



deploy

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 and the POSIX/BSD IPv6 socket API
- We do the same for Perl
- We consider common application porting issues
- We look at standards and recommendations

## Contributors

- Stig Venaas – UNINETT
- János Mohácsi – NIIF/Hungarnet



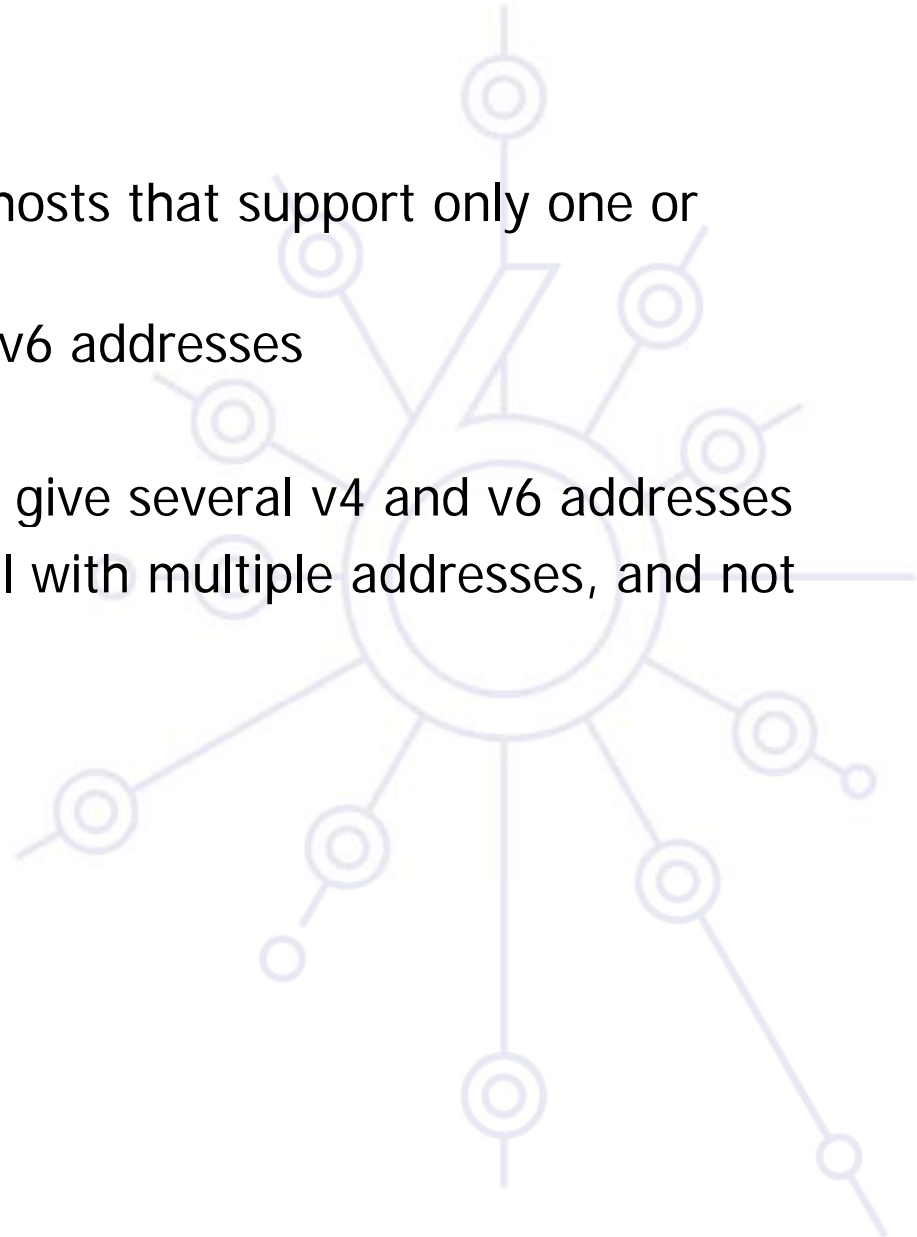


6 deploy

C IPv6 API

## Some basic principles

- The same binary should work on hosts that support only one or both IP protocols
- Interfaces may have both v4 and v6 addresses
  - usually multiple v6 addresses
- DNS lookups for a name may also give several v4 and v6 addresses
  - Applications often need to deal with multiple addresses, and not just use the first one





