

IPv6 Network Programming



Global IPv6 Summit in KOREA 2004

2004. 7. 5

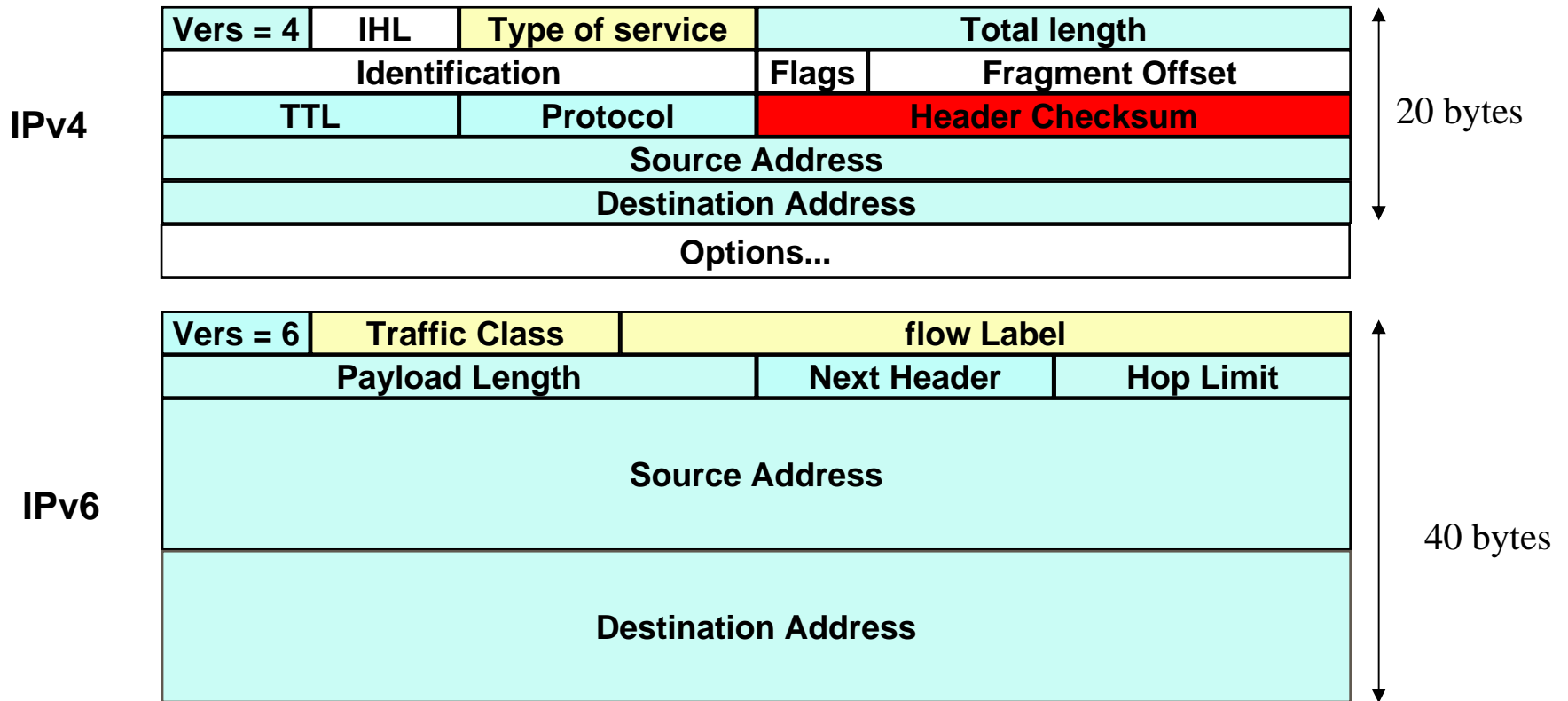
ETRI/PEC

TaeWan, You (twyou@etri.re.kr)

Contents

- ❑ IPv6 Protocol Overview
- ❑ Transition Mechanisms
- ❑ Porting to IPv6 Applications
 - Considerations for porting applications
- ❑ IPv6 socket APIs
 - RFC 3493: Basic socket interface extensions for IPv6
 - RFC 2292: Advanced sockets API for IPv6
- ❑ Developing protocol-independent applications
 - Draft-ietf-v6ops-application-transition-02.txt
- ❑ How to IPv6 programming with Winsock2.0

IPv6 - Basic Header



IPv6 Addressing

□ Basic Address Types

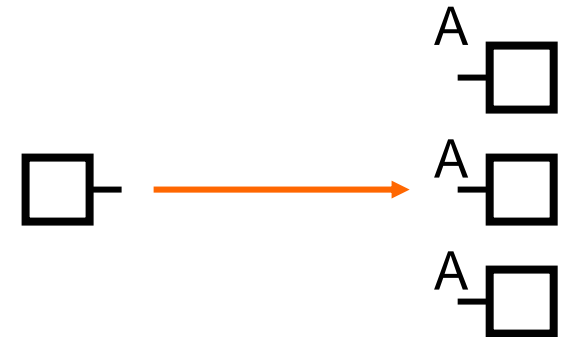
○ Unicast

- An identifier for a single interface
- Delivered to the single interface



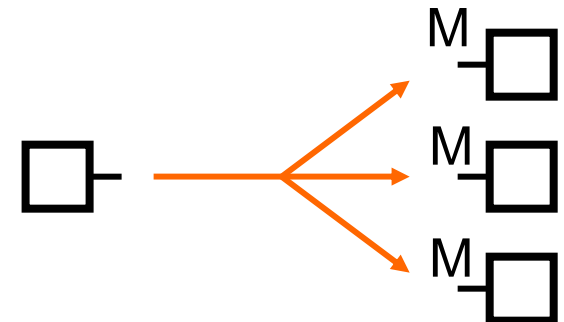
○ Anycast

- An identifier for a set of interfaces
- Delivered to one of the interfaces in a set



○ Multicast

- An identifier for a set of interfaces
- Delivered to all interfaces in a set



Representation of Address (1/2)

□ 128bits address scheme

- x:x:x:x:x:x:x:x
- 'x's are the hexadecimal values of the eight 16bits pieces of the address
- '::' indicates multiple groups of 16 bits of zeros.
 - 3ffe:2e01:0:0:0:31:0:21 -> 3ffe:2e01::31:0:21
- IPv6-address/prefix-length
 - 3ffe:0000:0000:cd30:0000:0000:0000:0000/64 -> 3ffe::cd30:0:0:0:0/64

□ An alternative form

- x:x:x:x:x:d.d.d.d
- 'd's are the decimal values of the four low-order 8-bit pieces of the address
 - 0:0:0:0:FFFF:129.144.52.38 -> ::FFFF:129.144.52.38

Representation of Address (2/2)

□ IPv6 literal addresses in URL's

- IPv6 literal address should be enclosed in "[" and "]" characters
- 2010:836B:4179::836B:4179
- `http://[2010:836B:4179::836B:4179]`
- `http://[2010:836B:4179::836B:4179]:80/index.html`

□ Specific case to give a zone identifier

- `Fe80::1%eth0`

□ Comparison to IPv4 for programming

- `129.254.112.56:5001`
- `[2001:230::1]:5001`

IPv6 - A lot of Address

	Prefix	Description
Unicast Address	2001::/16 3ffe::/16 2002::/16 ::FFFF:IPv4 address/96 Fe80::/10 ::1	Public Address Experimental Address 6to4 Address IPv4-mapped Address Link-local Address Loopback Address
Multicast Address	FF01::/16 FF02::/16 FF0e::/16	Node-Local scope Link-Local scope Global scope

Domain Name Service

❑ Extension DNS for IPv6

- IPv6 AAAA records supported starting in Bind 4.9.5 and 8.1.x
- Newer records such as A6, DNAME and Binary labels are supported starting Bind9

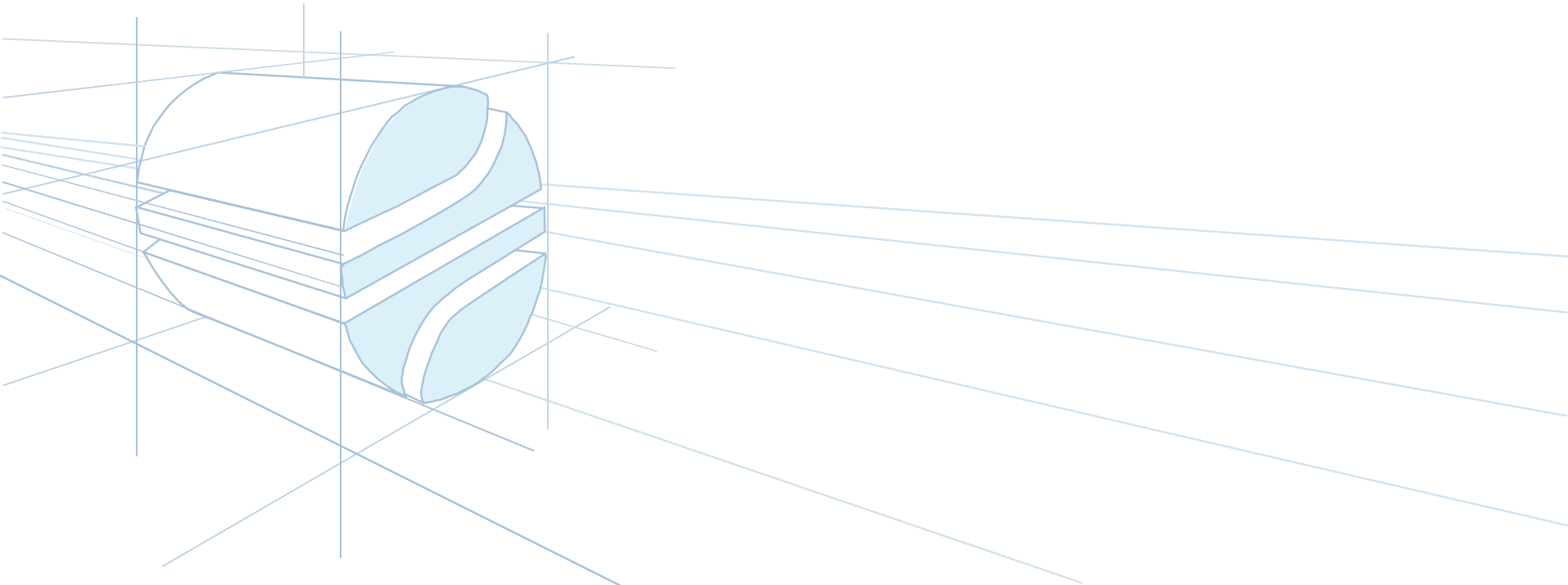
❑ IPv6 data queries over IPv4 and IPv6

- Current Bind distribution answers to IPv4 queries only
- Extensions to Bind 8.1.2 are available to allow IPv6 DNS queries

