Introduction to **Sockets** Programming in C

Sanjaya Kumar Jena

ITER, Bhubanewar
sanjayjena@soauniversity.ac.in

References

W. Richard Stevens, Bill Fenner, & Andrew M. Rudoff

Unix Network Programming

The Sockets Networking API

Volume-1, Third Edition

Client-Server communication

Server

1. passively waits for and responds to clients
2. passive socket
3. a Telnet server

Client-Server communication

Server

1. passively waits for and responds to clients
2. passive socket
3. a Telnet server

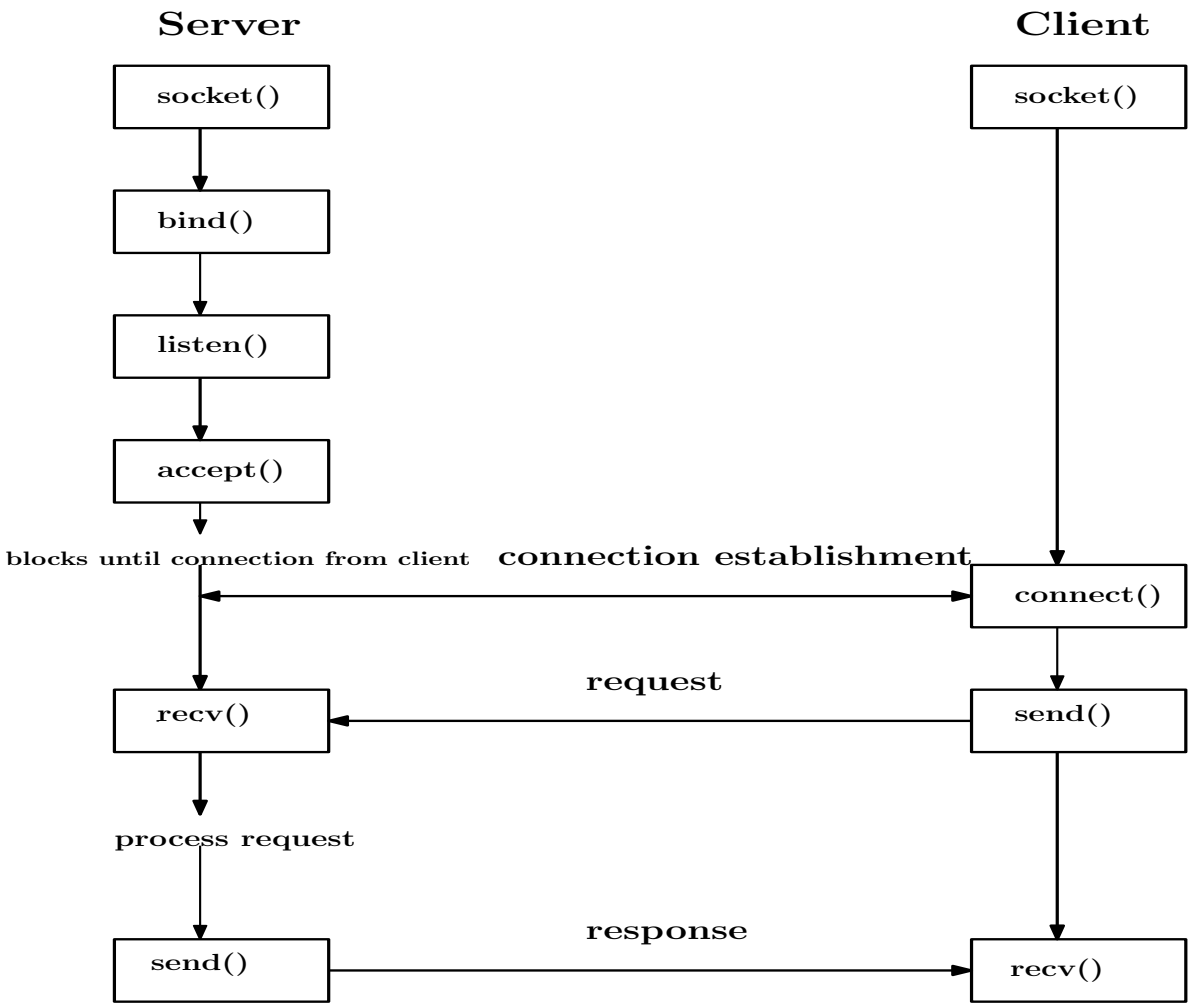
Client

1. initiates the communication
2. must know the address and the port of the server
3. active socket
4. a Telnet client