## 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

## 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
  - BSD has `sin6_len` as member too (also `sin_len` in `sockaddr_in`)
- ```
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`
  - I.e., we can use it in declarations and cast if necessary
  - For generic `sockaddr` pointer, use `struct *sockaddr`

## Socket functions

- Many functions are the same as before
  - `socket()`, `bind()`, `connect()`, `sendmsg()`, `sendto()`, `accept()`, `recvfrom()`, `recvmsg()`, `getpeername()`, `getsockname()`
  - All except `socket()` accept or return addresses but use pointer to struct `sockaddr` and address length, so no change needed
- `getaddrinfo()` – convert from name/text address to binary
- `getnameinfo()` – convert from binary to name or text address
- Functions for converting between binary and text address formats
  - For IPv4 there is `inet_addr()` and `inet_ntoa()`
  - With new API we have `inet_pton()` and `inet_ntop()` that works with both IPv4 and IPv6
  - But recommend using `getaddrinfo()/getnameinfo()`



## getaddrinfo()

- Arguments are IPv4/IPv6 address or FQDN, service name or port number and hints specifying what we want to do
- Gives us all we need for `socket()`, `bind()`, `connect()` etc
- `int getaddrinfo(const char *nodename, const char *servname, const struct addrinfo *hints, struct addrinfo **res);`
- `struct addrinfo`

```
int ai_flags; /* AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST, .. */
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 */
};
```
- `void freeaddrinfo(struct addrinfo *ai); /* for freeing results */`
- `const char *gai_strerror(int ecode); /* returns error string */`



## 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;
}
```



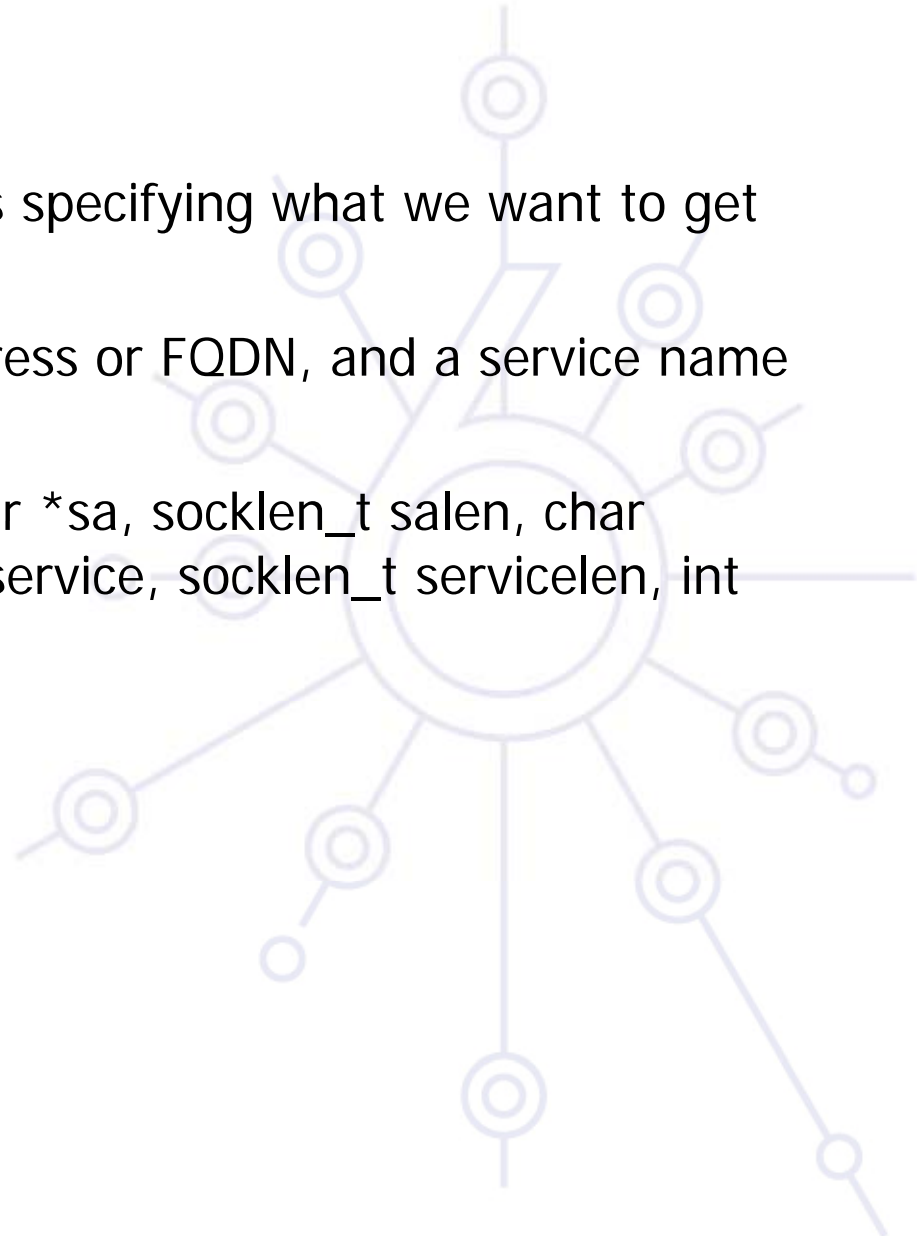
## Simple IPv4/IPv6 TCP client

```
/* 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();
```