❑ Root servers

- Not configured for IPv6 native queries now
- But AAAA records can be used on the current root servers

Transition to IPv6



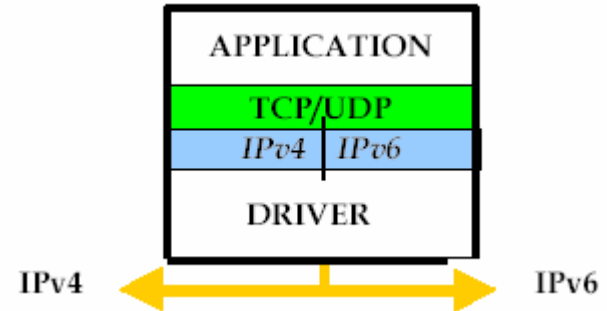
Transition Assumptions

- ❑ IPv6 is NOT backwards compatible with IPv4
- ❑ Facts of IPv6 Transition
 - Millions of nodes are running IPv4 today
 - Some nodes will never upgrade to IPv6
 - IPv4 and IPv6 will coexist for an extended period
 - No Flag Days
- ❑ Transition will be incremental
 - Possibly over several years
- ❑ No IPv4/IPv6 barriers at any time
- ❑ Must be easy for end user
 - Transition from IPv4 to dual stack must not break anything
- ❑ IPv6 is designed with transition in mind
 - Assumption of IPv4/IPv6 coexistence

Transition mechanisms

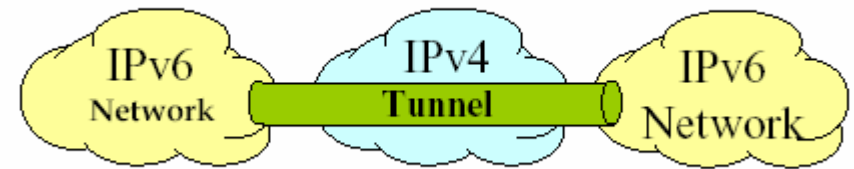
□ Host/Router

- Dual Stack (Dual IP layer)



□ Networks

- Tunneling



□ Gateway

- IPv4/6 Translator



Tunnel Types

❑ Configured tunnels

- Router to router

❑ Automatic tunnels

- Tunnel Brokers (RFC 3053)
 - Server-based automatic tunneling
- 6to4 (RFC 3056)
 - Router to router
- ISATAP (Intra-Site Automatic Tunnel Addressing Protocol)
 - Host to router, router to host
 - Maybe host to host
- 6over4 (RFC 2529)
 - Host to router, router to host
- Teredo
 - For tunneling through IPv4 NAT
- IPv64
 - For mixed IPv4/IPv6 environments
- DSTM (Dual Stack Transition Mechanism)
 - IPv4 in IPv6 tunnels

Translators

❑ Network level translators

- Stateless IP/ICMP Translation Algorithm (SIIT)(RFC 2765)
- NAT-PT (RFC 2766)
- Bump in the Stack (BIS) (RFC 2767)

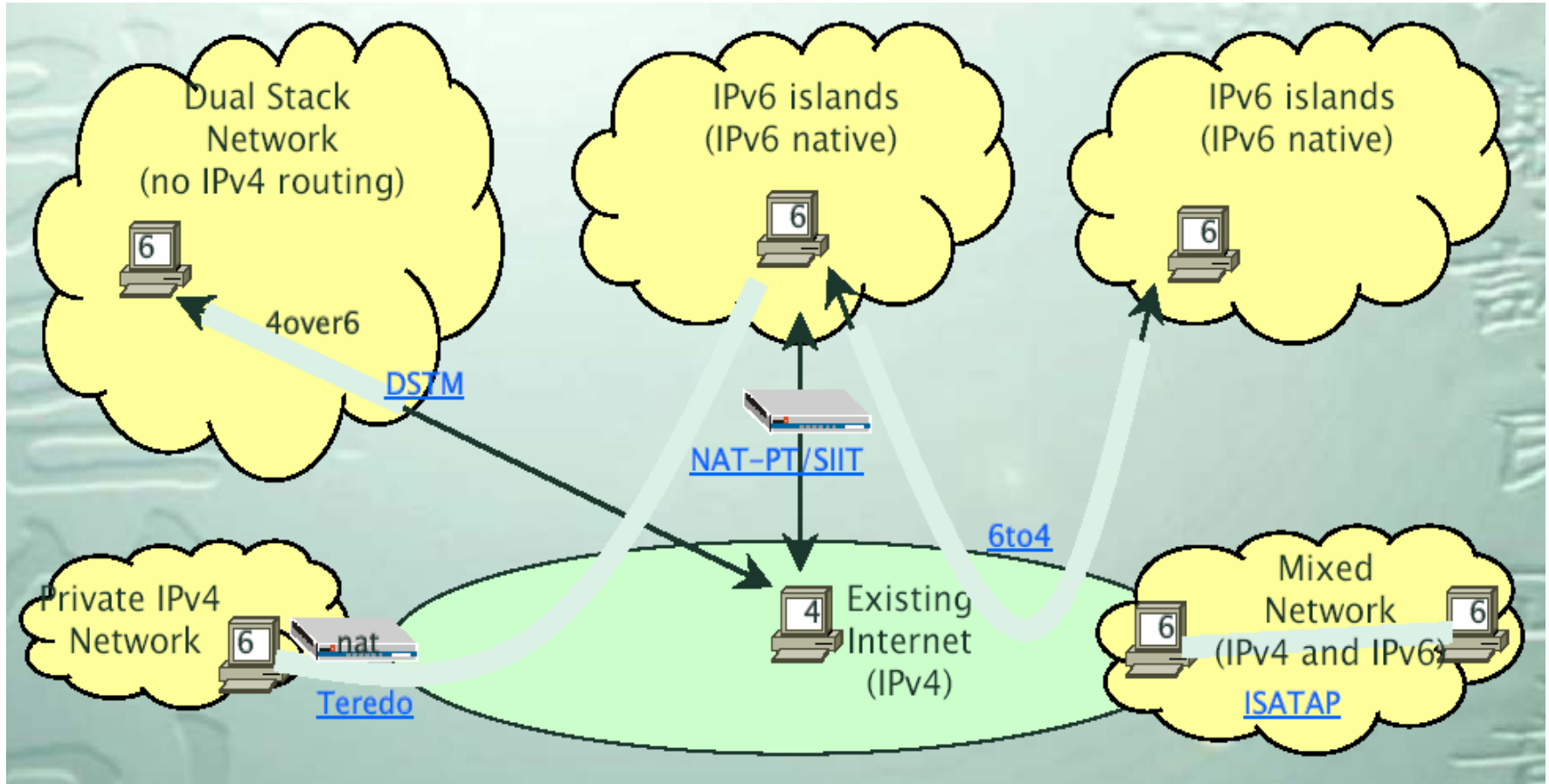
❑ Transport level translators

- Transport Relay Translator (TRT) (RFC 3142)

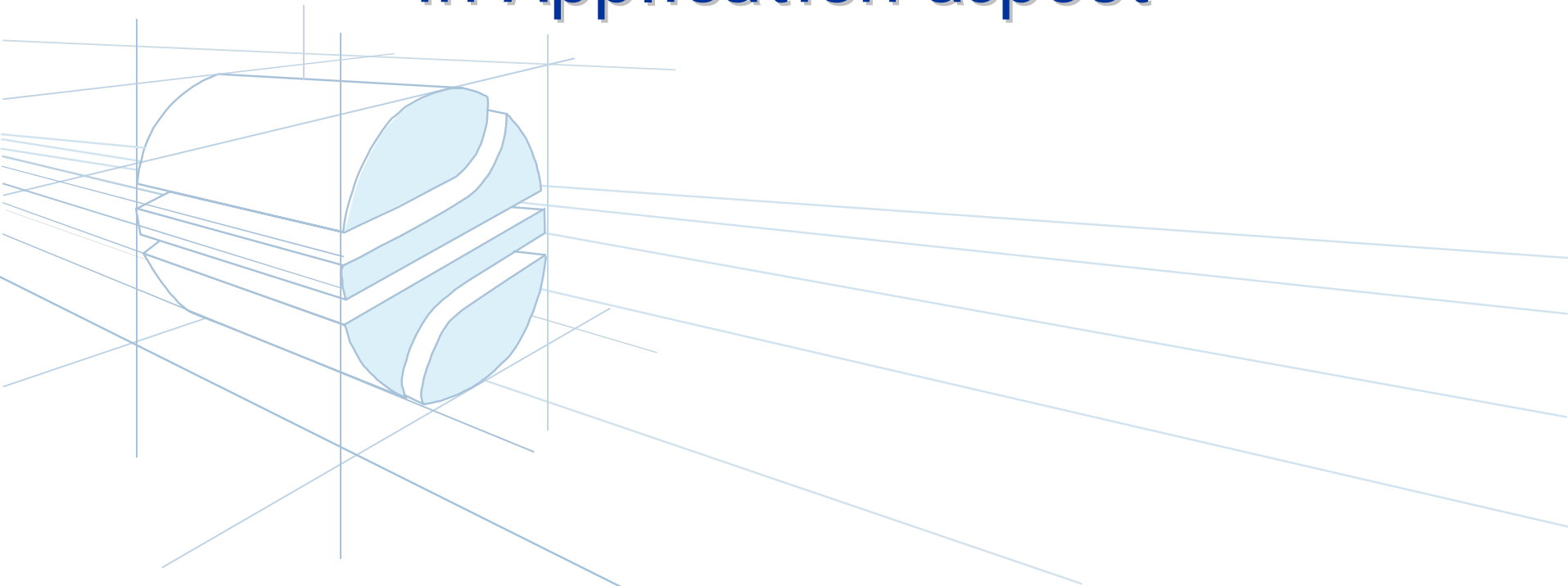
❑ Application level translators

- Bump in the API (BIA)(RFC 3338)
- SOCKS64 (RFC 3089)
- Application Level Gateways (ALG)