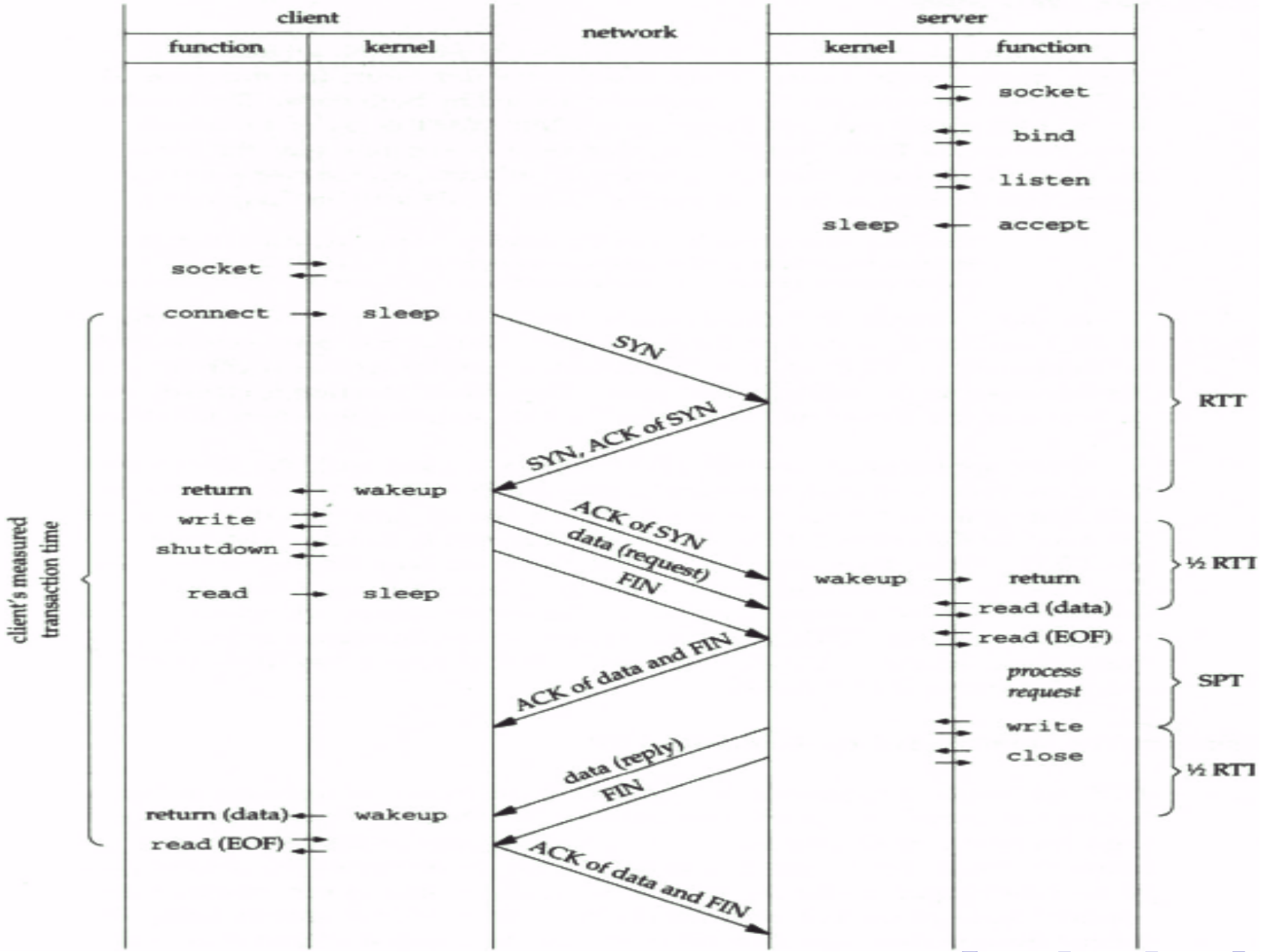
Sockets - Procedures

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Client - Server Communication - Unix