## getnameinfo()

- Arguments are sockaddr and flags specifying what we want to get back
- Gives us a text string with IP address or FQDN, and a service name or port number

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





## Converting sockaddr to string

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

```
char hbuf[NI_MAXHOST], pbuf[NI_MAXSERV];
```

```
/* convert to names where possible, like www.kame.net/http */
```

```
if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), pbuf, sizeof(pbuf), 0) != 0)  
    errx(1, "an error occured");
```

```
/* or a numeric address/service port, like 127.0.0.1/80 */
```

```
if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), pbuf, sizeof(pbuf),  
    NI_NUMERICHOST | NI_NUMERICSERV) != 0)  
    errx(1, "an error occured");
```

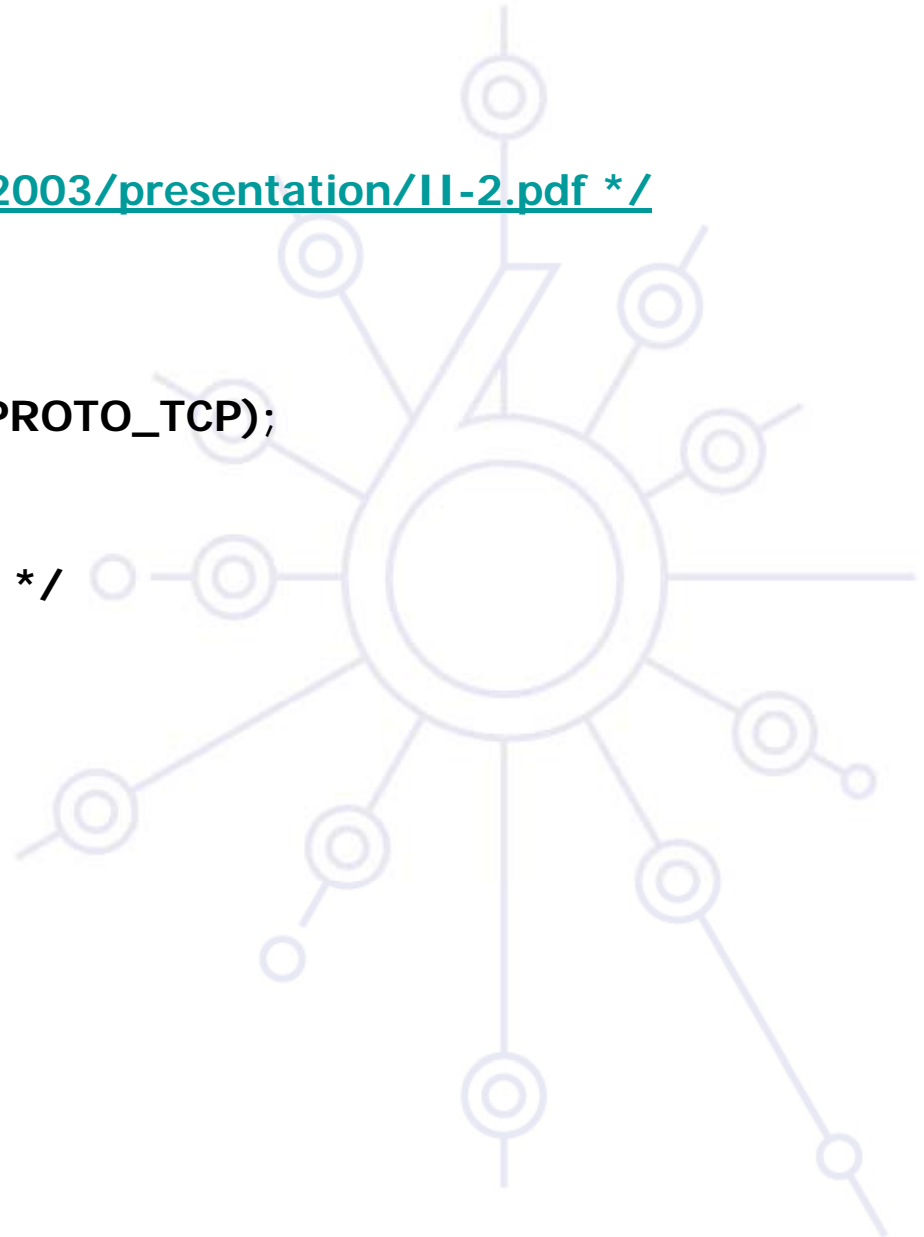




## Simple old IPv4 TCP server

`/* 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);
```

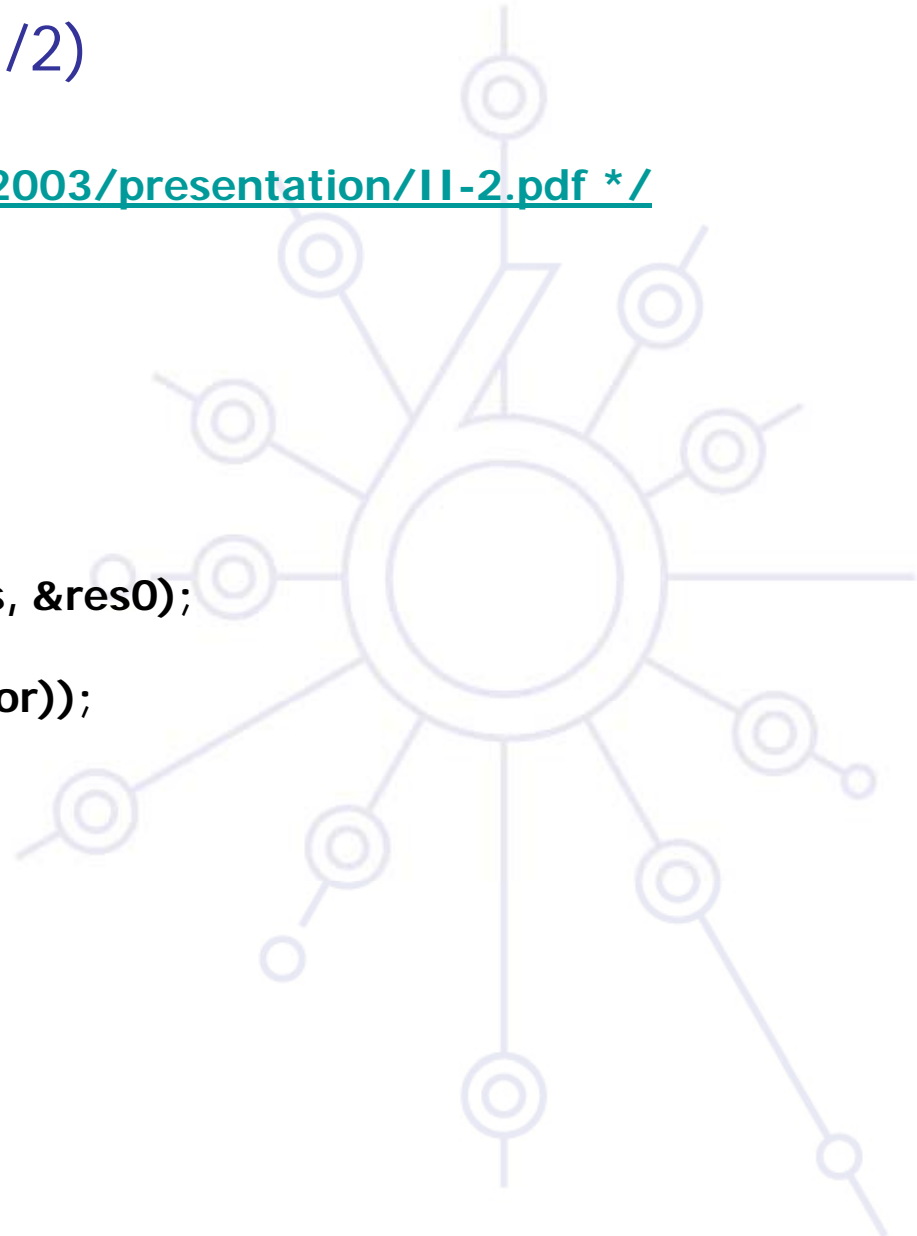




## Simple IPv4/IPv6 TCP server (1/2)

**/\* 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[] */
```