IPv6 Transition Mechanism Solution Map



Transition to IPv6 in Application aspect



Transition scenarios (1/3)

❑ Important requirement in IPv6 transition

- Existing services should work in the new environment
- Continue to work with IPv4 nodes

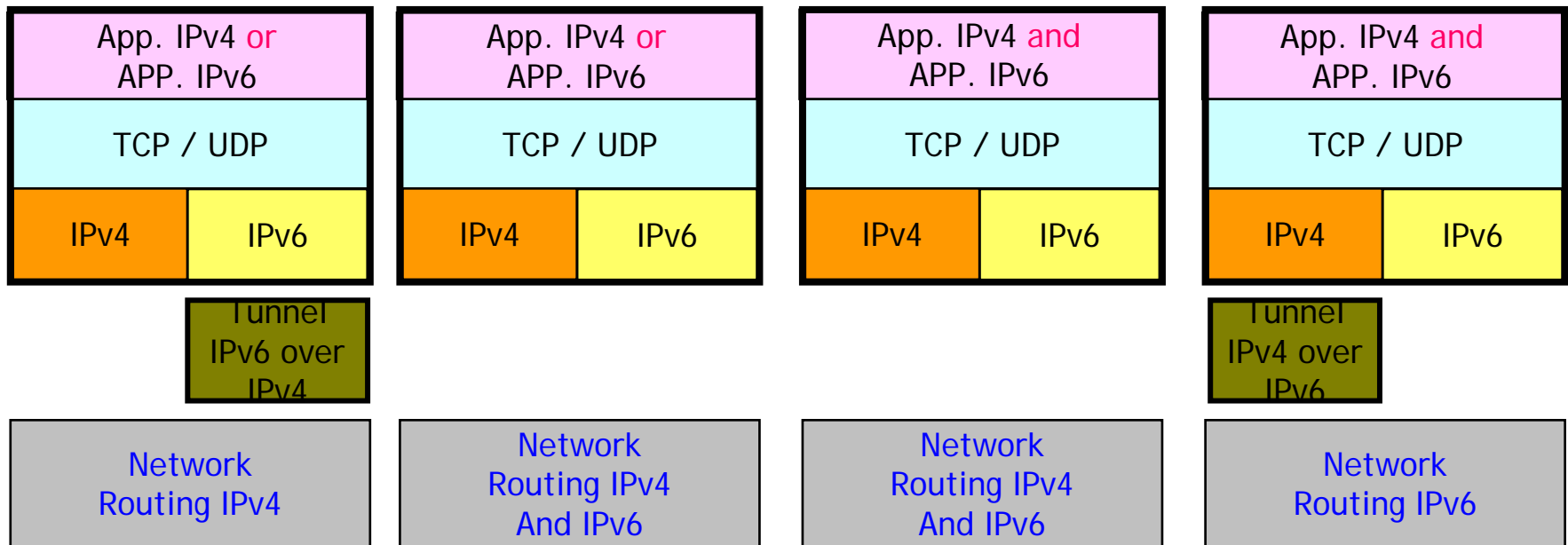
❑ For already working network the better solution

- Maintain the IPv4 stack and introduce IPv6 stack
 - Use the dual stack environment
- Solve the IPv4 and IPv6 interlocking problems
 - Needed address translator
 - Including headers and sometimes protocol payload.

Transition scenarios (2/3)

□ Transition steps

- Step 1: the network remains unchanged
- Step 2: IPv6 is available at network level unless applications remain unchanged
- Step 3: Application are ready to simultaneously on IPv4 and IPv6
- Step 4: IPv4 network has been removed



Transition scenarios (3/3)

❑ Application porting process

- Takes place between step 2 ~ 3.

❑ How to make such application adaptations

○ Maintaining two application versions

- Consists in developing a complete set of applications designed to work only over IPv6 transport layer
- During transition period
 - ✓ Necessary to provide data servers working both IPv4 and IPv6

○ Looks for changing existing applications

- Poring process
 - ✓ Review source code and produces a new version adapted to work over both IPv4 and IPv6 environments
- The porting process is a little bit more complex than IPv6 only support
 - ✓ Result is more flexible to be used during transition period.

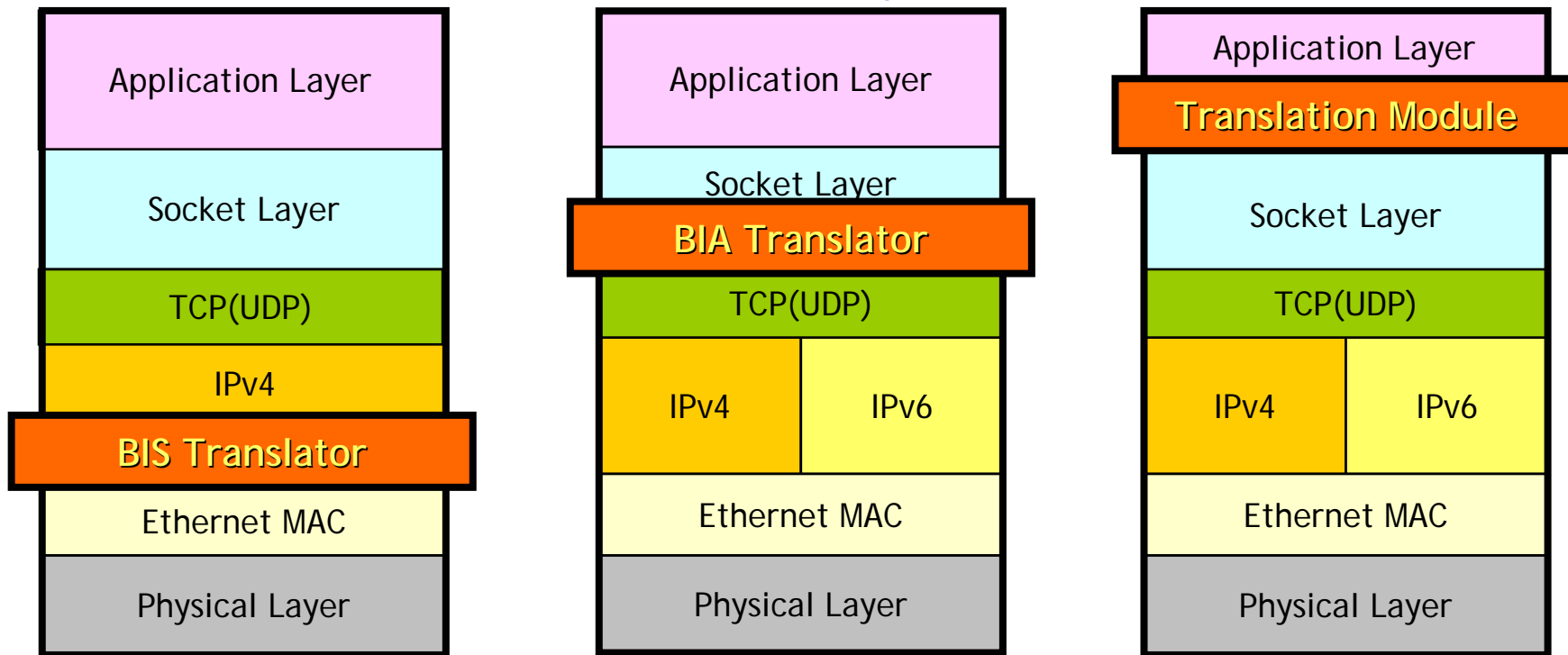
Transition to IPv6

- ❑ Differences between IPv4 socket interface and IPv6 socket interface
 - Name (DNS) and IP addresses management
 - Communication constants and data structures
 - Functions names and parameters
- ❑ Dual stack is required (IPv4, IPv6)
 - Continue using IPv4 applications
 - Used IPv6 network
- ❑ IPv6 scenery
 - Removing IPv4 network and operating only through IPv6 node
 - Need to translator from the IPv6 network interface into the IPv4 programming interface

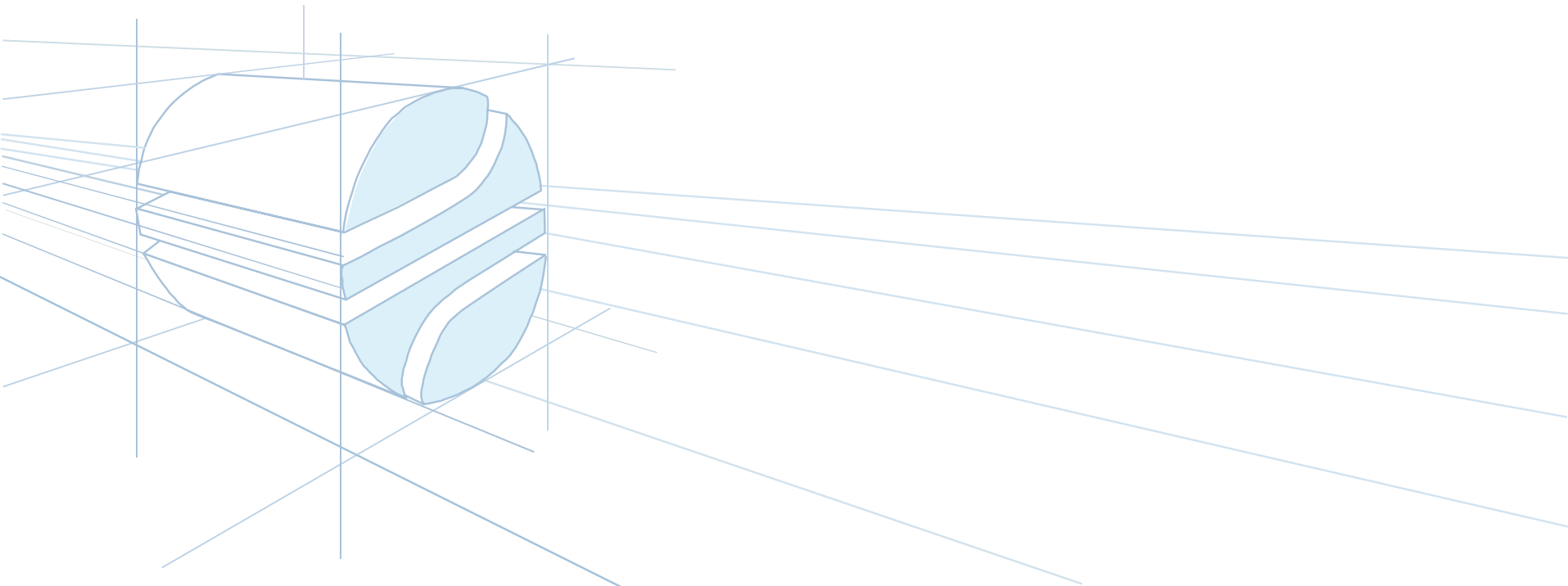
Transition to IPv6 without changing applications

