Implementing IPv6 Applications

Intro

- We will explain how to implement IPv6 applications
  - We assume knowledge of writing IPv4 applications
- We also look at porting IPv4 applications to IPv6
- We look at writing/porting applications written in C , Perl, Java and PHP
- We consider common application porting issues
- We look at standards and recommendations

Enabling application for IPv6

- Most IPv4 applications can be IPv6 enabled
  - Appropriate abstraction layers used
- Providing ‘Dual stack’ IPv4 and IPv6 is best
  - Run-time (preferable) or compile-time network mode (v6 and/or v4)
- All widely used languages are IPv6-enabled
  - E.g. C/C++, Java, Python, Perl
  - Some languages make it particularly easy
    - e.g Java
- Benefiting from IPv6 is a little more difficult
  - Though most functionality is the similar to IPv4
  - Add special functionality for IPv6 features
- IPv4 and IPv6 APIs have largely converged
- 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

Precautions for Dual Stack

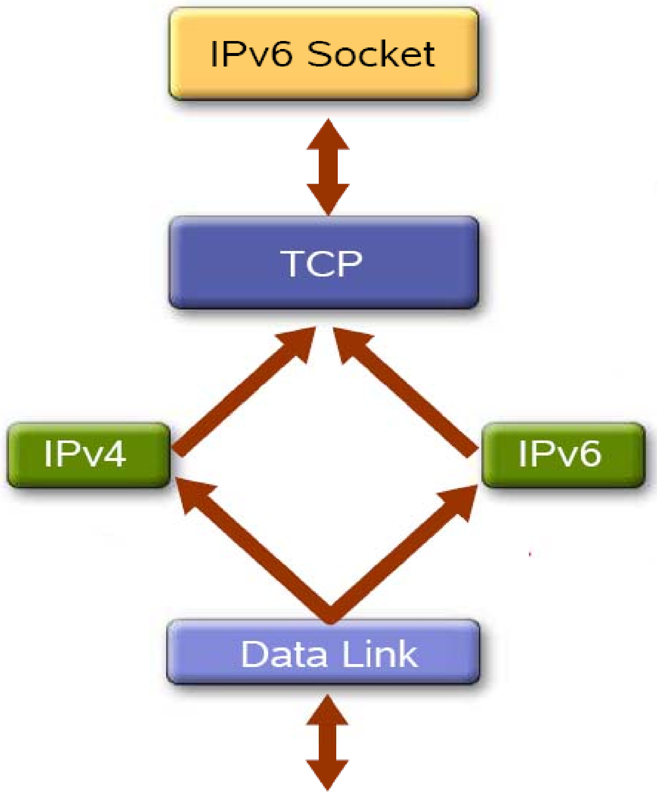
- Avoid any explicit use of IP addresses
  - Normally do Call by Name
- Ensure that calls to network utilities are concentrated in one subroutine
- Ensure that libraries and utilities used support both stacks
- Do not request functions that would not exist in both stacks
  - E.g. IPsec, MIP, Neighbour Discovery may vary

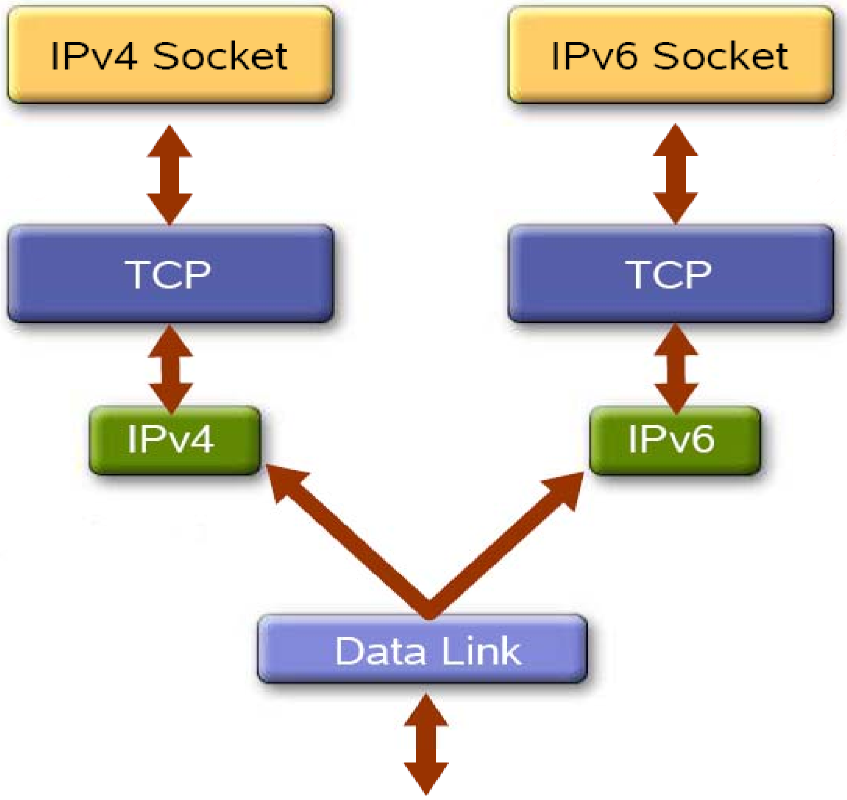
Dual stack configurations

Both IPv4 and IPv6 stacks will be available during the transition period

Dual network stack machine will allow to provide a service both for IPv4 and IPv6

