IPv6 Standard Application Programming

- APT-NAv6 Center Joint Workshop on IPv6 -

USAGI/WIDE Project / Keio University
YOSHIFUJI Hideaki
<yoshfuji@linux-ipv6.org>
<hideaki@yoshifuji.org>

Goal of This Session

- Learn Standard, Basic Socket API (RFC3493)
  - Overview of specification
  - Question / Answers / Discussion
- Learn Advanced Socket API (RFC3542)
  - If time allows
  - Overview of specification

Slides and sample codes will be available at:

http://www.linux-ipv6.org/materials/200603-APT-NAv6-IPV6-WORKSHOP/

Table of Contents

- IPv6-aware Network Application Programming
  - Basic Socket API(RFC3493)
  - Advanced Socket API(RFC3542)

Socket API Extensions for IPv6

- Sockets Interface
  - The de-facto standard Application Programming Interface (API) for TCP/IP applications
  - developed for Unix in the early 1980s, also been implemented on a wide variety of non-Unix systems
  - TCP/IP applications written using the sockets API have in the past enjoyed a high degree of portability
- For similar portability with IPv6 applications
  - RFC3493 (Basic Socket Interface Extensions for IPv6)
  - RFC3542 (Advanced Socket API for IPv6)

RFC3493

Basic Socket Interface Extensions for IPv6

- Source and binary compatibility
  - Changes do not break existing programs
- Minimum changes to the API
  - Simplify the task of conversion
- Interoperability with IPv6 and IPv4 hosts
  - Applications do not need to know which type of host they are communicating with
- 64-bit allignment
- MT-Safe

Compoments of Basic API Extentions

- Core socket functions
- Address data structures
- Name-to-address translation functions
- Address conversion functions

Core Socket Functions (1)

```
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *myaddr,
         socklen_t addrlen);
int connect(int sockfd, const struct sockaddr *serv_addr,
            socklen_t addrlen);
int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);
int listen(int sockfd, int backlog);
int getsockopt(int sockfd, int level, int optname,
               void *optval, socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

Core Socket Functions (2)

- Protocol independent framework
  - Protocol and/or address family number
  - Pointer to the socket address structure (via opaque sockaddr{}) and its length
- protocol/address family number and protocol-specific socket address structure defined for each protocol
- No need to change this framework
  - New address family: AF_INET6
    - Usually, PF_INET6 equals to AF_INET6
  - New socket options to support new functions

Address Structure(1)

- IPv6 address structure

```
struct in6_addr {
  uint8_t s6_addr[16];
};
```

  - Trivial, no scope information
- Question 1
  - Definition on your system
  - What you should note

Address Structure(2)

- IPv6 socket address structure

```
struct sockaddr_in6 {
  sa_family_t     sin6_family;    /* AF_INET6 */
  in_port_t       sin6_port;      /* port number */
  uint32_t        sin6_flowinfo;  /* flow information */
  struct in6_addr sin6_addr;      /* IPv6 address */
  uint32_t        sin6_scope_id;  /* Scope Identifier */
};
```

  - Used in (most of) socket API
  - AF_INET6
    - IPv6 address family number
    - In most systems, PF_INET6 is defined as AF_INET6
  - IPv6 address alligned on 64-bit boundary
  - Flow information and spope information

Address Structure(3)

- Storage for all socket address structures

```
#define _SS_MAXSIZE    128                    /* maxumum size */
#define _SS_ALIGNSIZE  (sizeof (int64_t))     /* desired alignment */

#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 */
   char         __ss_pad1[_SS_PAD1SIZE];
   int64_t      __ss_align;               /* force allignment */
   char         __ss_pad2[_SS_PAD2SIZE];
};
```

  - Storage for all socket address structures on the system
  - aligned on 64-bit boundary (w/ systems supporting IPv6)

Question 2: Address Structures

a) Look into the actual definitions of in6_addr{}, sockaddr_in6{}
   - netinet/in.h

b) Discuss what we should note

Question 3: Core Socket Functions

- Refer the following and make it support IPv6

```
int s;
struct sockaddr_in sn;
s = socket(AF_INET, SOCK_STREAM, 0);
/* setup sn */
if (connect(s, (struct sockaddr*)sn, sizeof(sn)) < 0)
    perror("connect");
/* ... */
```

Trivial Usage of Core Socket API

- IPv4:

```
int s;
struct sockaddr_in sn;
s = socket(AF_NET, SOCK_STREAM, 0);
/* setup sn */
connect(s, (struct sockaddr*)sn, sizeof(sn));
/* ... */
```

- IPv6:

```
int s;
struct sockaddr_in6 sn;
s = socket(AF_INET6, SOCK_STREAM, 0);
/* setup sn */
connect(s, (struct sockaddr*)sn, sizeof(sn));
/* ... */
```

Compatibility with IPv4 Nodes

- ::ffff:<IPv4-address>
  - IPv4-mapped address
  - All IPv4 addresses are mapped on IPv6 address space

    e.g. 192.168.0.1 => ::ffff:192.168.0.1
- Providing ability for IPv6 applications to interoperate with IPv4 applications
  - Specify the destination, to connect() or sendto()
  - Getting peer information via accept(), recvfrom(), getpeername()
  - Do not need to open socket for each protocol
- Note: IPv4-mapped address is not always available...

IPv4-mapped Address Pros/Cons

- Pros
  - Easy to convert IPv4 applications to IPv6
- Cons
  - Complexity of kernel
  - Complexity of access control
    - People usually build access control using IPv4 address
  - Note: “Some” systems do not support this...
    - OpenBSD, NetBSD (by default), FreeBSD 6.x (by default), Windows

Special Address Definitions

- IPv6 Wildcard Address(::)
  - const struct in6_addr in6addr_any;
    - sin6.sin6_addr = in6addr_any; // assignment
  - IN6ADDR_ANY_INIT
    - struct in6_addr in6 = IN6ADDR_ANY_INIT; //initialization
- IPv6 Loopback Address(::1)
  - const struct in6_addr in6addr_loopback;
    - sin6.sin6_addr = in6addr_loopback; //assignment
  - IN6ADDR_LOOPBACK_INIT
    - struct in6_addr in6 = IN6ADDR_LOOPBACK_INIT; //initialization

New Socket Options

- IPV6_UNICAST_HOPS
  - Controls the hop limit used in outgoing unicast IPv6 packets
- IPV6_MULTICAST_IF
  - Set the interface to use for outgoing multicast packets
- IPV6_MULTICAST_HOP
  - Set the hop limit to use for outgoing multicast packets
  - Default: 1
- IPV6_MULTICAST_LOOP
  - Loopback a copy of the datagram if the host itself joins the destination group of the outgoing multicast packet.
  - Default: 1
- IPV6_JOIN_GROUP / IPV6_LEAVE_GROUP
  - Joint / Leave a multicast group on a specified local interface
  - Note: New IGMPv3/MLDv2 Interface is also available
- IPV6_V6ONLY
  - Restricts AF_INET6 sockets to IPv6 communication only
  - Default: 0
    - Note: on some systems, default is 1.

Requirements of Name-Conversion Functions

- Protocol Independent
- Flexibility
- MT-Safety

Historical Name-vs-Address Translation Functions

- Name to address

```
struct hostent *gethostbyname(const char *name);
```

- Address to name

```
struct hostent *gethostbyaddr(const char *addr,
                              int len, int type);
```

- Host entry structure

```
struct hostent {
  char  *h_name;        /* official name */
  char **h_aliases;     /* alias list */
  char   h_addrtype;    /* type of address: AF_xxx */
  int    h_length;      /* length of the address: 4 for IPv4 */
  char **h_addr_list;   /* list of addresses from name server */
};
```

- Issues
  - The structure itself is af-independent, but gethostbyname().
  - Unflexible – no searching options
  - Not MT-safe

Historical Name-vs-Address Translation Functions (2)

- Name to address

```
struct hostent *getipnodebyname(const char *name, int af, int flags,
                                int *error_num);
```

- Address to name

```
struct hostent *getipnodebyaddr(const void *src, size_t len, int af,
                                int *error_num);
```

Pros

- Support both IPv6/IPv4 in similar semantics of gethostby{name,addr}()
- Searching options
  - AI_DEFAULT to the flags for standard query
- Numeric address, IPv4-mapped address
- MT-Safe
  - Free memory be freehostent()

Cons

- No scopes
- Uneasy to create socket
- Not widely deployed, deprecated now

Question 4

- Convert obs-client.c and obs-server.c to support IPv6
  - Hint: Use AI_DEFAULT for flags

Name-vs-Address Translation Functions (Protocol Independent)

- Name to address

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

- Address to name