□ Application interface translators

- BIS (Bump-in-the-Stack)
- BIA (Bump-in-the-API)
- ALG (Application-Level-Gateway)



Porting to IPv6 Applications



Porting applications

❑ Porting to IPv6 applications

- Converted without too much effort
- Exception cases
 - Special use of IPv4 or include advanced features
 - ✓ Multicast, raw sockets or other IP options

❑ Requirement for porting an existing application

- Network information storage: data structures
- Resolution and conversion address functions
- Communication API functions and pre-defined constants

❑ Other considerations

- Renumbering
- Multi-homing

Considerations

- ❑ Presentation format for IP address
 - 129.254.12.211:5001/[2001:230:1::1]:5001
- ❑ Transport layer API
 - Network information storage
 - Address conversions functions
 - Communication API functions
 - Network configuration options
- ❑ Name and address resolution
- ❑ Specific IP dependencies
 - IP address selection
 - Address framing

Porting source code

❑ Changes socket API for IPv6 support

- New socket address structure to carry IPv6 addresses
- New address conversion functions and several new socket options
- Explained in RFC-2553 which obsolete by RFC-3493
 - Basic BSD socket API
 - Basic IPv6 features required by TCP and UDP applications
 - Including multicasting, and providing complete compatibility for existing IPv4 applications
- Access to more advanced features is addressed in RFC-2292
 - Raw sockets, header configuration, etc.

Socket API for IPv6 applications



*RFC 3493: Basic Socket Interface Extensions
for IPv6*

RFC 2292: Advanced Sockets API for IPv6

RFC 3493

❑ 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 defined in another document.

- RFC 2292: Advanced Socket API for IPv6

What needs to be changed (1/2)

□ Core socket functions

- Setting up and tearing down TCP connections, and sending and receiving UDP packet
- Designed to be transport independent
- Need not change for IPv6
 - Protocol address are carried via opaque pointers

□ Address data structures

- Protocol-specific data structure for IPv4
 - Structure `sockaddr_in`
 - The structure is not large enough to hold the 16-octet IPv6 address as well as the other information
 - New address data structure must be defined for IPv6

What needs to be changed (2/2)

❑ Name-to-address translation functions

- `gethostbyname()`, `gethostbyaddr()`
- New functions are defined which support both IPv4 and IPv6

❑ Address conversion functions

- `Inet_ntoa()`, `inet_addr()`
 - Convert IPv4 addresses between binary and printable form
- Designed two analogous functions that convert both IPv4 and IPv6 addresses

❑ Miscellaneous features

- IPv6 hop limit header field
- Control the sending and receiving of IPv6 multicast packets

The socket interface chances for IPv6

- ❑ Address family and Protocol family
 - AF_INET, PF_INET
- ❑ Address Structure
 - Structure in_addr
- ❑ Socket Address Structure
 - Structure sockaddr_in
- ❑ The Socket functions
 - Socket(), bind(), connect(), accept(), and send() etc.
- ❑ Compatibility
- ❑ Wildcard address
- ❑ Loopback address

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 first argument to the socket()

□ New protocol family name

- PF_INET6 is defined in <sys/socket.h>
 - #defined PF_INET6 AF_INET6

IPv4	IPv6
AF_INET	AF_INET6
PF_INET	PF_INET6

IPv6 Address Structure

□ New address structure

- in6_addr structure
- Holds a single IPv6 address (126 bits) is defined <netinet/in.h>

```
struct in6_addr {
    uint8_t s6_addr[16]; /* IPv6 address */
};
```

- The IPv6 address is stored in network byte order

IPv4	IPv6
<pre>Struct in_addr { unsigned int s_addr }</pre>	<pre>Struct in6_addr { uint8_t s6_addr[16] }</pre>

Socket Address structure

- ❑ Each protocol-specific data structure
 - Design to be cast into a protocol-independent data structure (“sockaddr” structure)
- ❑ New socket address structure version for BSD
 - 4.3 BSD release version
 - 4.4 BSD release version

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

Socket Address structure (Cont'd)

□ 4.3 BSD-based systems

```
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 */
};
```

□ 4.4 BSD-based systems

```
struct sockaddr_in6 {
    uint8_t sin6_len; /* length of this struct */
    sa_family_t sin6_family; /* AF_INET6 */
    ...
}
```

The socket functions (1/2)

□ Create a socket descriptor

```
s = socket(AF_INET6, SOCK_STREAM, 0);
```

```
s = socket(AF_INET6, SOCK_DGRAM, 0);
```

○ Create AF_INET6 socket

- Use the sockaddr_in6 address structure

□ Other functions

○ No changes to the syntax of the other functions

- Use an opaque address pointers
- Carry an address length as a function argument

○ Bind(), connect(), sendmsg(), sendto(), accept(), recvfrom(), and recvmsg() etc.

The socket functions (2/2)

□ Other functions (Cont'd)

○ Value-Result Argument

- When socket address structure is passed by reference in socket function
- Different socket address structure between pass to kernel and pass to process

Socket address structure pass.

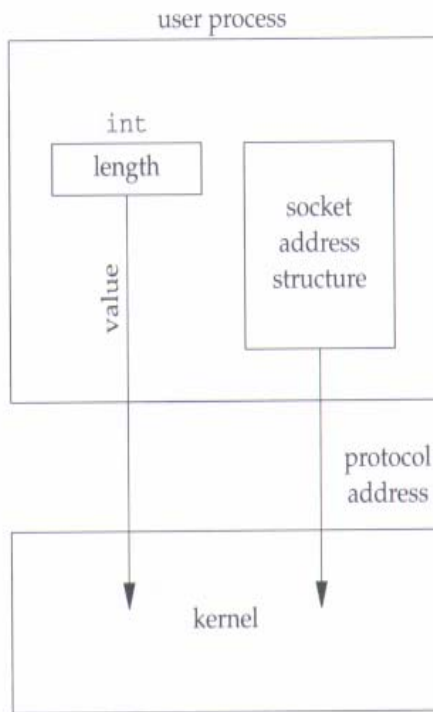


Figure 3.6 Socket address structure passed from process to kernel.

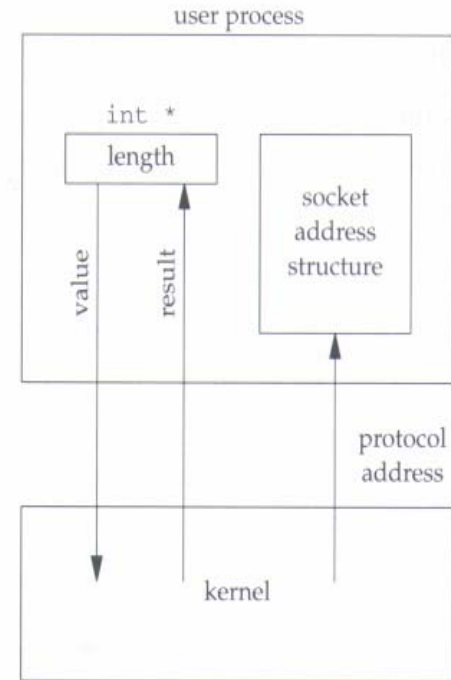


Figure 3.7 Socket address structure passed from kernel to process.

Bind, connect, sendto

Accept, recvfrom, getsockname,
getpeername

Compatibility

❑ Compatibility with IPv4 Applications

- Combination of IPv4/TCP, IPv4/UDP, IPv6/TCP and IPv6/UDP sockets simultaneously within the same process
- This applications should interoperate with IPv4 node.

❑ Compatibility with IPv4 nodes

- The API provides a different type of compatibility
- Use IPv4-mapped IPv6 address format

```
::FFFF:<IPv4-address>
```

- Allows the IPv4 address of an IPv4 node to be represented as an IPv6 address
- IN6_IS_ADDR_V4MAPPED() macro is provided

IPv6 Wildcard address (1/2)

- ❑ Wildcard address
 - Symbolic constant INADDR_ANY
- ❑ IPv6 wildcard address in two forms
 - Global variable named "in6addr_any"

```
extern const struct in6_addr in6addr_any;
```

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

IPv6 Wildcard address (2/2)

□ IPv6 wildcard address in two forms (Cont'd)

- Symbolic constant named IN6ADDR_ANY_INIT

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

- This constant can be use ONLY at declaration time

```
/* This is the WRONG way to assign an unspecified address */
```

```
struct sockaddr_in6 sin6;
```

```
...
```

```
sin6.sin6_addr = IN6ADDR_ANY_INIT; /* will NOT compile */
```

- IPv4 "INADDR_xxx" constants are all defined in host byte order
- IPv6 IN6ADDR_xxx constants and the IPv6 in6addr_xxx externals are defined in network byte order

IPv6 Loopback Address (1/2)

❑ IPv4 Loopback address

- INADDR_LOOPBACK

❑ Two forms of IPv6 loopback address

- Global variable

```
extern const struct in6_addr in6addr_loopback;
```

```
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)
```

```
...
```

- Use in6addr_loopback similarly IPv4

IPv6 Loopback Address (2/2)

□ Two forms of IPv6 loopback address (Cont'd)

- Symbolic constant is named "IN6ADDR_LOOPBACK_INIT"