Timeline of TCP Client-Server Interaction



Client - Server Communication - Unix

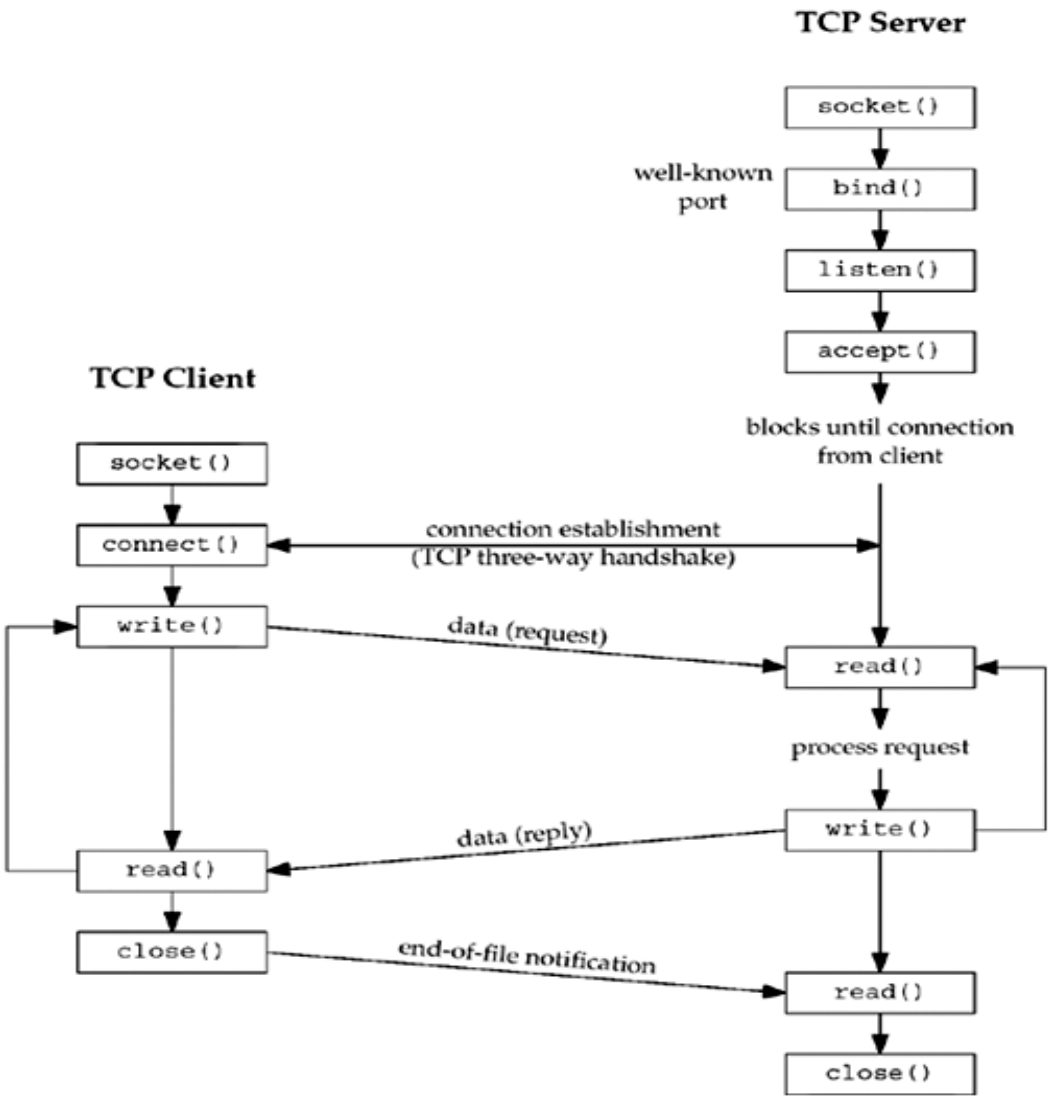


Figure: Socket functions for elementary TCP client/server

Client - Server Communication - Unix

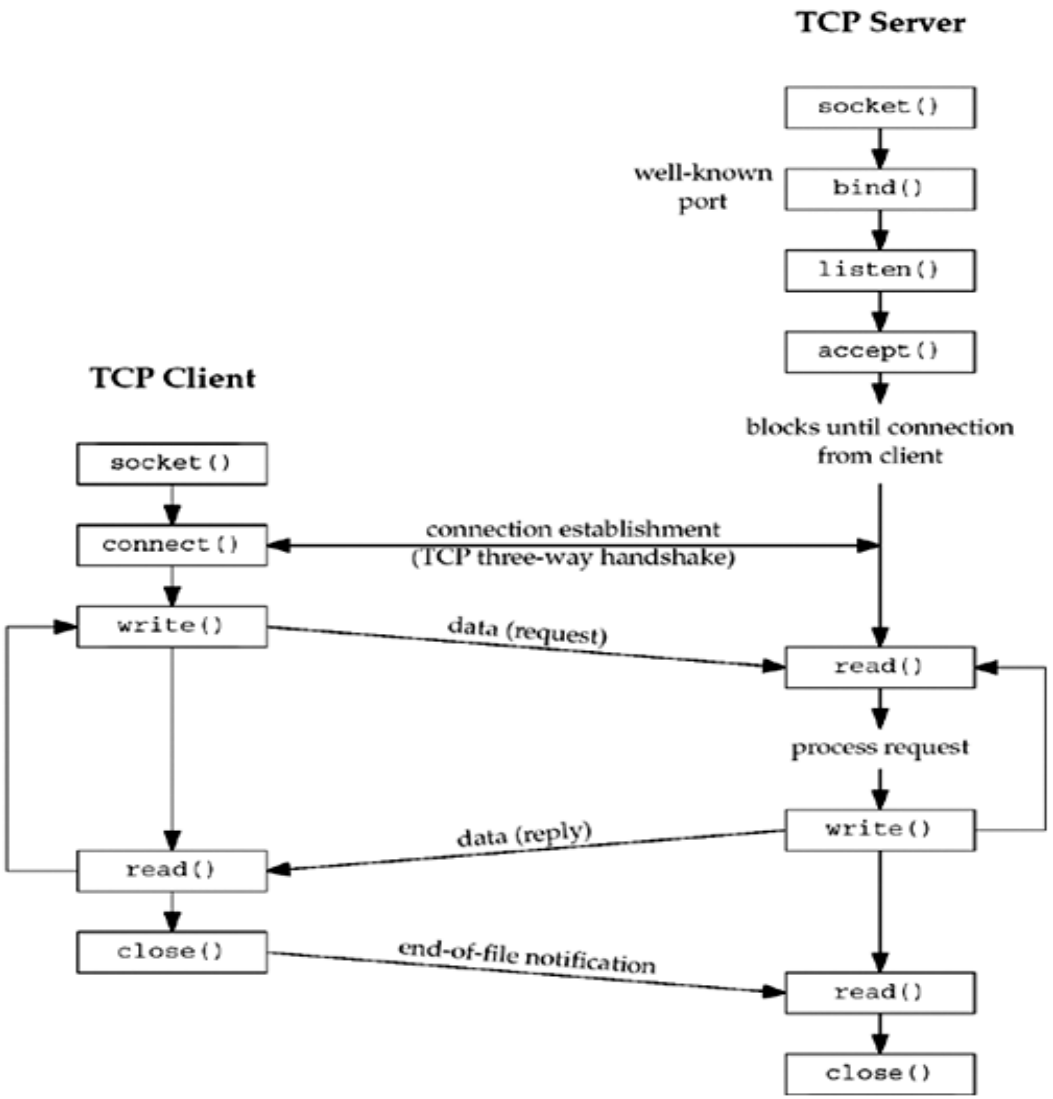


Figure: Socket functions for elementary TCP client/server

First, the server is started, then sometimes later, a client is started that connects to the server.