2 different implementations of network stack

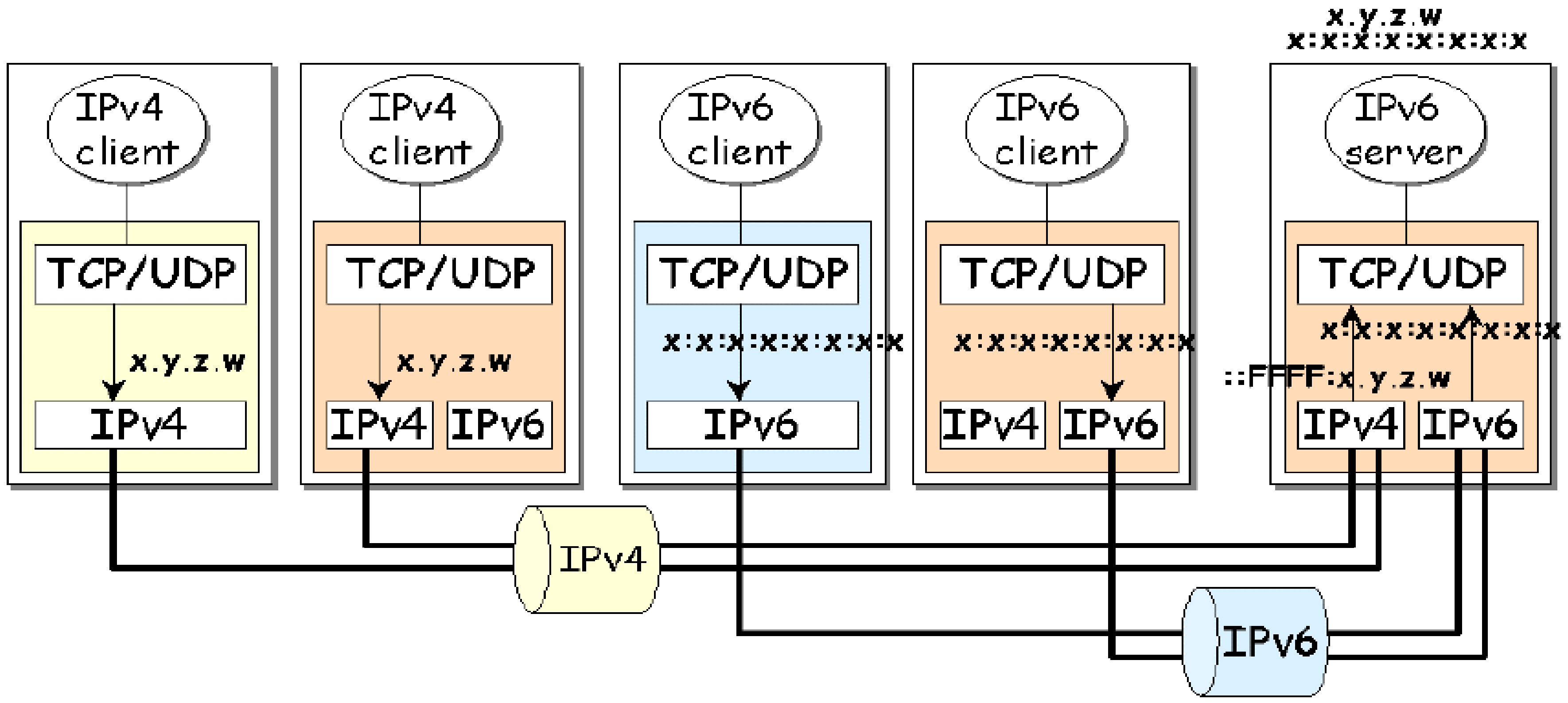




Source : Rino Nucara, GARR, EuChinaGRID IPv6 Tutorial

Mapping IPv4 address in IPv6

IPv6/IPv4 Clients connecting to an IPv6 server at dual stack node → 1 socket

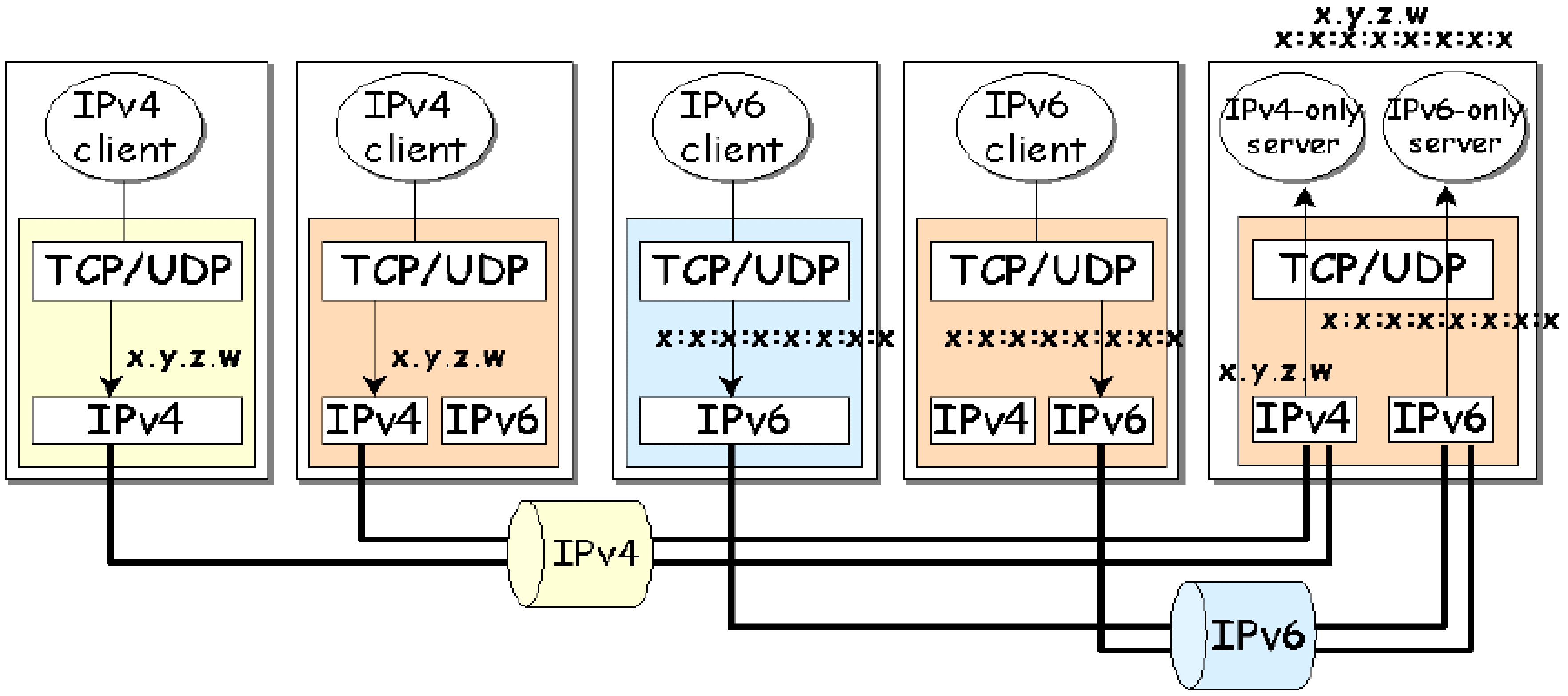


🟧	*Dual Stack*
🟨🟦	*Single IPv4 or IPv6 stacks*

Source : Programming guidelines on transition to IPv6 T. P de Miguel, E. M. Castro

IPv4-only and IPv6-only

IPv6/IPv4 Clients connecting to an IPv4-only server and IPv6 only server at dual stack node → 2 sockets



🟧 *Dual Stack or separated stack* 🟨🟦 *Single IPv4 or IPv6 stacks*

Source : Programming guidelines on transition to IPv6 T. P de Miguel, E. M. Castro

New Applications

- Simplified by writing apps using a high-level language
  - E.g. JAVA seamlessly supports dual stack
- Design the application in a protocol independent fashion
- Ensure both protocols will be simultaneously operable

Porting

- The hardest part is often parsing of config files and internal handling of addresses, not the socket code itself
- You may need to write code that works with both old API and new.
- It's not uncommon that large applications have some duplication of network code. When porting it might be a good idea to fix this
- If most parts are written in say Java, and small parts in say C, try to rewrite C part to be in Java or at least make sure that I/O is concentrated in certain regions

Porting – Abstract Network Layer

- Separate the transport module from the rest of application functional modules
  - application independent on the network system used
  - if the network protocol is changed, only the transport module should be modified
- The transport module should provide the communication channel abstraction with basic channel operations and generic data structures to represent the addresses
- If a new network protocol is added, application developers only need to implement a new instance of the channel abstraction which manages the features of this new protocol

IPv6 literal addresses in URL’s

From RFC 2732

Literal IPv6 Address Format in URL's Syntax To use a literal IPv6 address in a URL, the literal address should be enclosed in "[" and "]" characters. For example the following literal IPv6 addresses:

FEDC:BA98:7654:3210:FEDC:BA98:7654:3210

3ffe:2a00:100:7031::1

::192.9.5.5

2010:836B:4179::836B:4179

would be represented as in the following example URLs:

http://[FEDC:BA98:7654:3210:FEDC:BA98:7654:3210]:80/index.html

http://[3ffe:2a00:100:7031::1]

http://[::192.9.5.5]/ipng

http://[2010:836B:4179::836B:4179]

Effects on higher layers

- Affects anything that reads/writes/stores/passes IP addresses
  - Most IETF protocols have been updated for IPv6 compliance
- Bigger IP header must be taken into account when computing max payload sizes
- Packet lifetime no longer limited by IP layer
  (it never was, anyway!)
- New DNS record type: AAAA
- DNS lookups may give several v4 and/or v6 addresses
  - Applications may need to deal with multiple addresses
- Advanced mobility
  - Mobile IPv6, Network Mobility (NEMO)

Miscellaneous issues (1)

- URL format for literal IPv6 addresses (RFC 2732)
  - http://[2001:db8:dead:beef::cafe]:80/
- Entering IP addresses more difficult
  - Especially on a numeric/phone keypad
- Better to pass names than addresses in protocols, referrals etc. They can look up addresses in DNS and use what they need
  - If a dual-stack node can’t pass fqdn in protocol (referrals, sdp etc), it should be able to pass both IPv4 and IPv6 addresses
  - Important that other clients can distinguish between IPv4 and IPv6 belonging to same host, or being two different hosts
- In IPv4, we used variable-length subnet masks

Miscellaneous issues (2)

- Hosts will typically have several addresses
  - Dual-stack hosts both IPv4 and IPv6
  - May have multiple IPv6 addresses
    - Multihomed or global prefix + ULA for internal
    - Renumbering
- Addresses may change over time
  - Privacy addresses, e.g. every 24 hours
  - When renumbering

Conclusion

- Many existing applications are available in IPv6
  http://en.wikipedia.org/wiki/Comparison_of_IPv6_application_support
  http://ipv6.niif.hu/m/ipv6_apps_db
- Porting applications to IPv6 is straightforward
  - Provided certain guidelines are followed
- Heterogeneous environments provide the most challenges