```
int getnameinfo(const struct sockaddr *sa,
                socklen_t salen,
                char *host, size_t hostlen,
                char *serv, size_t servlen,
                int flags);
```

getaddrinfo()

```
int getaddrinfo(const char *nodename, const char   *servname,
                const struct addrinfo *hints,
                struct addrinfo **res);
struct addrinfo {
    int              ai_flags;     /* flags */
    int              ai_family;    /* protocol family PF_xxx */
    int              ai_socktype;  /* socket type SOCK_xxx */
    int              ai_protocol;  /* protocol type; IPPROTO_xxx in IP */
    socklen_t        ai_addrlen;   /* length of socket address structure */
    char            *ai_canonname; /* canonical name of the node */
    struct sockaddr *ai_addr;      /* socket address structure */
    struct addrinfo *ai_next;      /* next address information in the list */
};
```

- Protocol independent Name-to-address translation function
- Search by node name, service name and searching options
- Socket address structure instead of raw address structure
  - Scope information
- MT-Safe

Options of getaddrinfo() (1)

- ai_flags
  - AI_PASSIVE: socket address returned from the function will be used for bind().
    - switch for nodename==NULL
  - AI_CANONNAME: Request canonical name of the node
    - Result stored in the first element in the list
  - AI_NUMERICHOST / AI_NUMERICSERV: Assume nodename / servname as numeric address and do not look up nodename / servname
  - AI_ADDRCONFG: Search for addresses if local address for the corresponding protocol is available

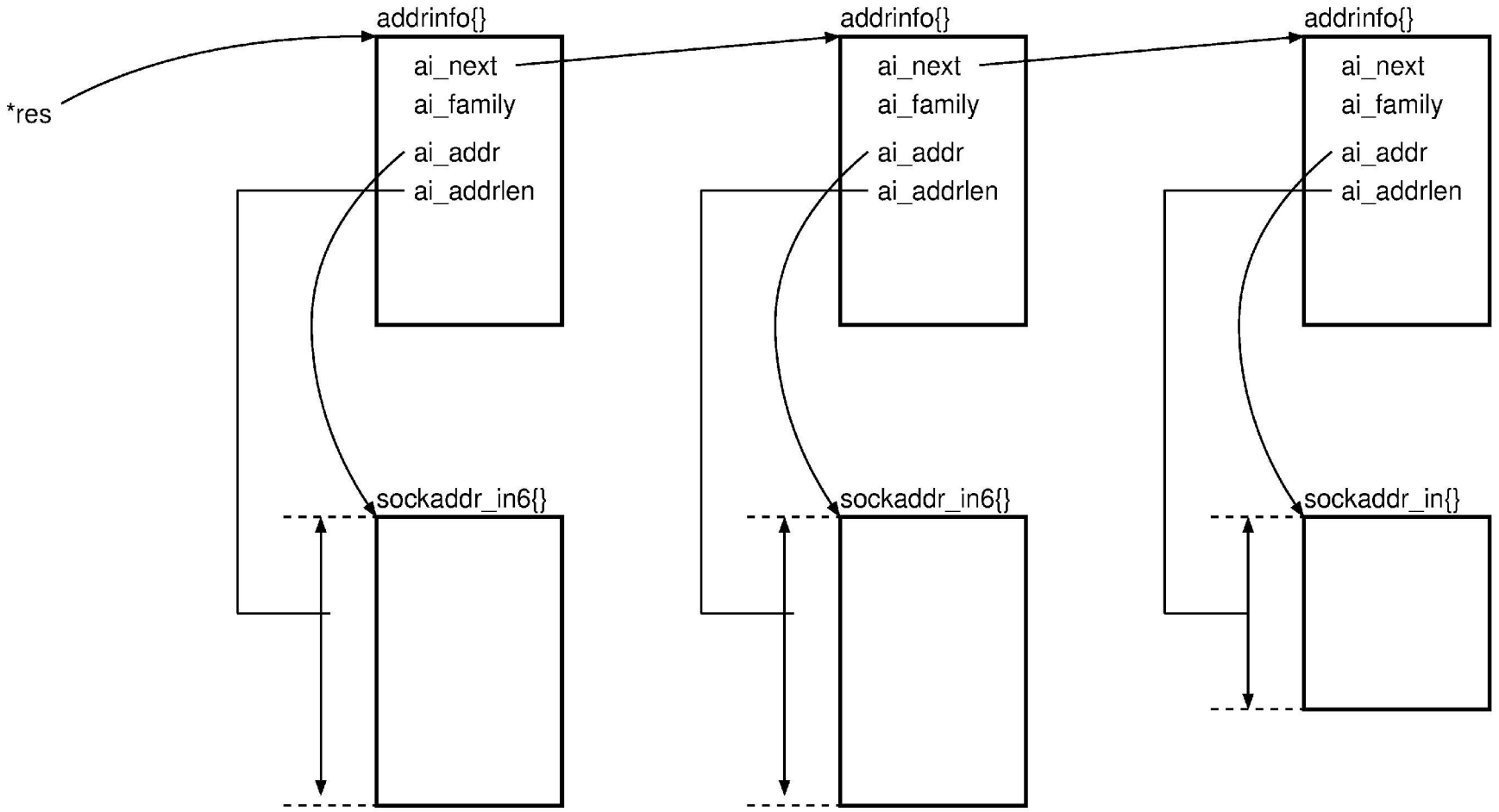
Options of getaddrinfo() (2)

- Searching options
  - NULL in nodename denotes loopback or wildcard
    - wildcard if AI_PASSIVE, otherwise loopback
    - numeric address allowed(e.g. “::1”)
  - do not search service if servname is NULL
    - numeric service allowed(e.g. “80”)
  - AF_UNSPEC in ai_family, 0 in ai_socktype, ai_protocol means caller does not care.

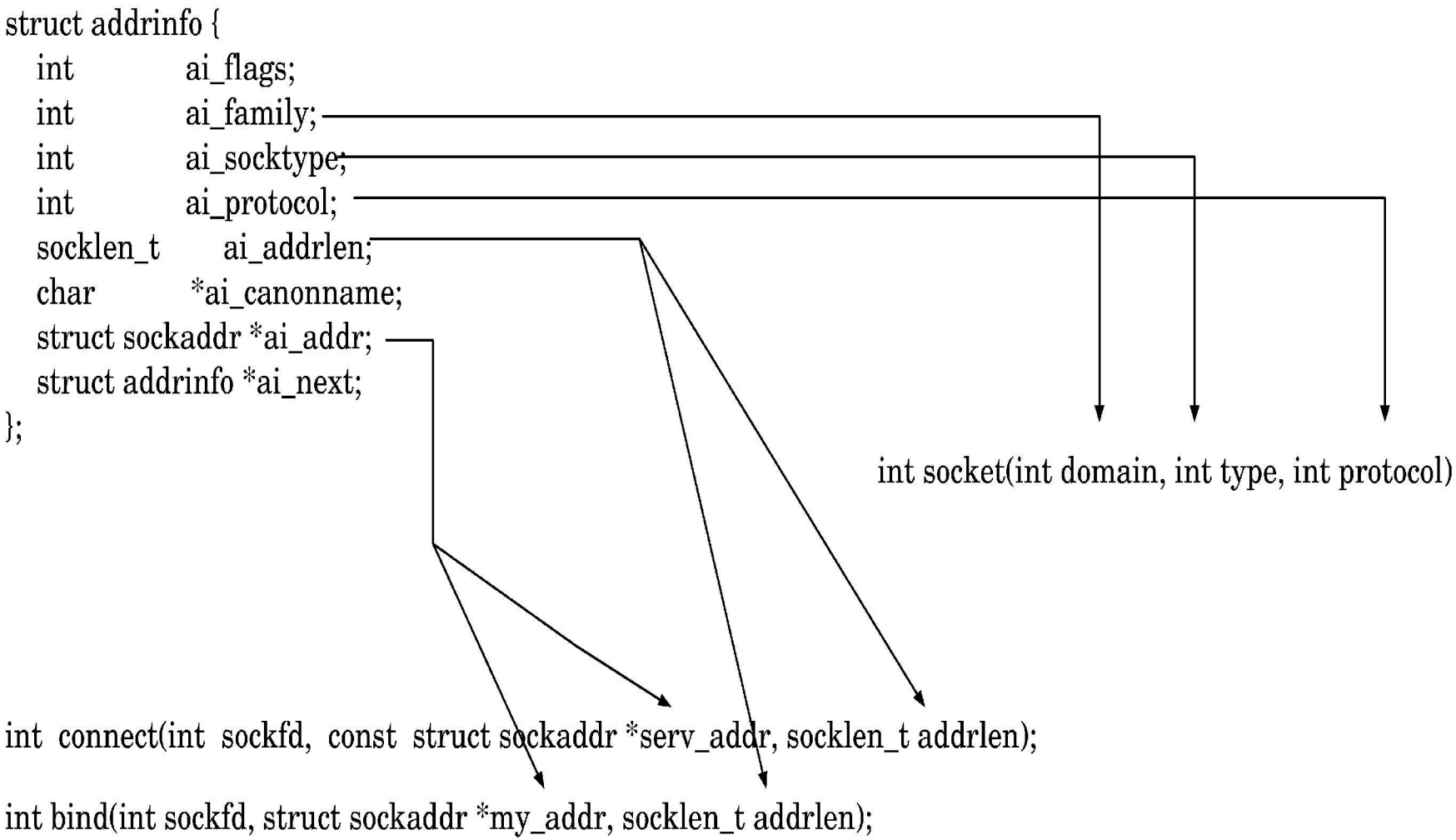
Result of getaddrinfo() (1)

- Return value
  - 0 if succeeded
    - Dynamically allocated result returned via res
      - freeaddrinfo(): free result
  - Otherwise getaddrinfo-specific error code
    - EAI_FAMILY, EAI_NONAME, EAI_SERVICE etc.
    - gai_strerror(): human readable string for error code
  - Result comes with information required to create socket and to connect (or to bind)
    - Address family, socket type, protocol, and length of the socket address structure
    - Yield differences among address families / protocols

Result of getaddrinfo() (2)



Usage of addrinfo{}



Example of getaddrinfo() for Clients