## Server issues

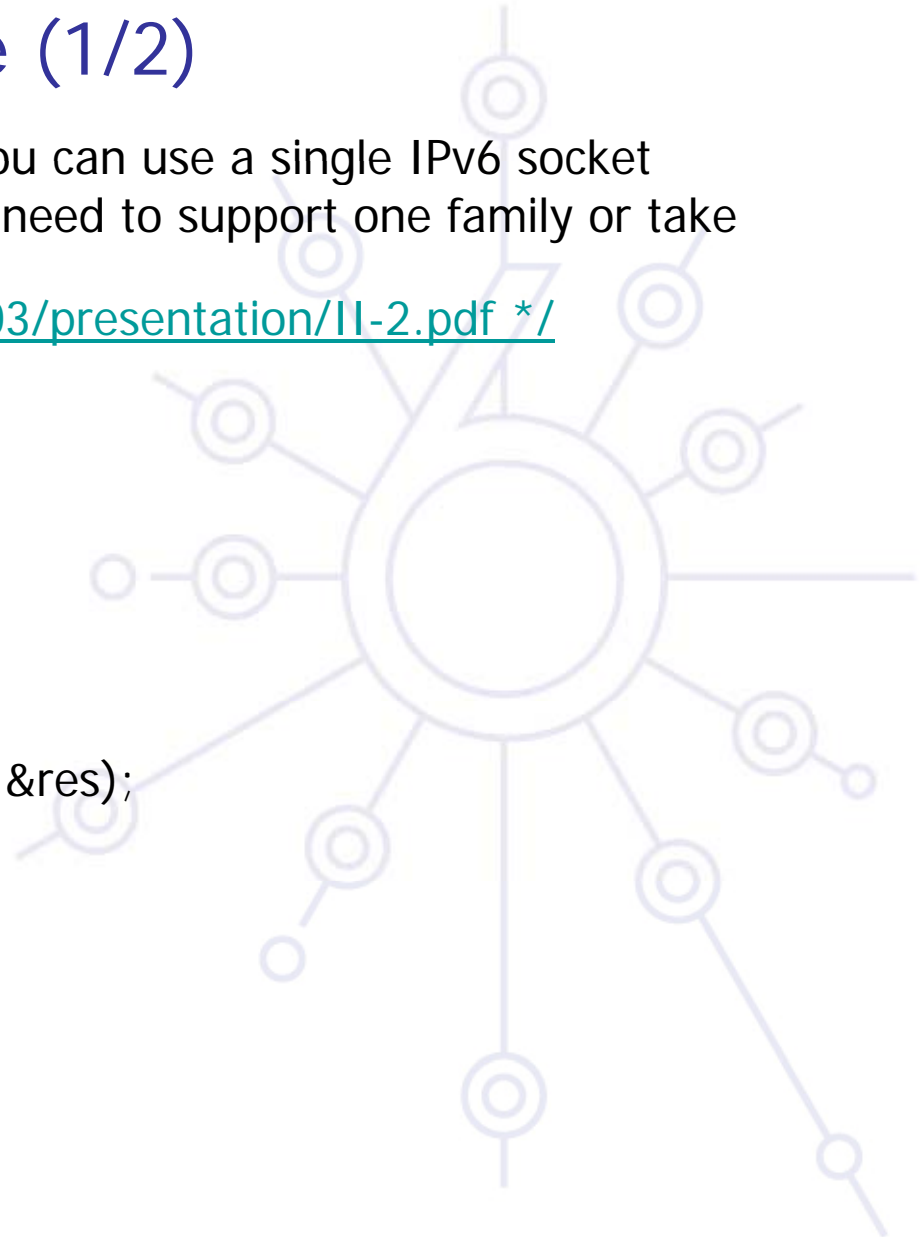
- Note that we typically end up with two sockets, one for v4 and one for v6
  - So need select or poll
- We also note the address family of each socket
  - Useful for some applications (e.g. {get,set}sockopt)
- Many operating systems support sending or receiving IPv4 on an IPv6 socket
  - One then only needs a single socket for receiving both
  - IPv4 address written as “::ffff:a.b.c.d”
- In the example we try to use IPV6\_V6ONLY to disable this
- One will typically bind to v6 socket first, then v4 socket. Linux by default uses mapped addresses, so v4 addresses are embedded within v6, and as a consequence doesn't allow a subsequent bind to addresses covered by an existing bind. But note that I haven't tested this lately.
- Since bind() behaviour is not well-defined, we only treat it as error if all binds fail