C IPv6 API

Basic IPv6 socket programming

- Will go through API within RFC 3493 (Basic Socket Interface Extensions for IPv6) and give recommendations on how to use it
- The Advanced API is specified in RFC 3542
- There is also POSIX, or The Single UNIX Specification, Version 3 at http://www.unix.org/version3/online.html
- RFC and POSIX are roughly the same with some minor differences. Useful to look at both

RFC 3493 - Basic Socket Interface Extensions for IPv6

- Basic Socket Interface Extensions for IPv6
  - RFC 2553's revision version
  - Support basic socket APIs for IPv6
  - Introducing a minimum of change into the system and providing complete compatibility for existing IPv4 applications
- TCP/UDP application is Required using IPv6
  - New socket address structure
  - New address conversion functions
  - Some new socket options
- Extensions for advanced IPv6 features are defied in another document.
  - RFC 2292: Advanced Socket API for IPv6

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 packet information such as the IPv6 header and the extension headers.
- Advanced APIs are also used to extend the capability of IPv6 raw socket

Socket API Changes

- Name to Address Translation Functions
- Address Conversion Functions
- Address Data Structures
- Wildcard Addresses
- Constant Additions
- Core Sockets Functions
- Socket Options
- New Macros

IPv6 Address Family and Protocol Family

New address family name

- AF_INET6 is defined in <sys/socket.h>
  - Distinguishes between sockaddr_in and sockaddr_in6
  - AF_INET6 is used as a first argument to the socket()
- New protocol family name
  - PF_INET6 is defined in <sys/socket.h>
    - #defined PF_INET6AF_INET6

IPv4	IPv6
AF_INET	AF_INET6
PF_INET	PF_INET6

IPv6 Address Structure

New address structure

- in6_addr structure
- is defined in <netinet/in.h>
- structin6_addr {

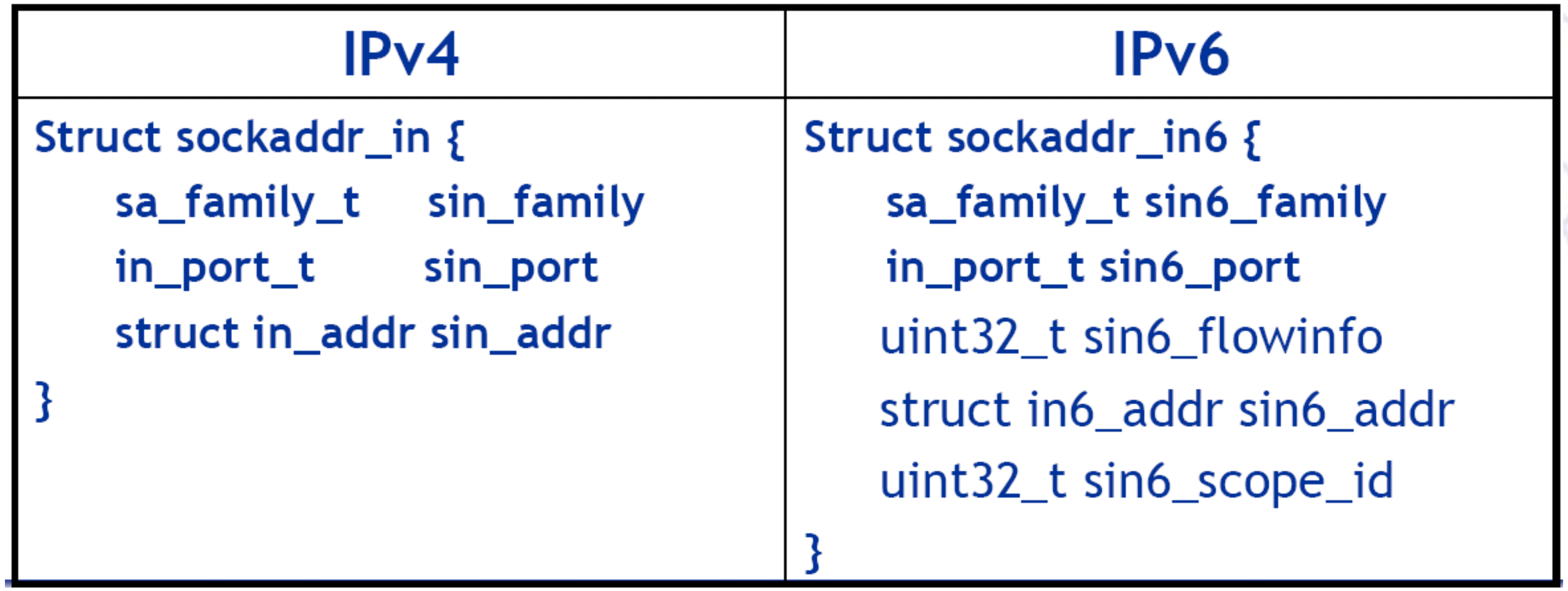
```
uint8_t s6_addr[16]; /* IPv6 address */
};
```

IPv4	IPv6

Struct in_addr {
    unsigned int s_addr
}
	Struct in6_addr {
	    uint8_t s6_addr[16]
	}

Address Structures

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



IPv4	IPv6
Struct sockaddr_in {	Struct sockaddr_in6 {
sa_family_t sin_family	sa_family_t sin6_family
in_port_t sin_port	in_port_t sin6_port
struct in_addr sin_addr	uint32_t sin6_flowinfo
}	struct in6_addr sin6_addr
	uint32_t sin6_scope_id
	}

Important definitions

```
•   PF_INET6, AF_INET6 (PF_INET, AF_INET for IPv4)
•   struct in6_addr {
•             uint8_t  s6_addr[16];      /* IPv6 address */
•       };
•   struct sockaddr_in6 {
•             sa_family_t     sin6_family;    /* AF_INET6 */
•             in_port_t       sin6_port;      /* transport layer port # */
•             uint32_t        sin6_flowinfo;  /* IPv6 flow information */
•             struct in6_addr sin6_addr;      /* IPv6 address */
•             uint32_t        sin6_scope_id;  /* set of interfaces for a scope */
•        };
```

- `sin6_flowinfo` not used (yet)
- Will discuss `sin6_scope_id` later

```
•   struct sockaddr_storage {
•             sa_family_t     ss_family;    /* address family */
•             char ss_pad... /* padding to make it large enough */
•      };
```

- Used when we need a struct to store any type of sockaddr

Address Data Structure:sockaddr_storage

- 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.