```
struct addrinfo hints, *res, *ai;
int s;
memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
if (gai = getaddrinfo(name, service, &hints, &res)) {
    printf("getaddrinfo failed: %s\n", gai_strerror(gai)); return -1;
}
for (ai = res; ai != NULL; ai = ai->ai_next) {
    s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (s < 0) continue;
    if (connect(s, ai->ai_addr, ai->ai_addrlen) < 0) {
        close(s);
        s = -1;
        continue;
    }
    break;
}
freeaddrinfo(res);
return s;
```

Example of getaddrinfo() for Servers

```
struct addrinfo hints, *res, *ai;
int s;

memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE;
gai = getaddrinfo(name, service, &hints, &res);
if (gai) {
    printf("getaddrinfo failed: %s\n", gai_strerror(gai));
    return NULL;
}
```

Example of getaddrinfo() for Servers (cont'ed)

```
for (ai = res; ai != NULL; ai = ai->ai_next) {
    s = socket(ai->ai_family, ai->ai_socktype,
               ai->ai_protocol);
    if (s < 0) continue;
    if (bind(s, ai->ai_addr, ai->ai_addrlen) < 0) {
        close(s);
        s = -1;
        continue;
    }
    break;
}
freeaddrinfo(res);
```

Note on Servers

- Semantics (or policy) of bind(2) is very different aming systems
  - AF_INET6 and AF_INET share ports
  - AF_INET6 and AF_INET share ports unless AF_INET6 socket does not set IPV6_V6ONLY
  - AF_INET6 and AF_INET share ports, but automatically separated
- Thus, how to listen is different among systems

getnameinfo()

```
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, socklen_t hostlen,
                char *serv, socklen_t servlen,
                int flags);
#define NI_MAXHOST  1025
#define NI_MAXSERV    32
```

- Address-to-name translation function
- Extract node name, service name from socket address structure and put in human readable format
- Not an address structure but a socket address structure
  - support scope architecture
    - e.g. “ff02::1%link0”, “fec0::1%site0”
- MT-Safe
- NI_MAXHOST, NI_MAXSERV: size enough for host, serv

Options for getnameinfo()

- Search options
  - flags
    - NI_NOFQDN: only the node name portion of the FQDN for local hosts
      - Default: FQDN
    - NI_NAMEREQD: return an error if the host's name cannot be located
      - Default: return in numeric form
    - NI_NUMERICHOST: Always return in numeric form
    - NI_NUMERICSERV: Always return in numeric form
    - NI_DGRAM: Look up service for datagram protocol (especially UDP)

Example of getnameinfo()