Client - Server Communication - Unix

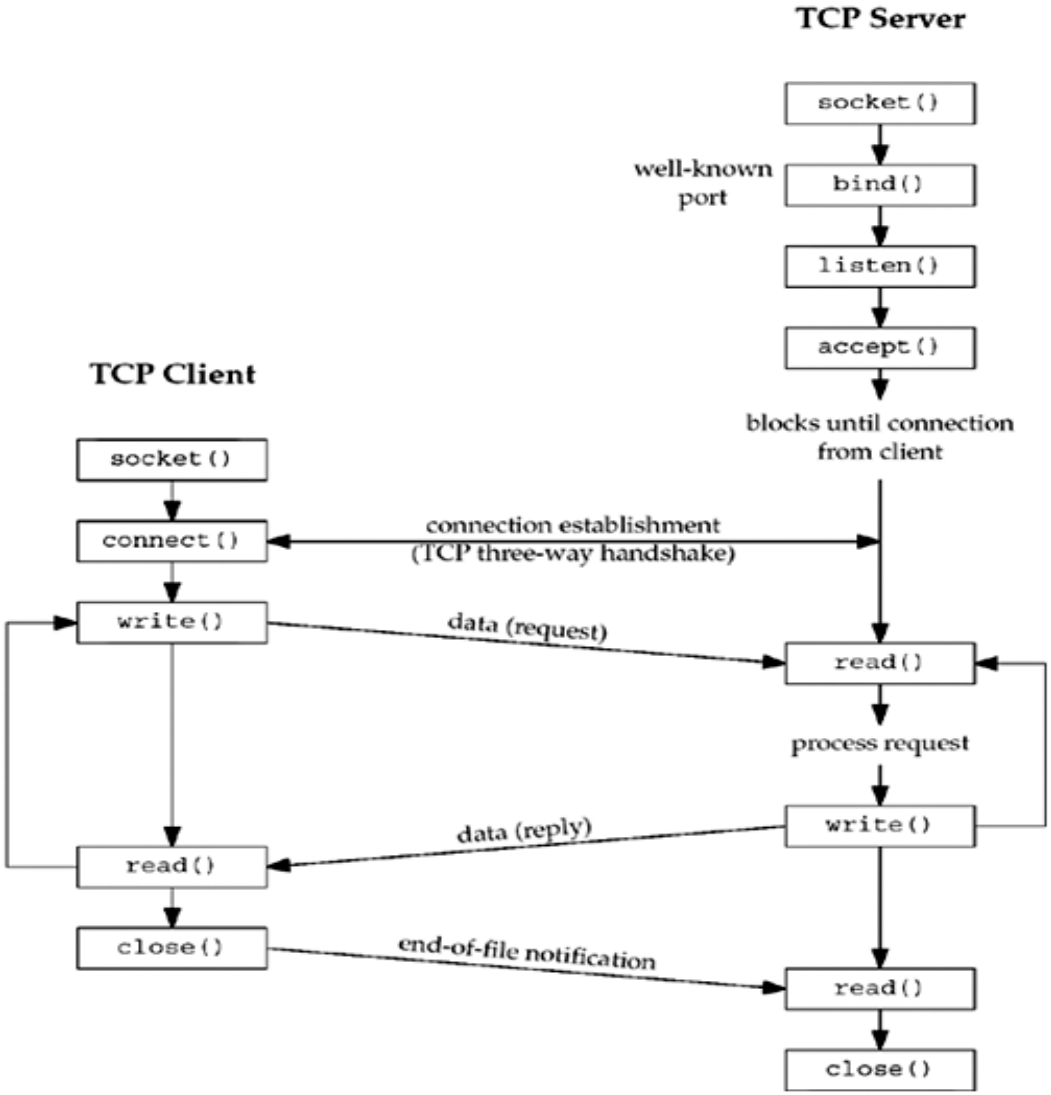


Figure: Socket functions for elementary TCP client/server

First, the server is started, then sometimes later, a client is started that connects to the server.

Assume that the client sends a request to the server, the server processes the request, and the server sends a reply back to the client.

Datatypes:: POSIX definition

Datatype	Description	Header
int8_t	Signed 8-bit integer	<sys/types.h>
uint8_t	Unsigned 8-bit integer	<sys/types.h>
int16_t	Signed 16-bit integer	<sys/types.h>
uint16_t	Unsigned 8-bit integer	<sys/types.h>
int32_t	Signed 32-bit integer	<sys/types.h>
uint32_t	Unigned 8-bit integer	<sys/types.h>
sa_family_t	Address family of socket address structure, normally any unsigned integer type (uint16_t)	<sys/socket.h>
socklen_t	Length of socket address structure, normally, uint32_t	<sys/socket.h>
in_addr_t	IPv4 address, normally, uint32_t	<netinet/in.h>
in_port_t	TCP or UDP port number, normally uint16_t	<netinet/in.h>

socket fields

IPV4 socket fields

length	family
port(16-bit)	address(32-bit)
char array unused field(8-bytes)	

socket fields

IPV4 socket fields

length	family
port(16-bit)	address(32-bit)
char array unused field(8-bytes)	

Generic socket fields

length	family
char array (14-bytes)	

socket fields

IPV4 socket fields

length	family
port(16-bit)	address(32-bit)
char array unused field(8-bytes)	

Generic socket fields

length	family
char array (14-bytes)	

Note: Any calls to the socket functions that pass a socket address structure from the process to the kernel (e.g. bind) must cast the pointer to the protocol-specific address structure to be a pointer to a generic address structures.