## 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 */
```



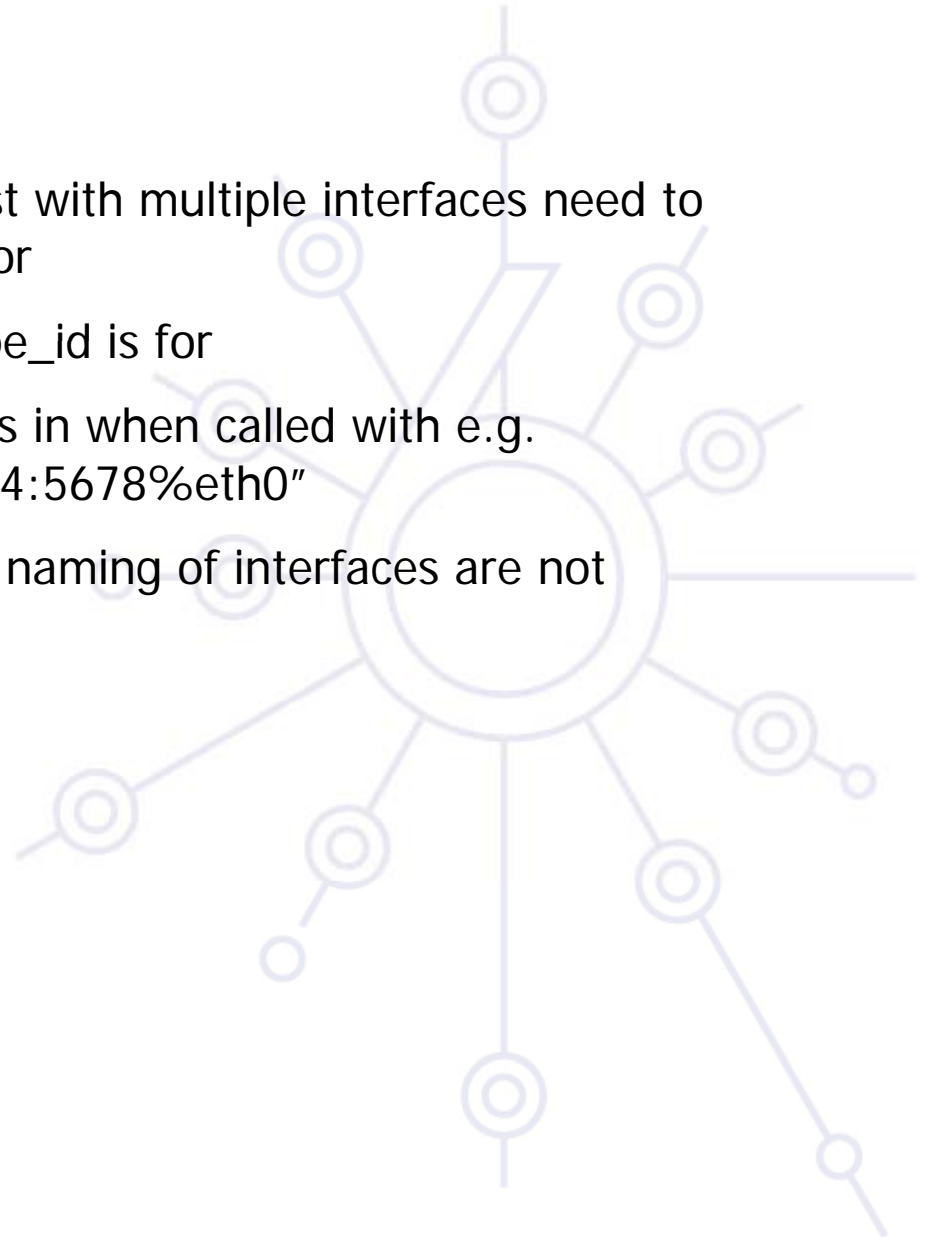


## One socket server example (2/2)