```
int s; /* socket */
ssize_t cc;
char buf[256];
struct sockaddr_storage ss;
socklen_t sslen = sizeof(ss);
char hbuf[NI_MAXHOST], serv[NI_MAXSERV];

sslen = sizeof(ss);
cc = recvfrom(s, buf, sizeof(buf),
              (struct sockaddr *)&ss, &sslen);
if (cc >= 0 &&
    getnameinfo((struct sockaddr *)&ss, sslen,
                hbuf, sizeof(hbuf), pbuf, sizeof(pbuf),
                NI_NUMERICHOST|NI_NUMERICSERV) == 0) {
        printf("%d bytes from %s port %s\n",
               (int)cc, hbuf, pbuf);
}
```

Question 5: getaddrinfo()

- Check if the given address (or port) is valid numeric address
- Port number is defined as macro
  - #define PORT_HTTP “80”
- Port number is given via variable
  - in_port_t port_http = 80;
- The “loopback” address for given family
- The “unspecified” address for give family

Question 6: getnameinfo()

- Given socket address structure, extract port number
- Given socket address structure, make another socket address structure with different port number

Question 7: Double Reverse Lookup

- Given socket address structure
  - Reverse look it up
  - check if the address is associated with the name
- Discussion

Interfaces

- Name-to-index

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

- Index-to-name

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

- All interfaces names and indexes

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

where

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

- Free memory

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

Free the dynamic memory allocated by if_nameindex()

Address conversion functions

- Binary-to-text

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

- Text-to-binary

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

Note: Only standard Ipv6 dotted-decimal format is accepted; it DOES NOT accept octal numbers, hexadecimal numbers, and fewer than 4 numbers

- Maximum size for address string (incl. NUL)

```
#define INET_ADDRSTRLEN    16
#define INET6_ADDRSTRLEN   46
```

Address Testing Macros

```
/* Test special addresses */
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 *);
```

Summary of Basic Socket API

- RFC3493: Basic Socket Interface Extensions for IPv6
- Core socket functions and address structures
  - PF_INET/AF_INET -> PF_INET6/AF_INET6
  - in_addr{} -> in6_addr{}
  - sockaddr_in{} -> sockaddr_in6{}
- Protocol independent programming
  - getaddrinfo(), getnameinfo()
    - Protocol independent
    - Scope
  - sockaddr_storage{}
    - Storage for all socket address structures

Note on Portability

- Old systems do not have modern functions, such as getaddrinfo()/getnameinfo()
  - Use tiny alternative implementation
    - e.g. OpenSSH
  - GNU autoconf, and preprocessor are your friends

Target of Advanced Socket API

- Basic Socket API
  - For TCP and UDP-based applications
- Advanced Socket API
  - Raw sockets
    - For ICMPv6
    - ping, traceroute
  - IPv6 header, extension headers
    - routing daemons

Advanced Socket API

- Basic constants and structures
- Basic semantic definitions
- Packet information
  - interface, local address, hop limit
- Access to the optional hop-by-hop options, destination options, routing header
- Additional features for improved application portability

IPv6 Structures

- IPv6 Header
  - struct ip6_hdr{}
- Hop-by-hop / destination options headers
  - struct ip6_hbh{}
  - struct ip6_dest{}
- Routing header
  - struct ip6_rthdr{}, struct ip6_rthdr0{}
- Fragment header
  - struct ip6_frag{}

Next Header Values and Protocol Names

- Constants / names for getprotobyname(3)
  - IPPROTO_HOPOPTS / hopopt
  - IPPROTO_IPV6 / ipv6
  - IPPROTO_ROUTING / ipv6-route
  - IPPROTO_FRAGMENT / ipv6-frag
  - IPPROTO_ESP / esp
  - IPPROTO_AH / ah
  - IPPROTO_ICMPV6 / ipv6-icmp
  - IPPROTO_NONE / ipv6-nonxt
  - IPPROTO_DSTOPTS / ipv6-opts

ICMPv6 Structures

- ICMPv6 Header

struct icmp6_hdr{}

- Router Solicitation/Router Advertisement Message

struct nd_router_solicit{}, struct nd_router_advert{};

- Neighbor Solicitation/Neighbor Advertisement Message

struct nd_neighbor_solicit{}, struct nd_neighbor_advert{};

- Redirect Message

struct nd_redirect{};

- Neighbor Discovery Options

struct nd_opt_hdr{};
struct nd_opt_prefix_info{}, struct nd_opt_rd_hdr{}, struct nd_opt_mtu{};

ICMPv6 Type/Code

- Errors
  - ICMP6_DST_UNREACH
    - ICMP6_DST_UNREACH_{NOROUTE,ADMIN,ADDR,NOPORT}
  - ICMP6_PACKET_TOO_BIG
  - ICMP6_TIME_EXCEEDED
    - ICMP6_TIME_EXCEED_{TRANSIT,REASSEMBLY}
  - ICMP6_PARAM_PROB
    - ICMP6_PARAMPROB_{HEADER,NEXTHEADER,OPTION}
- Echo
  - ICMP6_ECHO_REQUEST
  - ICMP6_ECHO_REPLY
- Multicast Listeners Discovery
  - ICMP6_MEMBERSHIP_QUERY
  - ICMP6_MEMBERSHIP_REPORT
  - ICMP6_MEMBERSHIP_REDUCTION

ICMPv6 Neighbor Discovery

- Types
  - ND_ROUTER_SOLICIT
  - ND_ROUTER_ADVERT
  - ND_NEIGHBOR_SOLICIT
  - ND_NEIGHBOR_ADVERT
  - ND_REDIRECT
- Code is always 0
- Options
  - ND_OPT_SOURCE_LINKADDR
  - ND_OPT_TARGET_LINKADDR
  - ND_OPT_PREFIX_INFORMATION
  - ND_OPT_REDIRECTED_HEADER
  - ND_OPT_MTU

Checksum

- Checksum incorporates the IPv6 pseudo-header