Pass Addresses

- 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 Addresses

- IPv4

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

- IPv6:

```
Astruct 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 loopback address

- The IPv6 loopback address is provided in two forms: a global variable and a symbolic constant
- 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)
. . .
```

- Second way is a symbolic constant named IN6ADDR_LOOPBACK_INIT:

```
#define IN6ADDR_LOOPBACK_INIT {{{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1}}}
struct in6_addr loopbackaddr = IN6ADDR_LOOPBACK_INIT;
```

Socket Options

A number of new socket options are defined for IPv6:

- IPV6_UNICAST_HOPS
- IPV6_MULTICAST_HOPS
- IPV6_MULTICAST_LOOP
- IPV6_JOIN_GROUP
- IPV6_LEAVE_GROUP
- IPV6_V6ONLY

- 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.
- 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.

Example Code

Without USE_IPV6_V6ONLY and the IPv4-mapped address is supported by the system:

```
telnet ::1 5001          →  Accept a connection from ::1
telnet 127.0.0.1 5001    →  Accept a connection from ::ffff:127.0.0.1
```

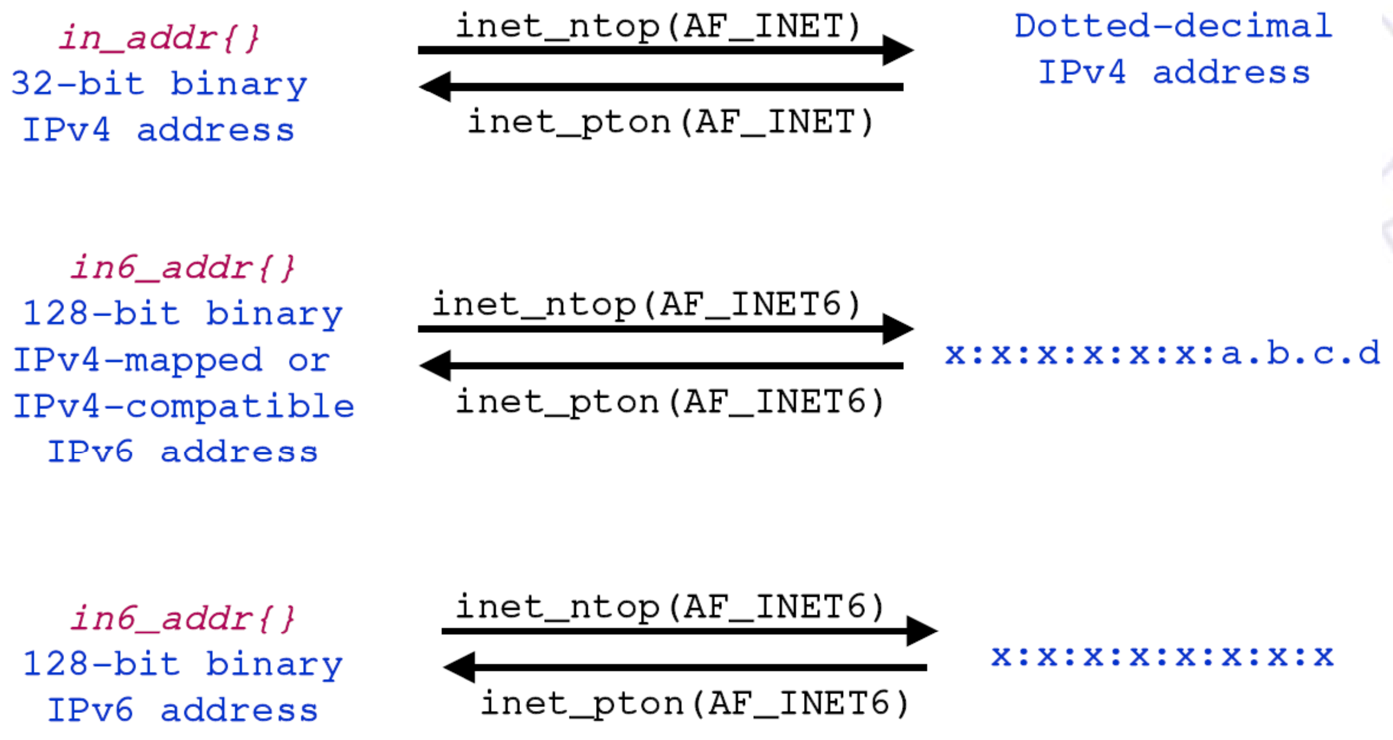
With USE_IPV6_V6ONLY

```
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
```

Address Conversion Functions

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



Core Socket Functions

Core APIs

- Use IPv6 Family and Address Structures
- socket() Uses PF_INET6

Functions that pass addresses

- bind()
- connect()
- sendmsg()
- sendto()

Functions that return addresses

- accept()
- recvfrom()
- recvmsg()
- getpeername()
- getsockname()

All the above function definitions are unchanged due to use of struct sockaddr and address length

Name to Address Translation

getaddrinfo()

- Node name to address translation is done in a protocol independent way
- 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.
- is very flexible and has several modes of operation.
- It returns a dynamically allocated linked list of addrinfo structures
- contains useful information (for example, sockaddr structure ready for use).