IPV4 Socket Address structure

1. An IPV4 socket address structure commonly called an **Internet socket structure**
2. It is named `sockaddr_in`.
3. It is defined by including the `<netinet/in.h>` header.

IPV4 socket address structure: `sockaddr_in`

```
struct sockaddr_in{
  uint8_t          sin_len;      /* length of structure(16) */
  sa_family_t      sin_family;   /* AF_INET */
  in_port_t        sin_port;     /* 16-bit TCP or UDP port number:
                                    network byte orderd */
  struct in_addr   sin_addr;     /* 32-bit IPv4 address */
  char             sin_zero[8];  /* unused */
};
```

IPV4 Socket Address structure

1. An IPV4 socket address structure commonly called an **Internet socket structure**
2. It is named `sockaddr_in`.
3. It is defined by including the `<netinet/in.h>` header.

IPV4 socket address structure: `sockaddr_in`

```
struct sockaddr_in{
  uint8_t          sin_len;      /* length of structure(16) */
  sa_family_t      sin_family;   /* AF_INET */
  in_port_t        sin_port;     /* 16-bit TCP or UDP port number:
                                    network byte orderd */
  struct in_addr   sin_addr;     /* 32-bit IPv4 address */
  char             sin_zero[8];  /* unused */
};
```

```
struct in_addr{
  in_addr_t    s_addr;          /* 32-bit IPv4 address:
                                   network byte ordered */
};
```

Generic Socket Address Structure

1. It is named **sockaddr**.
2. It is defined by including the <sys/socket.h> header.

```
struct sockaddr{
  uint8_t           sa_len;      /* length of structure(16)   */
  sa_family_t       sa_family;   /* Address family: AF_INET   */
  char              sa_data[14];  /* protocol specific address */
};
```

Mapping: IPV4 Socket Address Structure to Generic Socket Address Structure

	sa_len	sa_family	sa_data		
sockaddr	**Length**	**Family**	**Data(14 bytes)**		
	2 bytes	2 bytes	2 bytes	4 bytes	8 bytes
sockaddr_in	**Length**	**Family**	**Port**	**Address**	**Unused**

Note to Generic Socket Address Structure

Note to Generic Socket Address Structure

- A socket address structure (e.g. IPv4 socket address structure) is always passed by reference when passed as an argument to any socket functions.

Note to Generic Socket Address Structure

- A socket address structure (e.g. IPv4 socket address structure) is always passed by reference when passed as an argument to any socket functions.
- Any socket function (e.g. bind) that takes any one of these pointers as an argument must deal with socket address structures from any supported protocol families.

Note to Generic Socket Address Structure

- A socket address structure (e.g. IPv4 socket address structure) is always passed by reference when passed as an argument to any socket functions.
- Any socket function (e.g. bind) that takes any one of these pointers as an argument must deal with socket address structures from any supported protocol families.
- The pointer that is passed is declared to be the generic pointer type (`void *`). So, the pointer that is passed to be type casted to generic socket address structure.

Mapping: Example

```
struct sockaddr_in servaddr;
len=sizeof(struct sockaddr_in);
bind(sockfd, (struct sockaddr *)&servaddr, len);
```

Note

- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in **network byte order**.

Note

- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in **network byte order**.
- The 32-bit IPv4 address can be accessed in two different ways.

Note

- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in **network byte order**.
- The 32-bit IPv4 address can be accessed in two different ways.

For Example

```
struct sockaddr_in servaddr;
```

1. `servaddr.sin_addr` references the 32-bit IPv4 address as an `in_addr` structure.

Note

- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in **network byte order**.
- The 32-bit IPv4 address can be accessed in two different ways.

For Example

```
struct sockaddr_in servaddr;
```

1. `servaddr.sin_addr` references the 32-bit IPv4 address as an `in_addr` structure.
2. `servaddr.sin_addr.s_addr` references the same 32-bit IPv4 address as an `in_addr_t` (typically an unsigned 32-bit integer)

Note

- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in **network byte order**.
- The 32-bit IPv4 address can be accessed in two different ways.

For Example

```
struct sockaddr_in servaddr;
```

1. `servaddr.sin_addr` references the 32-bit IPv4 address as an `in_addr` structure.
2. `servaddr.sin_addr.s_addr` references the same 32-bit IPv4 address as an `in_addr_t` (typically an unsigned 32-bit integer)

- The `sin_zero` member is unused, but set it to zero when filling in one of these structure. **Although the most uses of the structure do not require that this member be 0, when binding a non wildcard IPv4 address, this member must be zero.**

Note

- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in **network byte order**.
- The 32-bit IPv4 address can be accessed in two different ways.