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

- Used at declaration time ONLY
- This constant cannot be used in an assignment

Comparison to IPv4 and IPv6

□ Sample code for IPv4

```
listenfd = Socket(AF_INET, SOCK_STREAM, 0);
```

```
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);
```

```
Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
```

IPv4	IPv6
INADDR_ANY	IN6ADDR_ANY_INIT / in6addr_any
INADDR_LOOPBACK	IN6ADDR_LOOPBACK_INIT / In6_addr_loopback
INADDR_BROADCAST	Not exist

Portability Additions (1/3)

□ Simple addition to the socket API

- struct sockaddr_storage
- Simplify writing code that is portable across multiple address families and platforms
- Designed goal
 - Large enough to accommodate all supported protocol-specific address structures
 - Aligned at an appropriate boundary so that pointers to it can be cast as pointers to protocol specific address structures and used to access the fields of those structures without alignment problems

Portability Additions (2/3)

□ Example implementation design

- Align on a 64-bit boundary

```
#define _SS_MAXSIZE 128 /* Implementation specific max size */
```

```
#define _SS_ALIGNSIZE (sizeof (int64_t))
```

```
#define _SS_PAD1SIZE (_SS_ALIGNSIZE - sizeof (sa_family_t))
```

```
#define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof (sa_family_t) + _SS_PAD1SIZE +  
_SS_ALIGNSIZE))
```

```
struct sockaddr_storage {  
    sa_family_t ss_family; /* address family */  
    /* Following fields are implementation specific */  
    char __ss_pad1[_SS_PAD1SIZE];  
    int64_t __ss_align; /* field to force desired structure */  
    char __ss_pad2[_SS_PAD2SIZE];  
};
```

Portability Additions (3/3)

□ Implementation include "sa_len" field

```
/* Definitions used for sockaddr_storage structure paddings design. */
#define _SS_PAD1SIZE (_SS_ALIGNSIZE - (sizeof (uint8_t) + sizeof (sa_family_t))
#define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof (uint8_t) + sizeof (sa_family_t) + _SS_PAD1SIZE +
    _SS_ALIGNSIZE))
struct sockaddr_storage {
    uint8_t ss_len; /* address length */
    sa_family_t ss_family; /* address family */
    /* Following fields are implementation specific */
    char __ss_pad1[_SS_PAD1SIZE];
        /* 6 byte pad, this is to make implementation specific pad up to alignment
        field that */
        /* follows explicit in the data structure */
    int64_t __ss_align; /* field to force desired structure */
        /* storage alignment */
    char __ss_pad2[_SS_PAD2SIZE];
        /* 112 byte pad to achieve desired size, */
        /* _SS_MAXSIZE value minus size of ss_len, */
        /* __ss_family, __ss_pad1, __ss_align fields is 112 */
};
```

Example for IPv6

□ Protocol Independent code

```
int Family = DEFAULT_FAMILY;
int SocketType = DEFAULT_SOCKETTYPE;
char *Port = DEFAULT_PORT;
char *Address = NULL;
int i, NumSocks, RetVal, FromLen, AmountRead;
SOCKADDR_STORAGE From;
...
...
// Since this socket was returned by the select(), we know we
// have a connection waiting and that this accept() won't block.
ConnSock = accept(ServSock[i], (LPSOCKADDR)&From, &FromLen);
...
...
RetVal = sendto(ServSock[i], Buffer, AmountRead, 0,
                (LPSOCKADDR)&From, FromLen);
```

Interface Identification (1/2)

- Interfaces are normally known by names
 - Le0, sl1, ppp2 and the like
 - On Berkeley-derived implementations
 - The kernel assigns a unique positive integer value to that interface
 - The API defined two functions that map between an interface name and index
 - Name-to-Index
 - Index-to-Name
 - Returns all the interface names and indexes
 - Return the dynamic memory allocated

Interface Identification (2/2)

❑ Name-to-Index

```
unsigned int if_nametoindex(const char *ifname);
```

❑ Index-to-Name

```
char *if_indextoname(unsigned int ifindex, char *ifname);
```

❑ Return all interface names and indexes

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

❑ Free memory

```
void if_freenameindex(struct if_nameindex *ptr);
```

Socket Options (1/5)

- All of these new options are at the IPPROTO_IPv6 level
 - Use a parameter in “getsockopt()” and “setsockopt()” calls
 - “IPv6_” is used in all of the new socket options
 - Defined in <netinet/in.h>
 - Options
 - Unicast Hop Limit
 - Sending and receiving multicast packets
 - IPv6_V6ONLY option for AF_INET6 sockets

Socket Options (2/5)

□ Unicast hop limit

○ IPv6_UNICAST_HOPS

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

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

Socket Options (3/5)

□ Sending and receiving multicast packets

- Three socket options for sending multicast packets
 - IPv6_MULTICAST_IF
 - ✓ Argument type: unsigned int
 - IPV6_MULTICAST_HOPS
 - ✓ Argument type: int
 - IPV6_MULTICAST_LOOP
 - ✓ Argument type: unsigned int
- Two socket options for reception of multicast packets
 - IPV6_JOIN_GROUP
 - IPV6_LEAVE_GROUP
 - ✓ All of these option's argument type is struct ipv6_mreq

```
struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr; /* IPv6 multicast addr */
    unsigned int ipv6mr_interface; /* interface index */
};
```

Socket Options (4/5)

- **IPV6_V6ONLY** option for **AF_INET6** sockets
 - Restricts **AF_INET6** sockets to IPv6 communications only
 - Send and receive IPv6 packets only (this option is turned on)
 - Default option is turned off

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

Socket Options (5/5)

- Summarizes the changes

IPv4	IPv6
IP_TTL	IPV6_UNICAST_HOPS
IP_MULTICAST_IF	IPV6_MULTICAST_IF
IP_MULTICAST_TTL	IPV6_MULTICAST_HOPS
IP_MULTICAST_LOOP	IPV6_MULTICAST_LOOP
IP_ADD_MEMBERSHIP	IPV6_JOIN_GROUP
IP_DROP_MEMBERSHIP	IPV6_LEAVE_GROUP
Struct ip_mreq	struct ipv6_mreq

Library functions (1/6)

□ New functions are needed

- Lookup IPv6 address in the DNS
 - Both forward lookup (nodename-to-address translation) and reverse lookup (address-to-nodename translation)
- Convert IPv6 addresses between their binary and textual form
- Commonly used functions for IPv4
 - `gethostbyname()`, `gethostbyaddr()`
- New functions are defined to handle both IPv4 and IPv6 addresses

Library functions (2/6)

❑ Protocol-independent node name and service name translation

```
int getaddrinfo(const char *nodename, const char *servname, const
                struct addrinfo *hints, struct addrinfo **res);
void freeaddrinfo(struct addrinfo *ai);
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 */
};
```

Library functions (3/6)

IPv4	IPv6
gethostbyaddr (xxx, 4, AF_INET)	<pre> struct addrinfo myaddr, *result; memset (&myaddr, 0, sizeof(myaddr)); myaddr.ai_family = PF_INET6; myaddr.ai_protocol = IPPROTO_IPV6; getaddrinfo (xxx, NULL, AF_INET6, &myaddr, &result) freeaddrinfo(result); </pre>
gethostbyname (name)	<pre> struct addrinfo myaddr, *result; memset (&myaddr, 0, sizeof(myaddr)); myaddr.ai_family = PF_INET6; myaddr.ai_flags = AI_PASSIVE; getaddrinfo (name, NULL, AF_INET6, &myaddr, &result) freeaddrinfo(result); </pre>

Library functions (4/6)

□ Socket address structure to node name and service name

○ Getnameinfo()

- Used to translate the contents of a socket address structure to a node name and/or service name

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

```
#include <netdb.h>
```

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

Library functions (5/6)

□ Address conversion functions

○ Two IPv4 functions

- `inet_addr()`, `inet_ntoa()` convert an IPv4 address between binary and text form

○ New functions convert both IPv6 and IPv4 addresses

```
#include <arpa/inet.h>
```

```
int inet_pton(int af, const char *src, void *dst);
```

```
const char *inet_ntop(int af, const void *src, char *dst,  
                      socklen_t size);
```

Library functions (6/6)

❑ Address testing macros

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

```
int IN6_IS_ADDR_UNSPECIFIED (const struct in6_addr *);  
int IN6_IS_ADDR_LOOPBACK (const struct in6_addr *);  
int IN6_IS_ADDR_MULTICAST (const struct in6_addr *);  
int IN6_IS_ADDR_LINKLOCAL (const struct in6_addr *);  
int IN6_IS_ADDR_SITELOCAL (const struct in6_addr *);  
int IN6_IS_ADDR_V4MAPPED (const struct in6_addr *);  
int IN6_IS_ADDR_V4COMPAT (const struct in6_addr *);  
int IN6_IS_ADDR_MC_NODELOCAL (const struct in6_addr *);  
int IN6_IS_ADDR_MC_LINKLOCAL (const struct in6_addr *);  
int IN6_IS_ADDR_MC_SITELOCAL (const struct in6_addr *);  
int IN6_IS_ADDR_MC_ORGLOCAL (const struct in6_addr *);  
int IN6_IS_ADDR_MC_GLOBAL (const struct in6_addr *);
```

Developing protocol-independent applications



RFC 3493: Basic Socket Interface Extensions for IPv6

RFC 2292: Advanced Sockets API for IPv6

RFC 2292

❑ Advanced Sockets API for IPv6

- Progress for changes to RFC 2133

❑ API Features

- Raw sockets under IPv6

- Interface identification

- Specifying the outgoing interface and determining the incoming interface

- IPv6 extension headers

- Hop-by-Hop options, Destination options, and the Routing header (source routing)

API Changed (Summary)

	IPv4	IPv6	
Data structures	AF_INET	AF_INET6	
	in_addr sockaddr_in	in6_addr sockaddr_in6	
Name-to-address functions	inet_aton() inet_addr()	inet_pton() *	IPv4 and IPv6 functions
	inet_ntoa()	inet_ntop() *	
Address conversion functions	gethostbyname() gethostbyaddr()	getipnodebyname() getipnodebyaddr getnameinfo() * getaddrinfo() *	

Developing Protocol-independent Applications

Draft-ietf-v6ops-application-transition-02.txt



Developing new Applications (1/9)

❑ IP version-independent structures

- Avoid structs `in_addr`, `in6_addr`, `sockaddr_in`, and `sockaddr_in6`
- Example)

```
struct in_addr in4addr;  
struct in6_addr in6addr;  
foo (&in4addr, AF_INET); // IPv4 case  
foo (&in6addr, AF_INET6); // IPv6 case
```

- Use `sockaddr_storage`

```
struct sockaddr_storage ss;  
int sslen;  
/* AF independent! - use sockaddr when passing a pointer */  
/* note: it's typically necessary to also pass the length explicitly */  
foo((struct sockaddr *)&ss, sslen);
```

Developing new Applications (2/9)

□ IP version-independent APIs

- New address independent variants that hide the gory details of name-to-address translation vice versa.