```
/* 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);
```

## 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



## More principles

- You want to have address family independent code where possible
  - Not separate code for v4 and v6
  - Try to use `sockaddr_in/sockaddr_in6` rather than `in_addr/in6_addr`
  - We then have address family together with the address
  - There is struct `sockaddr_storage` that is large enough for v6 (and `sockaddr_un`) that can be used for memory allocation and can be typecast to `sockaddr_in` etc if necessary
  - For pointers we can use struct `sockaddr *`



deploy

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_ADDRESS` - 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

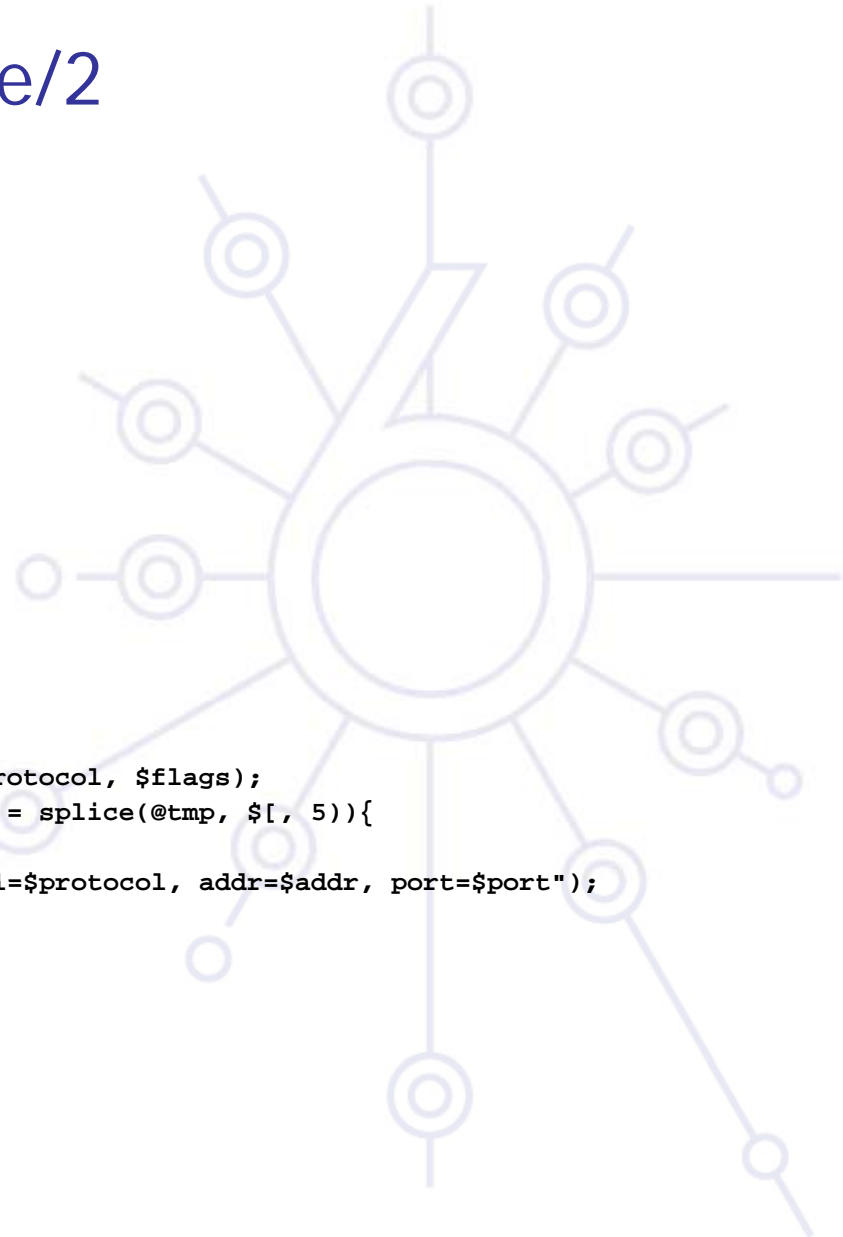
```
• use Getopt::Std;
• use Socket;
• use Socket6;
• use strict;