For Example

```
struct sockaddr_in servaddr;
```

1. `servaddr.sin_addr` references the 32-bit IPv4 address as an `in_addr` structure.
2. `servaddr.sin_addr.s_addr` references the same 32-bit IPv4 address as an `in_addr_t` (typically an unsigned 32-bit integer)

- The `sin_zero` member is unused, but set it to zero when filling in one of these structure. **Although the most uses of the structure do not require that this member be 0, when binding a non wildcard IPv4 address, this member must be zero.**
- **Socket address structures are used only a given host: The structure itself is not communicated between different hosts, although certain fileds (e.g. the IP address and port) are used for communication.**

IPV6 Socket Address Structure

1. It is named **sockaddr_in6**.
2. It is defined by including the <netinet/in.h> header.

IPV6 socket address structure: sockaddr_in6

```
struct sockaddr_in6{
  uint8_t           sin6_len;      /* length of this struct(28) */
  sa_family_t       sin6_family;    /* AF_INET6 */
  in_port_t         sin6_port;      /* transport layer port #:
                                       network byte orderd */
  uint32_t          sin6_flowinfo; /* flow information, undefined */
  struct in6_addr   sin6_addr;      /*  IPv6 address */
  uint32_t          sin6_scope_id; /* set for interfaces
                                          for a scope */
};
```

IPV6 Socket Address Structure

1. It is named **sockaddr_in6**.
2. It is defined by including the <netinet/in.h> header.

IPV6 socket address structure: sockaddr_in6

```
struct sockaddr_in6{
  uint8_t           sin6_len;      /* length of this struct(28) */
  sa_family_t       sin6_family;    /* AF_INET6 */
  in_port_t         sin6_port;      /* transport layer port #:
                                       network byte orderd */
  uint32_t          sin6_flowinfo; /* flow information, undefined */
  struct in6_addr   sin6_addr;      /*  IPv6 address */
  uint32_t          sin6_scope_id; /* set for interfaces
                                          for a scope */
};
```

```
struct in6_addr{
    uint8_t    s6_addr[16];           /* 128-bit IPv6 address:
                                         network byte ordered */
};
```

Note on IPV6 Socket Address Structure

- The **sin6_len** constant must be defined if the system supports the length member for socket address structure.
- The IPv6 family is **AF_INET6**, where as the IPv4 family is **AF_INET**
- The members in this structure are ordered so that if the **sockaddr_in6** structure is 64-bits aligned, so is the 128-bit **sin6_addr** member.
- The **sin6_flowinfo** member is divided into two fields:
  1. The low-order 20 bits are flow label
  2. The hig-order 12 bits are reserved
- The **sin6_scope_id** identifies the scope zone in which a scoped address is meaningful, most commonly an interface index for a link-local address.

New Generic Socket Address Structure as part of the IPv6 Socket API

1. It is named **`sockaddr_storage`**.
2. **`sockaddr_storage`** is large enough to hold any socket address type supported by the system.
3. It is defined by including the `<netinet/in.h>` header.

New Generic Socket Address Structure as part of the IPv6 Socket API

1. It is named **sockaddr_storage**.
2. **sockaddr_storage** is large enough to hold any socket address type supported by the system.
3. It is defined by including the <netinet/in.h> header.

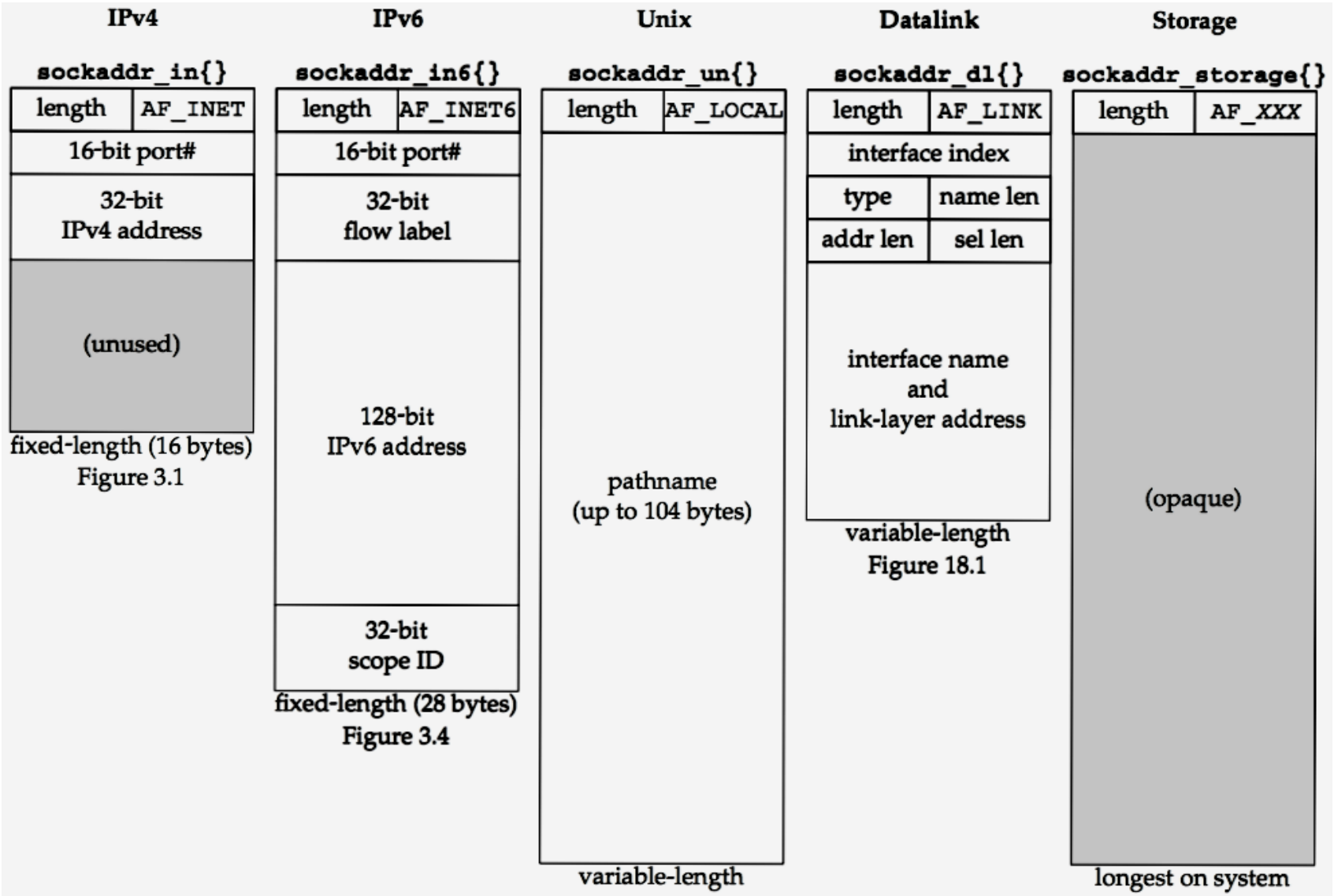
```
struct sockaddr_storage{
  uint8_t           ss_len;      /* length of this structure   */
  sa_family_t       ss_family;   /* Address family: AF_XXXX value */

  /* implementation-dependent elements to provide:

   * a) alignment sufficient to fulfil the alignment requirements
        of all socket address types that the system supports.

   * b) enough storage to hold any type of socket address that the
        system supports.
  */
};
```