```
int  offset = 2;
setsockopt(fd, IPPROTO_IPV6, IPV6_CHECKSUM,
           &offset, sizeof(offset));
```

- offset is where the checksum is located
  - -1 to disable checksumming

Notes

- Only for raw sockets other thant ICMPv6 sockets
- Odd is invalid

ICMPv6 Type Filtering

Filtering ICMPv6 messages by the type field

```
struct icmp6_filter;

/* initialization */
void ICMP6_FILTER_SETPASSALL (struct icmp6_filter *);
void ICMP6_FILTER_SETBLOCKALL(struct icmp6_filter *);

/* pass/filter one-by-one */
void ICMP6_FILTER_SETPASS ( int, struct icmp6_filter *);
void ICMP6_FILTER_SETBLOCK( int, struct icmp6_filter *);

/* test if specified message */
int  ICMP6_FILTER_WILLPASS (int, const struct icmp6_filter *);
int  ICMP6_FILTER_WILLBLOCK(int, const struct icmp6_filter *);
```

Example

```
struct icmp6_filter myfilt;
fd = socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6);
ICMP6_FILTER_SETBLOCKALL(&myfilt);
ICMP6_FILTER_SETPASS(ND_ROUTER_ADVERT, &myfilt);
setsockopt(fd, IPPROTO_ICMPV6, ICMP6_FILTER, &myfilt, sizeof(myfilt));
```

Optional Information in IPv6 and Extension Headers

- Send/Receive interface and source/destination address
- Hop limit
- Next hop address
- Traffic class
- Extension headers
  - Hop-by-hop options header
  - Destination options header(s)
  - Routing header

Access to Optional Information

- Receiver side
  - Ancillary data
- Sender side
  - Sticky options
  - Ancillary data

Receiving Optional Information via Ancillary Data (1)

- Socket options to receive optional information
  - IPV6_RECVPKTINFO
    - interface, local address
  - IPV6_RECVHOPLIMIT
  - IPV6_RECVRTHDR
  - IPV6_RECVHOPOPTS
  - IPV6_RECVDSTOPTS
  - IPV6_RECVTCLASS
- Usage

