv6 server socket.

- Advanced usage:
 - It is possible to create a sub-class of ThreadingTCPServer which will solve these three problems.
 - The code should then use this sub-class instead of ThreadingTCPServer.
 - See the report on IPv6 programming with C/C++, Perl, Python and Java at <https://edms.cern.ch/document/971407> for more information.

- **High-level networking libraries exist in Perl, but most of them are not IPv6 compliant.**
- **The most famous high-level IPv6 compliant library is `IO::Socket::INET6`.**
 - This library currently provides very interesting functionality in mixed IPv4 and IPv6 environments.
 - But according to the README file of the package, the library will surely become IPv6-only in the future. If this happens, then using this library in a mixed IPv4 and IPv6 environment will be at least very inefficient, if not impossible.

- **The new `boost::asio` library (available since boost 1.35) may be used in an IPv6 compliant way.**

- For clients, refer to the “Synchronous TCP daytime client” sample code of the `boost::asio` tutorial which is IPv6 compliant.
- For servers, the only subtle point is to avoid the constructor of `tcp::acceptor` which automatically opens, binds and listens, because we need to set the `IPV6_V6ONLY` socket option to 0; and setting this option can only be done after the 'open' and before the 'bind'.

How to set this socket option with boost:

```
acceptor.set_option(ip::v6_only(false));
```

- **For more information see the report at <https://edms.cern.ch/document/935729>**

- **For a socket server with basic TCP functionality, it is possible to build an xinetd-based service:**
 - Xinetd will handle the low-level networking code itself.
 - Xinetd can handle IPv6 compliance of a service by just adding the option “**flags=IPv6**” in the file **/etc/xinetd.d/<service_name>**.
In this case xinetd will create an IPv6 socket which will accept both IPv4 and IPv6 connections.