The Five socket address structure

1. IPv4 socket
2. IPv6 socket
3. Unix domain socket(Text Book Figure 15.1)
4. Datalink socket(Text Book Figure 18.1)
5. Storage socket

Comparison of various Socket Address Structure

Length field, and family field assumed to be 1 byte



Server program: server.c

```c
/*Socket : Day Time Server*/
#include<stdio.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<stdlib.h>
#include<string.h>
#include<time.h>
int main(int argc, char **argv)
{
  int listenfd,connfd,len;
  struct sockaddr_in servaddr,clientaddr;
  char buff[1024];
  time_t ticks;
  len=sizeof(struct sockaddr_in);
  listenfd=socket(AF_INET,SOCK_STREAM,0);
  servaddr.sin_family=AF_INET;
  servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
  servaddr.sin_port=htons(0);
  bind(listenfd,(struct sockaddr *)&servaddr, sizeof(servaddr));
  getsockname(listenfd, (struct sockaddr *)&servaddr, &len);
  printf("After bind ephemeral port=%d\n",(int)ntohs(servaddr.sin_port));
  listen(listenfd,5);
  connfd=accept(listenfd, (struct sockaddr *)&clientaddr,&len);
  ticks=time(NULL);
  snprintf(buff,sizeof(buff),"%s\r\n",ctime(&ticks));
  write(connfd,buff,strlen(buff));
  write(connfd,"ITER",4);
  close(connfd);
}
```

Server program: client.c

```c
/*Socket : Day Time Client*/
#include<stdio.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<stdlib.h>
#include<string.h>
#include<time.h>
#include<arpa/inet.h>
int main(int argc, char *argv[])
{
 int sockfd,n,conn,len;
 int len;
 char recvline[1024];
 struct sockaddr_in servaddr;
 len=sizeof(struct sockaddr_in);
 sockfd=socket(AF_INET,SOCK_STREAM,0);
 servaddr.sin_family=AF_INET;
 servaddr.sin_addr.s_addr=inet_addr(argv[1]);//get ip from server
 servaddr.sin_port=htons(atoi(argv[2])); // Get the port from the server
 connect(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr));
 n=read(sockfd,recvline,1024);
 printf("%d\n",n);
 recvline[n]=0;
 printf("%s",recvline);
 close(sockfd);
}
```

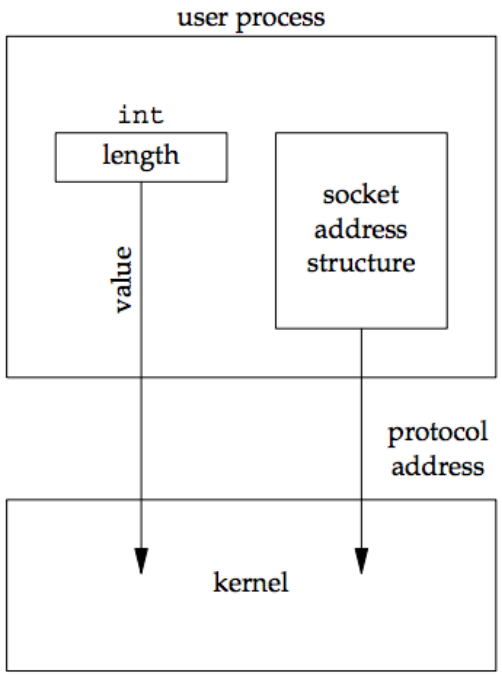
Value-Result Arguments-I

Socket address structure passed from **process** to **Kernel**

- `bind, connect, send, sendto, sendmsg` etc.