- gethostbyname()
- gethostbyaddr()

- Getaddrinfo() can return multiple addresses

- Localhost.

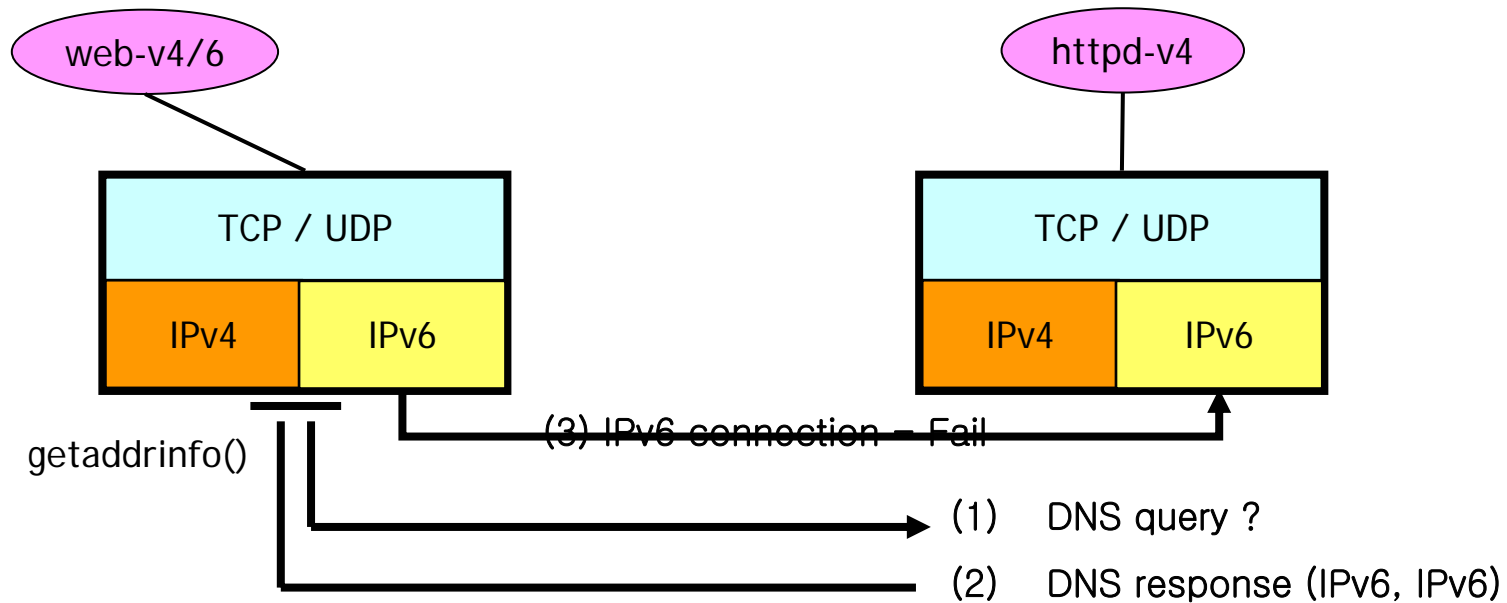
IN A	127.0.0.1
IN A	127.0.0.2
IN AAAA	::1

- BAD EXAMPLE

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

Developing new Applications (3/9)

- ❑ Can't Know the application version by DNS name resolving



Developing new Applications (4/9)

❑ BAD example of Client and Server codes

```

struct addrinfo hints, *res;
...
memset(&hints, 0, sizeof(hints));
hints.ai_flags = AI_PASSIVE;
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo(NULL, SERVICE,
&hints, &res);
...
sockfd = socket(res->family, res->ai_socktype,
res->ai_protocol);
if (sockfd < 0) {
    /* handle socket error */
}
if (bind(sockfd, res->ai_addr, res-
>ai_addrlen) < 0) {
    /* handle bind error */
}
/* ... */
freeaddrinfo(res);

```

Client

```

struct addrinfo hints, *res;
...
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo(SERVER_NODE,
SERVICE, &hints, &res);
if (error != 0) {
    /* handle getaddrinfo error */
}
sockfd = socket(res->family, res->ai_socktype,
res->ai_protocol);
if (sockfd < 0) {
    /* handle socket error */
}
if (connect(sockfd, res->ai_addr, res-
>ai_addrlen) < 0) {
    /* handle connect error */
}
/* ... */
freeaddrinfo(res);

```

Server

Developing new Applications (5/9)

- Iterated Jobs for Finding the working address
 - In a client code, when multiple addresses are returned from `getaddrinfo()`
 - We should try all of them until connection succeeds
 - When a failure occurs with `socket()`, `connect()`, `bind()`, or some other function
 - The code should go on to try the next address

Developing new Applications (6/9)

TCP Server Application

```
#define MAXSOCK 2
struct addrinfo hints, *res;
...
memset(&hints, 0, sizeof(hints));
hints.ai_flags = AI_PASSIVE;
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

error = getaddrinfo(NULL, SERVICE, &hints, &res);
...
for (aip=res; aip && nsock < MAXSOCK; aip=aip->ai_next) {
    sockfd[nsock] = socket(aip->ai_family, aip->ai_socktype,
                          aip->ai_protocol);

    if (sockfd[nsock] < 0) {
        switch errno {
            case EAFNOSUPPORT:
            case /* ... */
                if (aip->ai_next)
                    continue;
                else {
                    ...
                }
        }
    }
}
```

Developing new Applications (7/9)

```
else {
    int on = 1;
    /* optional: works better if dual-binding to wildcard address */
    if (aip->ai_family == AF_INET6) {
        setsockopt(sockfd[nsock], IPPROTO_IPV6, IPV6_V6ONLY,
            (char *)&on, sizeof(on));
        /* errors are ignored */
    }
    if (bind(sockfd[nsock], aip->ai_addr,
        aip->ai_addrlen) < 0) {
        /* handle bind error */
        close(sockfd[nsock]);
        continue;
    }
    if (listen(sockfd[nsock], SOMAXCONN) < 0) {
        /* handle listen errors */
        close(sockfd[nsock]);
        continue;
    }
}
nsock++;
}
```

```
freeaddrinfo(res);
```

Developing new Applications (8/9)

```
struct addrinfo hints, *res, *aip;  
int sockfd, error;
```

TCP Client Application

```
memset(&hints, 0, sizeof(hints));  
hints.ai_family = AF_UNSPEC;  
hints.ai_socktype = SOCK_STREAM;
```

```
error = getaddrinfo(SERVER_NODE, SERVICE, &hints, &res);
```

```
...
```

```
for (aip=res; aip; aip=aip->ai_next) {
```

```
    sockfd = socket(aip->ai_family, aip->ai_socktype, aip->ai_protocol);
```

```
    if (sockfd < 0) {  
        switch errno {  
            case EAFNOSUPPORT:  
            case EPROTONOSUPPORT:  
                if (aip->ai_next)  
                    continue;  
                else {  
                    /* handle unknown protocol errors */  
                    break;  
                }  
        }  
    }
```

```
...
```

```
}
```

Developing new Applications (9/9)

```
else {
    if (connect(sockfd, aip->ai_addr, aip->ai_addrlen) == 0)
        break;

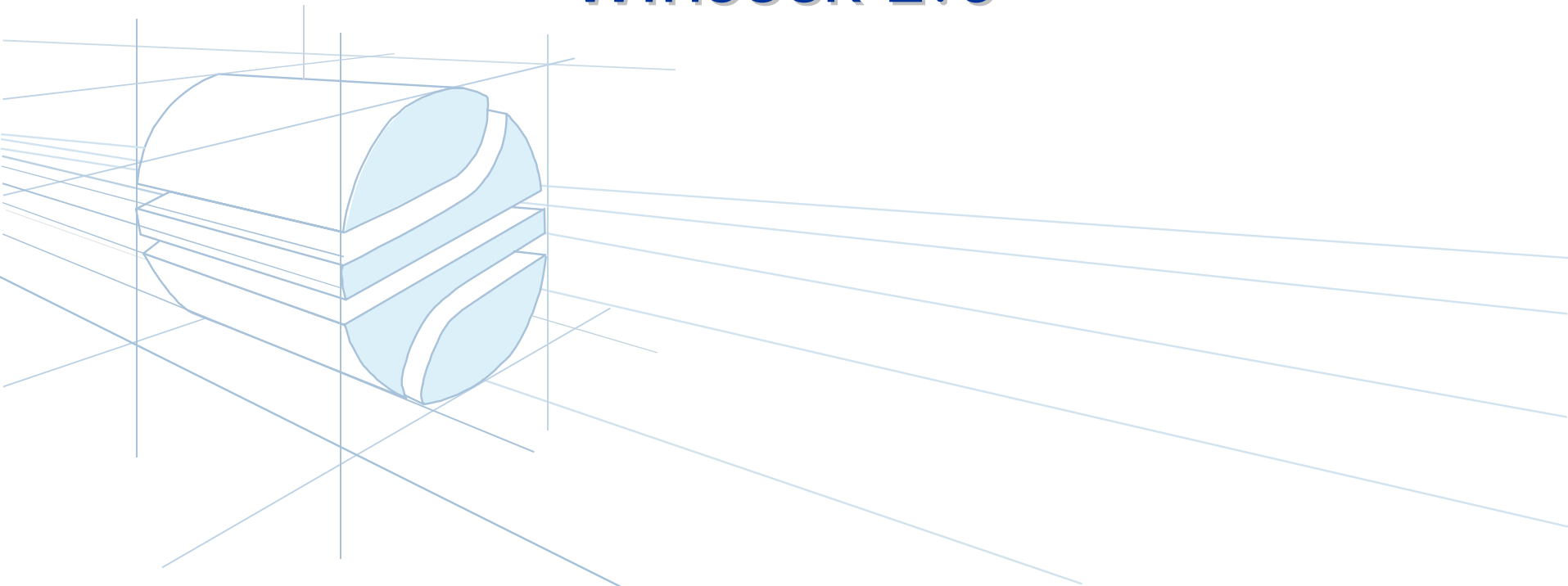
    /* handle connect errors */
    close(sockfd);
    sockfd=-1;
}

if (sockfd > 0) {
    /* socket connected to server address */

    /* ... */
}
```

```
freeaddrinfo(res);
```