• 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');
```

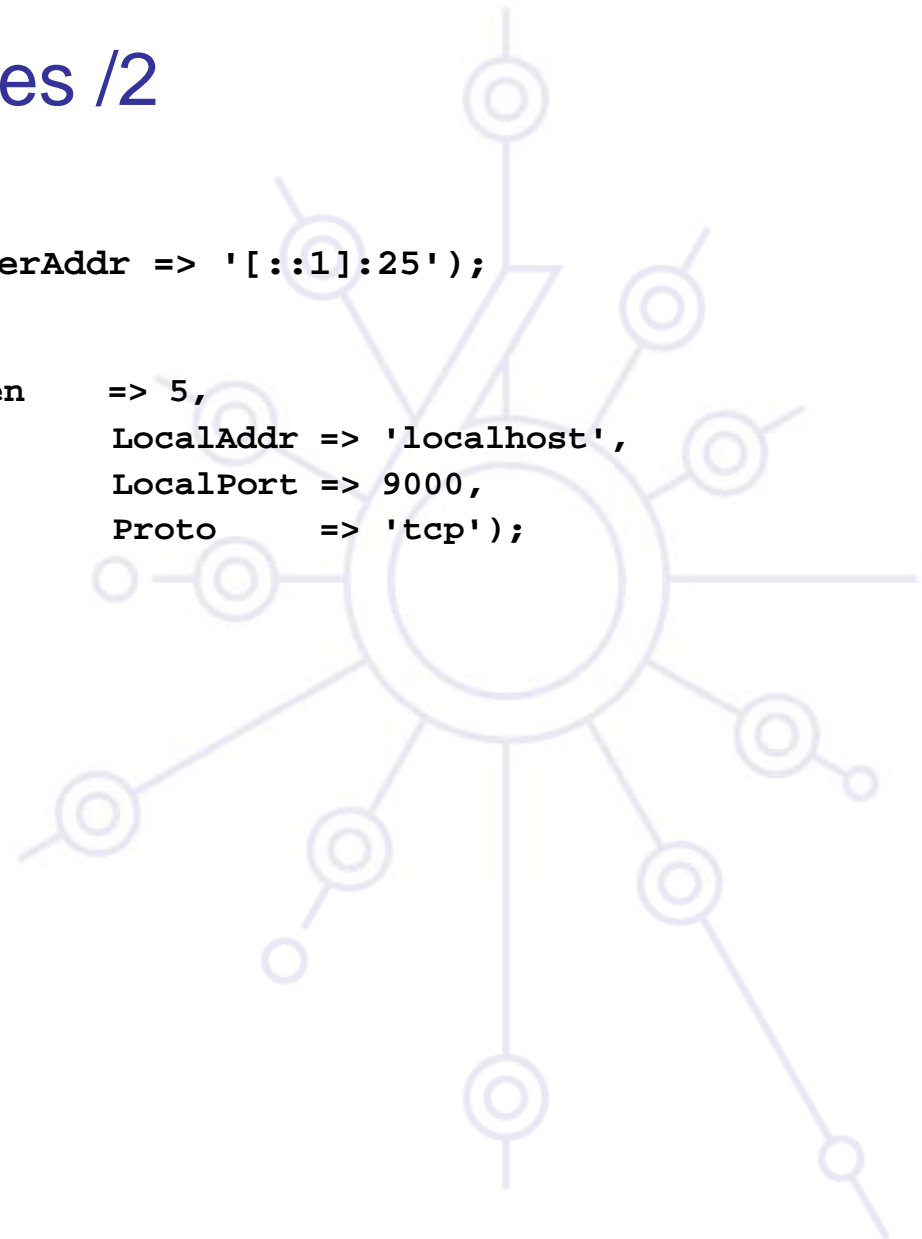
## IO::Socket::INET6 examples /2

- 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');`





6 deploy

Porting, issues and references

## 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. May end up with lots of ifdefs using old or new as appropriate. Might be good to put this code at a low level and create wrappers around it
- It's not uncommon that large applications have some duplication of network code. When porting it might be a good idea to fix this

## Miscellaneous issues 1

- For IPv6 UDP checksum is mandatory since there is no checksum in IP header
  - Problematic for applications that can cope with bit errors (e.g. video streaming?). Might be better to have a bit error than losing packet
  - UDP-Lite RFC 3828 is a solution
- connect() might try for like 30s if no response
  - When trying all addresses from getaddrinfo() we may not want to have 30s timeout for each

## Miscellaneous issues 2

- 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



## Miscellaneous issues 3

- 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



## 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://gsync.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%20a%20ppl\\_v1.pdf](http://www.viagenie.qc.ca/en/ipv6/presentations/IPv6%20porting%20a%20ppl_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](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