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

freeaddrinfo()

- it frees addrinfo structure returned by getaddrinfo(), along with any additional storage associated with those structures

  *void freeaddrinfo(struct addrinfo *ai);*

getnameinfo()

- getnameinfo is the complementary function to getaddrinfo: it takes a socket address and returns a character string describing the host and another character string describing the service.

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

Scope ID

- When using link local addresses a host with multiple interfaces need to know which interface the address is for
- This is what sockaddr_in6's sin6_scope_id is for
- getaddrinfo() can automatically fill this in when called with e.g. "www.kame.net%eth0" or "fe80::1234:5678%eth0"
- This notation is standardized, but the naming of interfaces are not

Porting application to IPv6

- To port IPv4 applications in a multiprotocol environment, developers should look out for these basic points
  - Use DNS names instead of numerical addresses
  - Replace incidental hardcoded 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, RFC2732 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

- Applications in a dualstack 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 cannot communicate directly, even in dualstack hosts. There are some different methods to implement such communication, but they are out of scope of this document.

Rewriting an application

- To rewrite an application with IPv6 compliant code, the first step is to find all IPv4dependent 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*·
- Developers should pay attention to hardcoded numerical address, host names, and binary representation of addresses.
- It is recommended to made 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.

Simple old IPv4 TCP client

```
/* borrowed from http://www.ipv6.or.kr/summit2003/presentation/II-2.pdf */

struct hostent *hp;
int i, s;
struct sockaddr_in sin;
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
hp = gethostbyname("www.kame.net");
for (i = 0; hp->h_addr_list[i]; i++) { /* not so common to loop through all */
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_len = sizeof(sin); /* only on BSD */
    sin.sin_port = htons(80);
    memcpy(&sin.sin_addr, hp->h_addr_list[i],
    hp->h_length);
    if (connect(s, &sin, sizeof(sin)) < 0)
        continue;
    break;
}
```

Simple IPv4/IPv6 TCP client

```
/* borrowed from http://www.ipv6.or.kr/summit2003/presentation/II-2.pdf */
struct addrinfo hints, *res, *res0;
int error, s;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo("www.kame.net", "http", &hints, &res0);
if (error)
      errx(1, "%s", gai_strerror(error));
/* res0 holds addrinfo chain */
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;
      error = connect(s, res->ai_addr, res->ai_addrlen);
      if (error) {
              close(s);
              s = -1;
              continue;
      }
      break;
}
freeaddrinfo(res0);
if (s < 0)
      die();
```

Simple old IPv4 TCP server

```
/* from http://www.ipv6.or.kr/summit2003/presentation/II-2.pdf */

int s;
struct sockaddr_in sin;
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_len = sizeof(sin); /* only on BSD */
sin.sin_port = htons(80);
if (bind(s, &sin, sizeof(sin))>= 0)
    exit(1);
listen(s, 5);
```

Simple IPv4/IPv6 TCP server (1/2)

```
/* from http://www.ipv6.or.kr/summit2003/presentation/II-2.pdf */

struct addrinfo hints, *res, *res0;
int s, i, on = 1;
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);
if (error) {
    fprintf(stderr, "%s", gai_strerror(error));
    exit(1);
}
/* res0 has chain of wildcard addrs */
```

Simple IPv4/IPv6 TCP server (2/2)

```
/* borrowed from http://www.ipv6.or.kr/summit2003/presentation/II-2.pdf */
i = 0;
for (res = res0; res; res = res->ai_next) {
     s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
     if (s < 0)
             continue;
#ifdef IPV6_V6ONLY
     if (res->ai_family == AF_INET6 && setsockopt(s, IPPROTO_IPV6, IPV6_V6ONLY, &on,
     sizeof(on)) < 0) {
             close(s);
             continue;
     }
#endif
     if (bind(s, res->ai_addr, res->ai_addrlen) >= 0) {
             close(s);
             continue;
     }
     listen(s, 5);
     socktable[i] = s;
     sockfamily[i++] = res->ai_family;
}
freeaddrinfo(res0);
if (i == 0)
     errx(1, "no bind() successful");
/* select()/poll() across socktable[] */
```

One socket server example (1/2)

With support for mapped addresses you can use a single IPv6 socket

Also single v4 or v6 socket if you only need to support one family or take family as an argument on startup

```
/* from http://www.ipv6.or.kr/summit2003/presentation/II-2.pdf */
int af = AF_INET6; /* or AF_INET */
struct addrinfo hints, *res;
int s, i, on = 1;
memset(&hints, 0, sizeof(hints));
hints.ai_family = af;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
error = getaddrinfo(NULL, "http", &hints, &res);
if (error)
    exit(1);
if (res->ai_next) {
    fprintf(stderr, "multiple addr");
    exit(1);
}
/* res has chain of wildcard addrs */
```

One socket server example (2/2)

```
/* borrowed from http://www.ipv6.or.kr/summit2003/presentation/II-2.pdf */
s = socket(res->ai_family, res->ai_socktype,
res->ai_protocol);
if (s < 0)
    exit(1);
#ifdef IPV6_V6ONLY
/* on here for v6 only, set off for mapped addresses if applicable */
if (res->ai_family == AF_INET6 && setsockopt(s, IPPROTO_IPV6, IPV6_V6ONLY,
    &on, sizeof(on)) < 0) {
    close(s);
    continue;
}
#endif
if (bind(s, res->ai_addr, res->ai_addrlen) < 0)
    exit(1);
listen(s, 5);
freeaddrinfo(res);
```

Java IPv6 API

IPv6 Support in Java

- **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 hardcoded 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 nonspecific functions in the address translation.