```
setsockopt(sock, IPPROTO_IPV6, IPV6_xxx, on, sizeof(on));
```

Receiving Optional Information via Ancillary Data (2)

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_IPV6	IPV6_PKTINFO	in6_pktinfo{}
IPPROTO_IPV6	IPV6_HOPLIMIT	int
IPPROTO_IPV6	IPV6_NEXTHOP	sockaddr_in6{}
IPPROTO_IPV6	IPV6_HOPOPTS	ip6_hbh{}
IPPROTO_IPV6	IPV6_DSTOPTS	ip6_dst{}
IPPROTO_IPV6	IPV6_RTHDR	ip6_rthdr{}

Sending Optional Information via Ancillary Data

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_IPV6	IPV6_PKTINFO	in6_pktinfo{}
IPPROTO_IPV6	IPV6_HOPLIMIT	int
IPPROTO_IPV6	IPV6_NEXTHOP	sockaddr_in6{}
IPPROTO_IPV6	IPV6_HOPOPTS	ip6_hbh{}
IPPROTO_IPV6	IPV6_RTHDRDSTOPTS	ip6_dst{}
IPPROTO_IPV6	IPV6_RTHDR	ip6_rthdr{}
IPPROTO_IPV6	IPV6_DSTOPTS	ip6_dst{}

Sending Optional Information via Socket Options

- Sticky options to send optional information
  - IPV6_PKTINFO
  - IPV6_HOPLIMIT
  - IPV6_RTHDR
  - IPV6_HOPOPTS
  - IPV6_DSTOPTS
  - IPV6_RTHDRDSTOPTS
  - IPV6_TCLASS
- Data are the same ones for ancillary data

Helpers for IPv6 Options (1)

- `int inet6_opt_init(void *extbuf, socklen_t extlen);`
  - Initialize buffer data for options header
- `int inet6_opt_append(void *extbuf, socklen_t extlen,`
  `int offset, uint8_t type, socklen_t len,`
  `uint_t align, void **databufp);`
  - Add one TLV option to the option header
- `uint8_t *inet6_opt_finish(void *extbuf, socklen_t extlen,`
  `int offset);`
  - Finish adding TLV options to the options header
- `int inet6_opt_set_val(void *databuf, int offset, void *val,`
  `socklen_t vallen);`
  - Add one component of the option content to the option

Helpers for IPv6 Options (2)

- `int inet6_opt_next(void *extbuf, socklen_t extlen, int offset,`
  `uint8_t *typep, socklen_t *lenp,`
  `void **databufp);`
  - Extract the next option from the options header
- `int inet6_opt_find(void *extbuf, socklen_t extlen, int offset,`
  `uint8_t type, socklen_t *lenp,`
  `void **databufp);`
  - Extract an option of a specified type from the header
- `int inet6_opt_get_val(void *databuf, int offset, void *val,`
  `socklen_t vallen);`
  - Retrieve one component of the option component

Helper for Routing Header (1)

- `size_t inet6_rth_space(int type, int segments);`
  - Return size required for routing header
- `struct cmsghdr *inet6_rth_init(void *bp,`
  `socklen_t bp_len,`
  `int type,`
  `int segments);`
  - Initialize buffer data for routing header
- `int inet6_rth_add(void *bp,`
  `const struct in6_addr *addr);`
  - Add one Ipv6 address to the routing header

Helper for Routing Header (2)

- `int inet6_rth_reverse(const void *in, void *out);`
  - Reverse a routing header
- `int inet6_rth_segments(const void *bp);`
  - Return number of segments in a routing header
- `struct in6_addr *inet6_rth_getaddr(const void *bp, int index);`
  - Fetch one address from a routing header

Path MTU Discovery

- IPV6_USE_MIN_MTU
  - -1: Disable PMTUD for multicast but unicast
  - 0 : Enable PMTUD
  - 1 : Disable PMTUD

  Note: when disabled, IPv6 Minimum Link MTU (1280) is used

Fragmentation

- IPV6_DONTFRAG
- IPV6_RECVMTU / IPV6_MTU
  - To receive Path MTU information

```
struct ip6_mtuinfo{
    struct sockaddr_in6 ip6m_addr;
    uint32_t ip6m_mtu;
};
```

Summary of Advanced Socket API

- Supplement to the Basic API
  - Raw sockets
    - Protocol numbers and names
    - IPv6 header / extension headers
    - Checksumming
    - Path MTU discovery / fragmentation
  - Outgoing/Incoming Interface
  - Additional extensions
    - IN6_ARE_ADDR_EQUAL()
    - rresvport_af(), rcmd_af(), rexec_af()

Question 8

- Discuss ip6_mtuinfo{}
- Discuss positive odd value for IPV6_CHECKSUM

Summary

- Overview of IPv6 Application Programming
- People do not need to know the address structures in detail any longer
- Portability is important
- However, semantics might be different among systems...
- GNU autoconf is your friend