IPv6 Socket Programming with WinSock 2.0



Requirements for IPv6 programming

IPv6 stack

- Windows XP + SP1
- Windows 2000 + IPv6 Technical Preview

Compiler

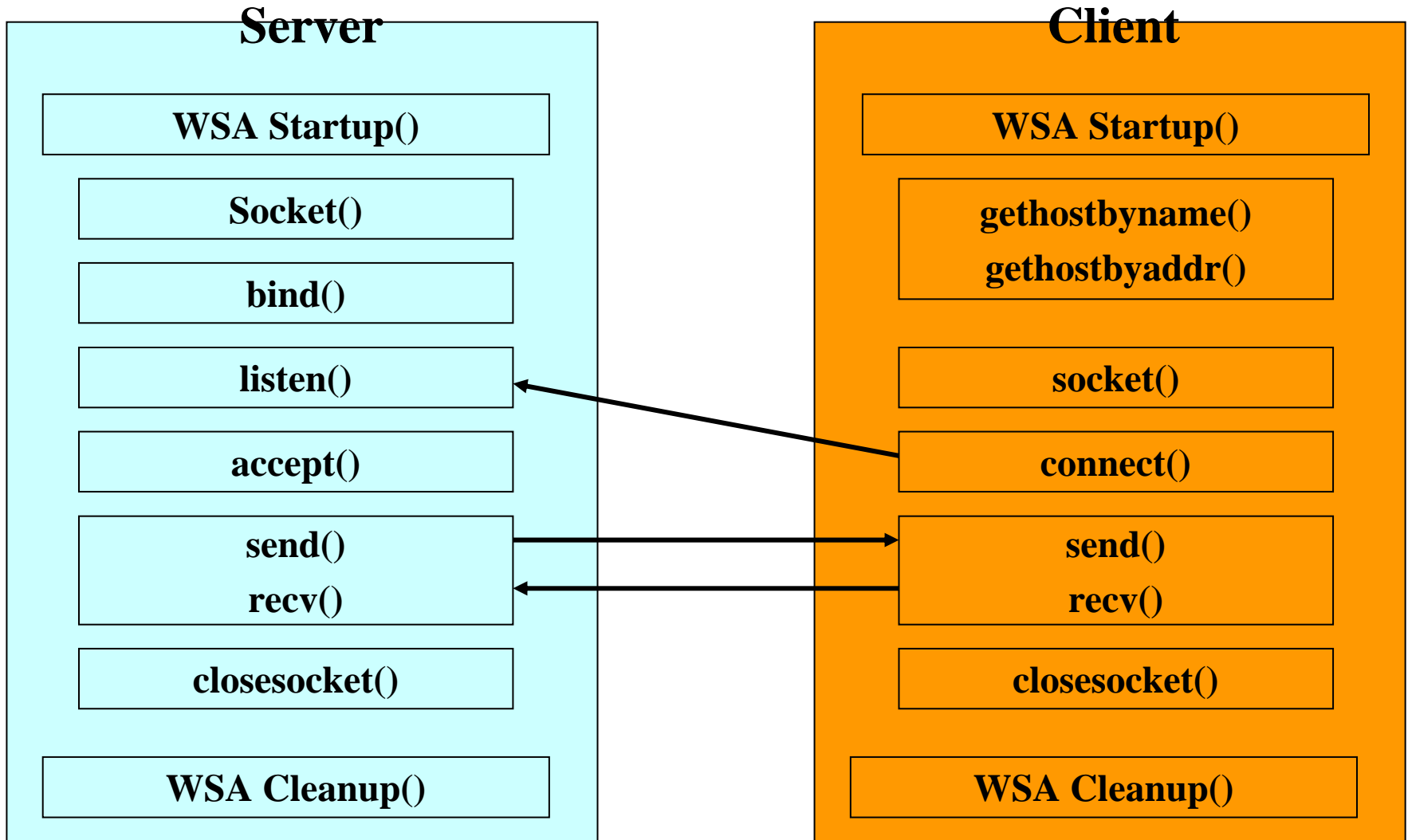
- Visual C++

Platform SDK

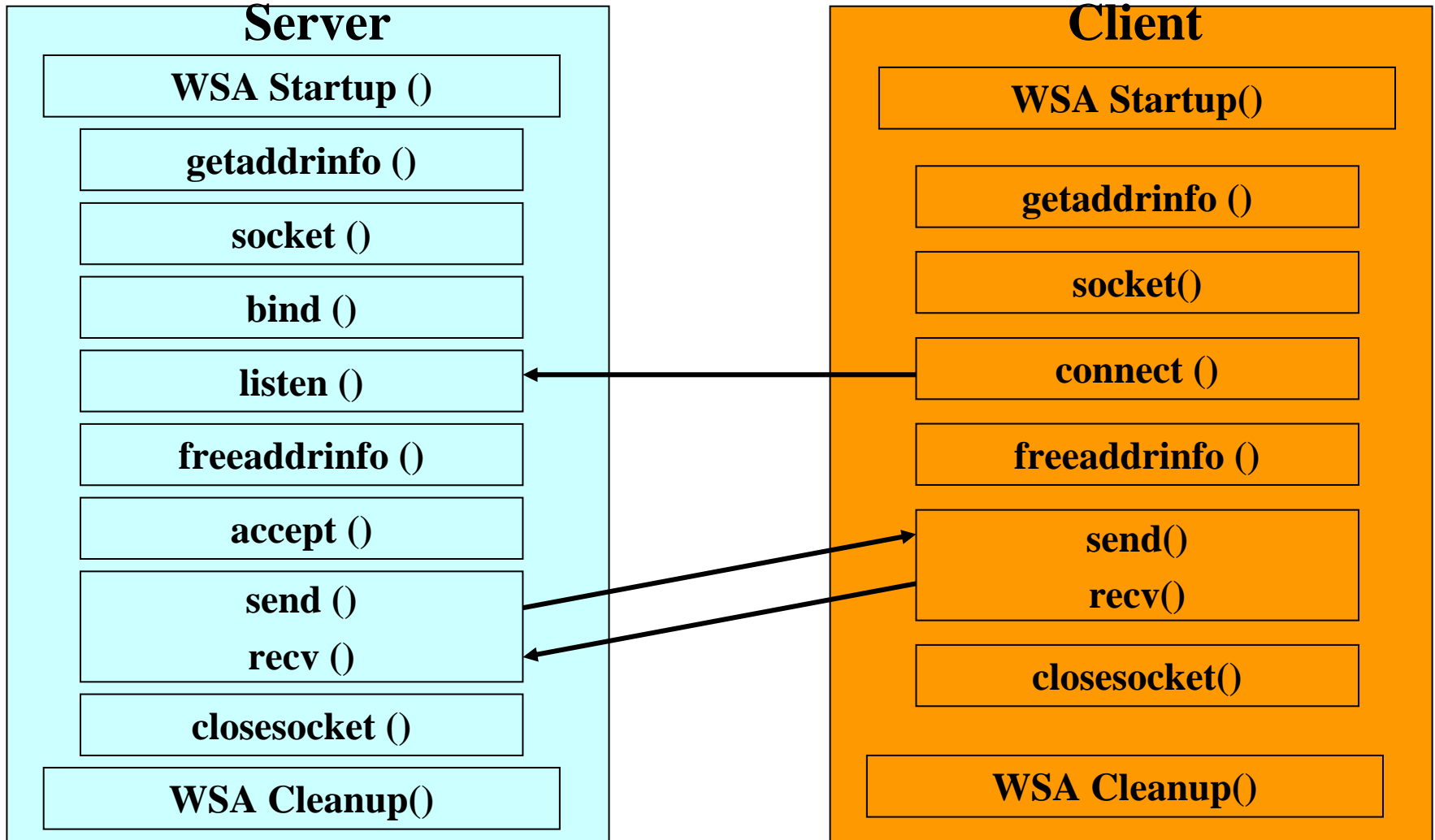
- Core SDK is only required

[http://www.microsoft.com/msdownload/platforms
dk/sdkupdate/](http://www.microsoft.com/msdownload/platformsdk/sdkupdate/)

IPv4 Application



IPv6 Application



Sample Sever Code for IPv6 (1/3)

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <wspiapi.h>
```

// #include <tpipv6.h> For IPv6 Tech Preview.

(addition)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
...
int __cdecl main(int argc, char **argv)
{
    struct addrinfo hints, *results, *AI;
    SOCKADDR_STORAGE From;
```

```
...
// Load Winsock
if ((retval = WSStartup(MAKEWORD(2, 2), &wsaData)) != 0)
{
    // exit program
}
```

(1)

```
...
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET6;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
// If interface is NULL then request the passive "bind" address
hints.ai_flags = ((interface == NULL) ? AI_PASSIVE : 0);
```

(2)

```
retval = getaddrinfo(interface, port, &hints, &results);
if (retval != 0)
{ // getaddrinfo fail; }
```

Sample Sever Code for IPv6 (2/3)

```
for (i = 0, AI = result; AI != NULL; AI = AI->ai_next, i++) {
```

```
// Create the socket according to the parameters returned
```

```
server_sockets[i] = socket(AI->ai_family, AI->ai_socktype, AI->ai_protocol );
```

```
if (server_sockets[socket_count] == INVALID_SOCKET){
    fprintf(stderr, "socket failed: %d\n", WSAGetLastError());
    continue;
}
```

(3)

```
// Bind the socket to the address returned
```

```
retval = bind(server_sockets[i], AI->ai_addr, (int)AI->ai_addrlen);
```

```
if (retval == SOCKET_ERROR) {
    fprintf(stderr, "bind failed: %d\n", WSAGetLastError());
    continue;
}
```