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) { [..] }
```

IPv6 Support in Java

- For new applications Ipv6 specific new classes and APIs can be used.
- **InetAddress**
  - This class represents an IP address. It provides address storage, name-address translation methods, address conversion methods, as well as address testing methods.
  - In J2SE 1.4, this class is extended to support both IPv4 and IPv6 addresses.
  - Utility methods are added to check address types and scopes.

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
```

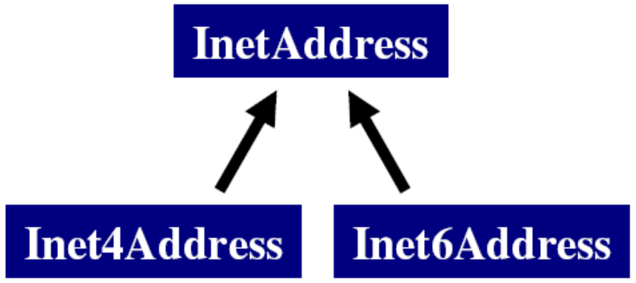
```
InetAddress[ ] alladr=ia.getAllByName("www.kame.net");
for(int i=0;i<alladr.length;i++) { System.out.println(  alladr[i]  );  }
print:
  www.kame.net/203.178.141.194
  www.kame.net/2001:200:0:8002:203:47ff:fea5:3085
```

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 IPv6only method, or when it cares to know the class types of the IP address, will it ever become aware of Inet4Address and Inet6Address.

Inet4Address and Inet6Address

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



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.

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 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.

IPv6 Networking Properties

The new methods introduced are:

InetAddress:

- isAnyLocalAddress
- isLoopbackAddress
- isLinkLocalAddress
- isSiteLocalAddress
- isMCGlobal
- isMCNodeLocal
- isMCLinkLocal
- isMCSiteLocal
- isMCOrgLocal
- getCanonicalHostName
- getByAddr

Inet6Address:

- isIPv4CompatibleAddress

Code 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
```

PHP and IPv6

PHP and IPv6

- IPv6 is supported by default in PHP4.3.4 and PHP5.2.3 versions.
- Few functions have been defined to support IPv6.
- sparse documentation is provided for IPv6 programming
- 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.
- 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.

PHP and IPv6

- fsockopen
  - Initiates a socket connection to the resource specified by target. PHP supports targets in the Internet and Unix domains as described in Appendix P, List of Supported Socket Transports. A list of supported transports can also be retrieved using stream_get_transports().
- checkdnsrr
  - 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.
  - Note: AAAA type added with PHP 5.0.0
- gethostbyname
  - Returns the IP address of the Internet host specified by hostname or a string containing the unmodified hostname on failure.

Pear Net_IPv6 Package

- 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

Perl IPv6 API

IPv6 API of Perl5

- relying on the IPv6 support of underlying operating system
- you can write Perl applications with direct access to sockets
- new IPv6 API for DNS name resolution is important for seamless operation
- With simple API creating `sockaddr_in6` might be tedious
- There are two modules for Basic IPv6 API
  - Socket6
  - IO::Socket::INET6

Perl implementation of new IPv6 DNS + socket packing API

- Socket6 module **- available via CPAN**
- **implemented functions:**
- **`getaddrinfo()` - see usage later**
- **`gethostbyname2 HOSTNAME, FAMILY` - family specific gethostbyname**
- **`getnameinfo` NAME, [FLAGS] - see usage later**
- **`getipnodebyname HOST, [FAMILY, FLAGS]` - list of five elements - usage not recommended**
- **`getipnodebyaddr FAMILY, ADDRESS` - list of five elements - usage not recommended**
- **`gai_strerror ERROR_NUMBER` - returns a string of the error number**
- **`inet_pton FAMILY, TEXT_ADRESS` - text->binary conversion**
- **`inet_ntop FAMILY, BINARY_ADDRESS` - binary-> text conversion**

Perl implementation of new IPv6 DNS + socket packing API/2

- `pack sockaddr in6 PORT, ADDR` **- creating sockaddr_in6 structure**
- `pack_sockaddr_in6_all PORT, FLOWINFO, ADDR, SCOPEID` **- complete implementation of the above**
- `unpack sockaddr in6 NAME` **- unpacking sockaddr_in6 to a 2 element list**
- `unpack_sockaddr_in6_all NAME` **- unpacking sockaddr_in6 to a 4 element list**
- `in6addr_any` **- 16-octet wildcard address.**
- `in6addr_loopback` **- 16-octet loopback address**

Simple getaddrinfo() example

```
•   use Getopt::Std;
•   use Socket;
•   use Socket6;
•   use strict;

•   my $inet6 = defined(eval 'PF_INET6');

•   my %opt;
•   getopts(($inet6 ? 'chpsn46' : 'chpsn4'), \%opt);
•   if ($opt{'h'}){
•       print STDERR ("Usage: $0 [-h | [-c] [-n] [-p] [-s] [-4" .
•                         ($inet6 && "|-6") . "] [host [serv]]]\n" .
•                         "-h    : help\n" .
•                         "-c    : AI_CANONNAME flag\n" .
•                         "-n    : AI_NUMERICHOST flag\n" .
•                         "-p    : AI_PASSIVE flag\n" .
•                         "-s    : NI_WITHSCOPEID flag\n" .
•                         ($inet6 ? "-4|-6: PF_INET | PF_INET6" : "-4    : PF_INET") .
•                         "\n");
•       exit(4);
•   }
•   my $host = shift(@ARGV) if (@ARGV);
•   my $serv = shift(@ARGV) if (@ARGV);
•   die("Too many arguments\n") if (@ARGV);
•   die("Either -4 or -6, not both should be specified\n") if ($opt{'4'} && $opt{'6'});
```

Simple getaddrinfo() example/2