(4)

```
// If a TCP socket, call listen on it
```

```
if (AI->ai_socktype == SOCK_STREAM) {
```

```
    if ( listen(server_sockets[i], 5) == SOCKET_ERROR) {
        continue;
    }
}
```

(5)

```
}
```

```
...
```

```
}
```

```
freeaddrinfo(results);
```

(6)

Sample Sever Code for IPv6 (3/3)

```
while(1) {
    FromLen = sizeof(From);
    ...
    sc = accept(s, (LPSOCKADDR *)&From, &Fromlen);
    if (sc == INVALID_SOCKET)
    {
        fprintf(stderr, "accept failed: %d\n", WSAGetLastError());
        return -1;
    }
    while (1) {
        bytecount = recv(sc, Buffer, DEFAULT_BUFFER_LEN, 0);
        if (bytecount == SOCKET_ERROR)
        {
            fprintf(stderr, "recv failed: %d\n", WSAGetLastError());
            return -1;
        }
        ...
        bytecount = send(sc, Buffer, bytecount, 0);
        if (bytecount == SOCKET_ERROR)
        {
            fprintf(stderr, "send failed: %d\n", WSAGetLastError());
            return -1;
        }
    }
}
...
closesocket(sc);
WSACleanup();
```

(7)

(8)

(8)

(9)

(10)

IPv6 stack Installation (1/2)

□ IPv6 stack installation

○ Microsoft

- Windows 95, 98
 - ✓ Trumpet (<http://www.trumpet.com.au/ipv6.htm>)
- Windows 2000
 - ✓ MSR IPv6 stack
(<http://www.research.microsoft.com/msripv6/msripv6.htm>)
 - ✓ Technical Preview IPv6 stack
(<http://msdn.microsoft.com/downloads/sdks/platform/tpipv6.asp>)
- Windows XP or 2003
 - ✓ Include
 - ✓ On Dos command, input 'netsh interface ipv6 install'

○ Installation Guide in Korean

- <http://www.vsix.net/customer/ipv6Install.jsp>

IPv6 stack Installation (2/2)

□ IPv6 stack installation (Cont'd)

○ Linux

- Linux Kernels version 2.2 and above ship with an IPv6

○ BSD

- FreeBSD 4.0-RELEASE (and more recent) integrates KAME IPv6 stack, and shipped with IPv6 turned on by default
- Support KAME
 - ✓ FreeBSD 2.2.8 + KAME
 - ✓ FreeBSD 3.4 + KAME
 - ✓ FreeBSD 4.0+ KAME

○ Solaris

- Solaris 8 ships with an IPv6 implementation built in.

Other References (1/3)

❑ IPv6 Sample codes

- Provide by Microsoft Platform SDK
- C:\Program Files\Microsoft SDK\Samples\netds\WinSock\

❑ Visual C++ Setting

- 'Tools' / 'Options' ... -> 'Directory' tap, setting SDK, which is downloaded, to 'Include' and 'library'
 - C:\Program Files\Microsoft SDK\include
 - C:\Program Files\Microsoft SDK\lib
- Setting up for WS2_32.lib
 - 'Project' / 'Settings' -> In 'Link' tap, write down "WS2_32.lib" in 'Object/library modules:' window.
 - ✓ error LNK2001: unresolved external symbol xxx

Other References (2/3)

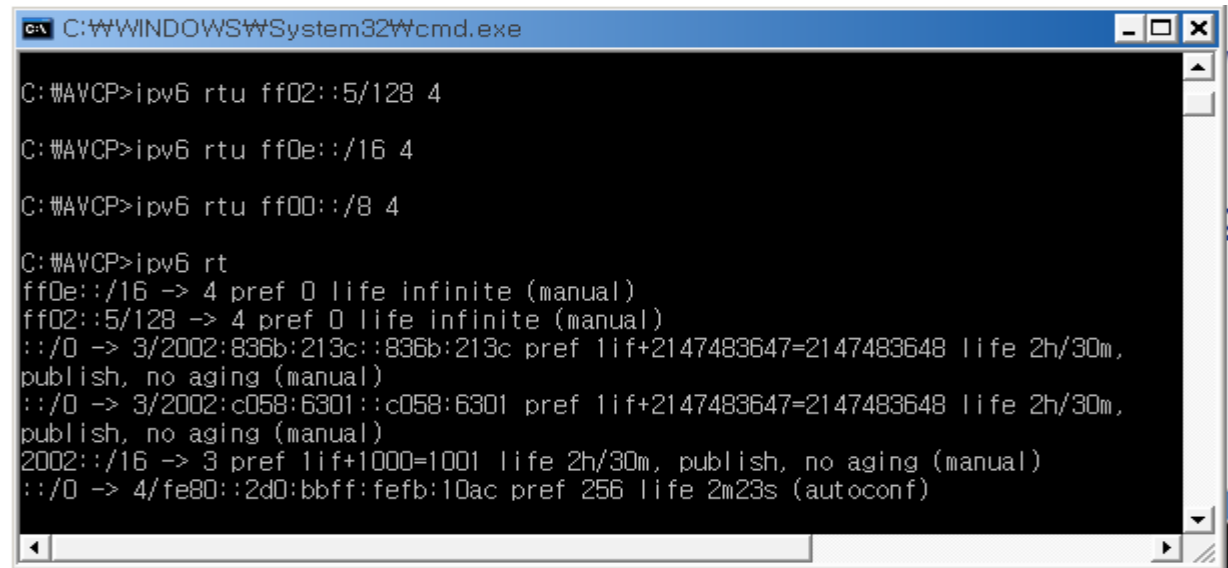
□ Setup Multicast Address

○ Adding multicast prefix to routing table in Windows

- ipv6 rtu ff02::5/128 4
- ipv6 rtu ff0e::/16 4
- ipv6 rtu ff00::/8 4

○ Confirm

- “ipv6 rt”



```
C:\WINDOWS\System32\cmd.exe
C:\#AVCP>ipv6 rtu ff02::5/128 4
C:\#AVCP>ipv6 rtu ff0e::/16 4
C:\#AVCP>ipv6 rtu ff00::/8 4

C:\#AVCP>ipv6 rt
ff0e::/16 -> 4 pref 0 life infinite (manual)
ff02::5/128 -> 4 pref 0 life infinite (manual)
::/0 -> 3/2002:836b:213c::836b:213c pref 11f+2147483647=2147483648 life 2h/30m,
publish, no aging (manual)
::/0 -> 3/2002:c058:6301::c058:6301 pref 11f+2147483647=2147483648 life 2h/30m,
publish, no aging (manual)
2002::/16 -> 3 pref 11f+1000=1001 life 2h/30m, publish, no aging (manual)
::/0 -> 4/fe80::2d0:bbff:febf:10ac pref 256 life 2m23s (autoconf)
```

Other References (3/3)

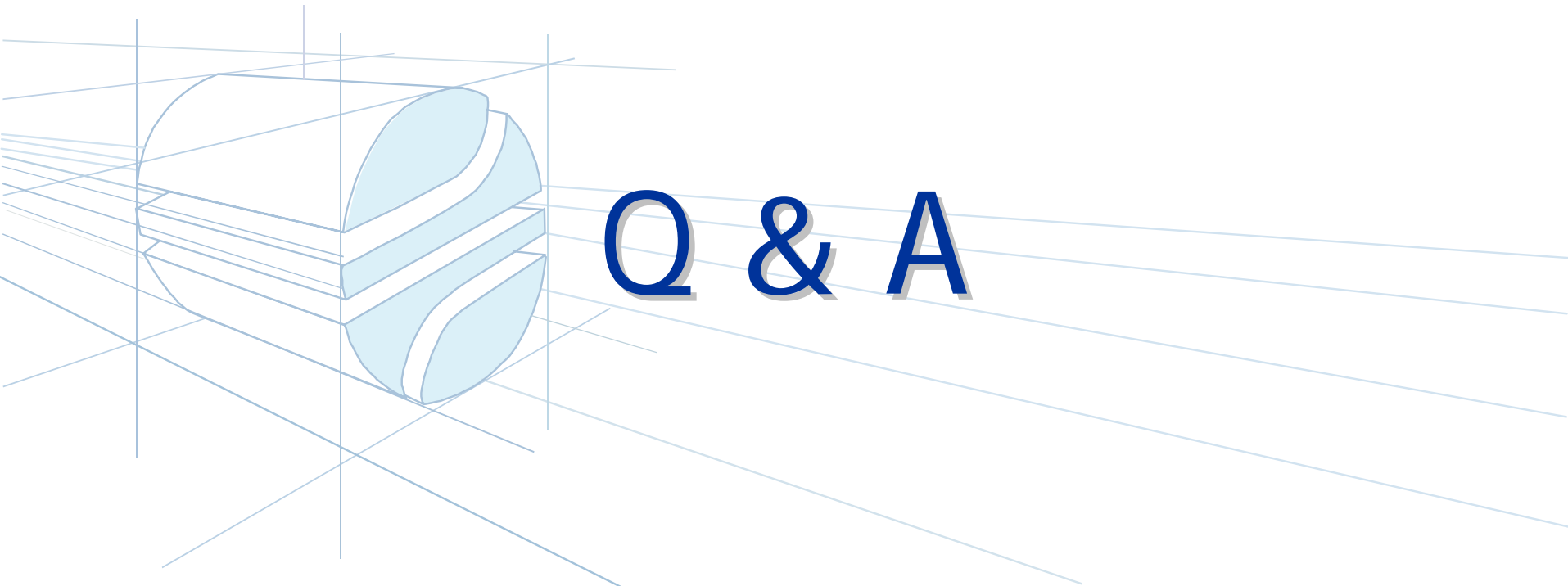
❑ The Problem of IPv6 Multicast in Windows

- The Windows XP case can not receive Multicast packet without Service Pack #1
- The case of Windows 2000 Technical Preview is unstable
 - Sometimes the part of IPv6_JOIN_GROUP occurs error!

❑ Solution

- We should install service pack in Windows XP
- Attention
 - IPv6 multicast can not operate with "Advanced Networking pack for windows XP"

Thank you !!



Q & A