```
•   my $af = PF_UNSPEC;
•   $af = PF_INET if ($opt{'4'});
•   $af = PF_INET6 if ($inet6 && $opt{'6'});

•   my $flags = 0;
•   $flags |= AI_PASSIVE if ($opt{'p'});
•   $flags |= AI_NUMERICHOST if ($opt{'n'});
•   $flags |= AI_CANONNAME if ($opt{'c'});

•   my $nflags = NI_NUMERICHOST | NI_NUMERICSERV;
•   $nflags |= NI_WITHSCOPEID if ($opt{'s'});

•   my $socktype = SOCK_STREAM;
•   my $protocol = 0;

•   my @tmp = getaddrinfo($host, $serv, $af, $socktype, $protocol, $flags);
•   while (my($family,$socktype,$protocol,$sin,$canonname) = splice(@tmp, $[, 5)){
•       my($addr, $port) = getnameinfo($sin, $nflags);
•       print("family=$family, socktype=$socktype, protocol=$protocol, addr=$addr, port=$port");
•       print(" canonname=$canonname") if ($opt{'c'});
•       print("\n");
•   }
```

Object oriented Perl socket API

- **using basic socket API - sometimes complicated**
- **`IO::Socket::INET` makes creating socket easier - inherits all functions from `IO::Socket + IO::Handle`**
- `IO::Socket::INET6` **- generalisation of IO:Socket:INET to be protocol neutral - available from CPAN**
- **new methods:**
  - `sockdomain()` - Returns the domain of the socket - AF_INET or AF_INET6 or else
  - `sockflow ()` - Return the flow information part of the sockaddr structure
  - `sockscope ()` - Return the scope identification part of the sockaddr structure
  - `peerflow ()` - Return the flow information part of the sockaddr structure for the socket on the peer host
  - `peerscope ()` - Return the scope identification part of the sockaddr structure for the socket on the peer host

IO::Socket::INET6 examples

- **Trying to connect to peer trying all address/families until reach**

```
$sock = IO::Socket::INET6->new(PeerAddr => 'ipv6.niif.hu',
                               PeerPort => 'http(80)',
                               Multihomed => 1 ,
                               Proto    => 'tcp');
```

- **Connecting via IPv4 only - backward compatibility with `IO::Socket::INET`**

```
$sock = IO::Socket::INET6->new(PeerAddr => 'ipv6.niif.hu',
                               PeerPort => 'http(80)',
                               Domain => AF_INET ,
                               Multihomed => 1 ,
                               Proto    => 'tcp');
```

IO::Socket::INET6 examples /2

- **using literal ipv6 address**

```
$sock = IO::Socket::INET6->new(PeerAddr => '[::1]:25');
```

- **setting up a listening socket**

```
$sock = IO::Socket::INET6->new(Listen    => 5,
                               LocalAddr => 'localhost',
                               LocalPort => 9000,
                               Proto     => 'tcp');
```

Further reading

Further reading

- RFCs
  - RFC 3493: Basic Socket Interface Extensions for IPv6 (obsoletes RFC 2553)
    - see getaddrinfo for an example of client/server programming in an IPv4/IPv6 independent manner using some of the RFC 3493 extensions
  - RFC 3542: Advanced Sockets Application Program Interface (API) for IPv6 (obsoletes RFC 2292)
  - RFC 4038: Application Aspects of IPv6 Transition

Further Reading /2

- Links
  - Address-family independent socket programming for IPv6
    http://www.ipv6.or.kr/summit2003/presentation/II-2.pdf
  - Porting applications to IPv6 HowTo
    http://gsyc.escet.urjc.es/~eva/IPv6-web/ipv6.html
  - Porting Applications to IPv6: Simple and Easy - By Viagenie - http://www.viagenie.qc.ca/en/ipv6/presentations/IPv6%20porting%20appl_v1.pdf
  - Guidelines for IP version independence in GGF specifications
    http://www.ggf.org/documents/GWD-I-E/GFD-I.040.pdf
  - IPv6 Forum Programming and Porting links
    http://www.ipv6forum.org/modules.php?op=modload&name=Web_Links&file=index&req=viewlink&cid=56
  - FreeBSD Developers' Handbook Chapter on IPv6 Internals - http://www.freebsd.org/doc/en/books/developers-handbook/ipv6.html

Further Reading /3

- Links
  - Freshmeat IPv6 Development Projects - http://freshmeat.net/search/?q=IPv6
  - FutureSoft IPv6 - a portable implementation of the next generation Internet Protocol Version 6, complying with the relevant RFCs and Internet drafts - http://www.futsoft.com/ipv6.htm
  - IPv6 Linux Development Tools from Deepspace.net - http://www.deepspace6.net/sections/sources.html
  - Libpnet6 - an advanced networking library with full IPv6 support - http://pnet6.sourceforge.net/
  - USAGI Project - Linux IPv6 Development Project

    http://www.linux-ipv6.org/
- Books
  - IPv6 Network Programming by Jun-ichiro itojun Hagino
  - UNIX Network Programming (latest version) by W. Richard Stevens
  - IPv6 : Theory, Protocol, and Practice, 2nd Edition by Pete Loshin
  - IPv6 Network Administration, O'Reilly

Questions