Example:

```
struct sockaddr_in servaddr;
len=sizeof(struct sockaddr_in);
bind(sockfd, (struct sockaddr *)&servaddr, len);
```



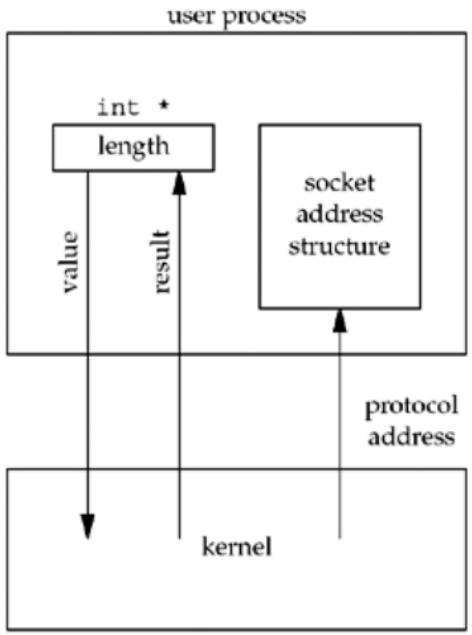
Value-Result Arguments-II

Socket address structure passed from **kernel** to **process**

- `accept`, `recvfrom`, `recvmsg`, `getpeername`, `getsockname` etc.

Example:

```
struct sockaddr_in servaddr;
len=sizeof(struct sockaddr_in);
accept(sockfd, (struct sockaddr *)&servaddr, &len);
```



Value-Result Arguments contd...

- In value-result arguments-II, pointer to the socket address structure along with pointer to an integer containing the size of the structure is passed from the kernel to the process.

Value-Result Arguments contd...

- In value-result arguments-II, pointer to the socket address structure along with pointer to an integer containing the size of the structure is passed from the kernel to the process.

- The reason that the **size** changes from an integer to be a pointer to an integer.

Value-Result Arguments contd...

- In value-result arguments-II, pointer to the socket address structure along with pointer to an integer containing the size of the structure is passed from the kernel to the process.
- The reason that the **size** changes from an integer to be a pointer to an integer.
- The **size** is both a ***value*** and a ***result***.

Value-Result Arguments contd...

- In value-result arguments-II, pointer to the socket address structure along with pointer to an integer containing the size of the structure is passed from the kernel to the process.
- The reason that the **size** changes from an integer to be a pointer to an integer.
- The **size** is both a ***value*** and a ***result***.
  - a ***value*** when the function(e.g. bind) is called. It tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in.

Value-Result Arguments contd...

- In value-result arguments-II, pointer to the socket address structure along with pointer to an integer containing the size of the structure is passed from the kernel to the process.

- The reason that the **size** changes from an integer to be a pointer to an integer.

- The **size** is both a ***value*** and a ***result***.

  - a ***value*** when the function(e.g. bind) is called. It tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in.

  - a ***result*** when the function(e.g. accept) returns. It tells the process how much information the kernel actually stored in the structure.

Value-Result Arguments contd...

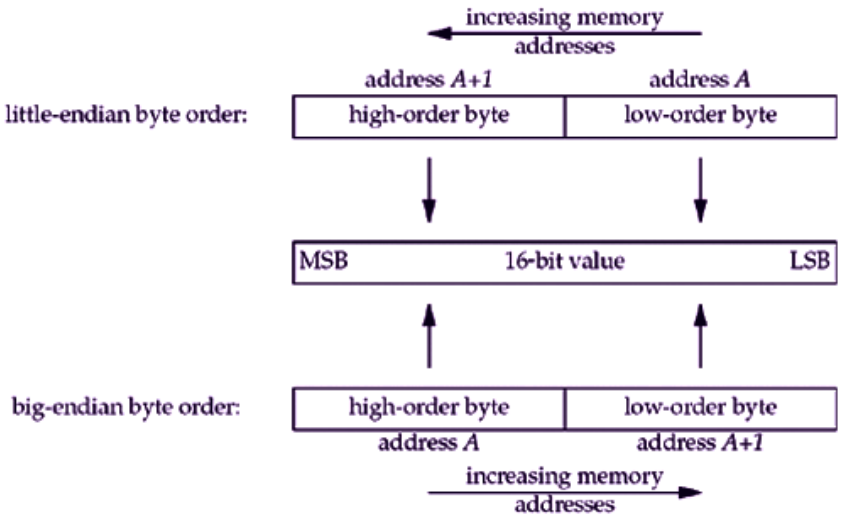
- In value-result arguments-II, pointer to the socket address structure along with pointer to an integer containing the size of the structure is passed from the kernel to the process.
- The reason that the **size** changes from an integer to be a pointer to an integer.
- The **size** is both a ***value*** and a ***result***.
  - a ***value*** when the function(e.g. bind) is called. It tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in.
  - a ***result*** when the function(e.g. accept) returns. It tells the process how much information the kernel actually stored in the structure.
  - This type of argument is called **value-result argument**.

Byte Ordering Functions

Two ways to store the bytes in memory:

1. **little-endian byte order:** with the lower-order byte at the starting address.
2. **big-endian byte order:** with the higher-order byte at the starting address.

Let us consider a 16-bit integer that is made up of 2 bytes.

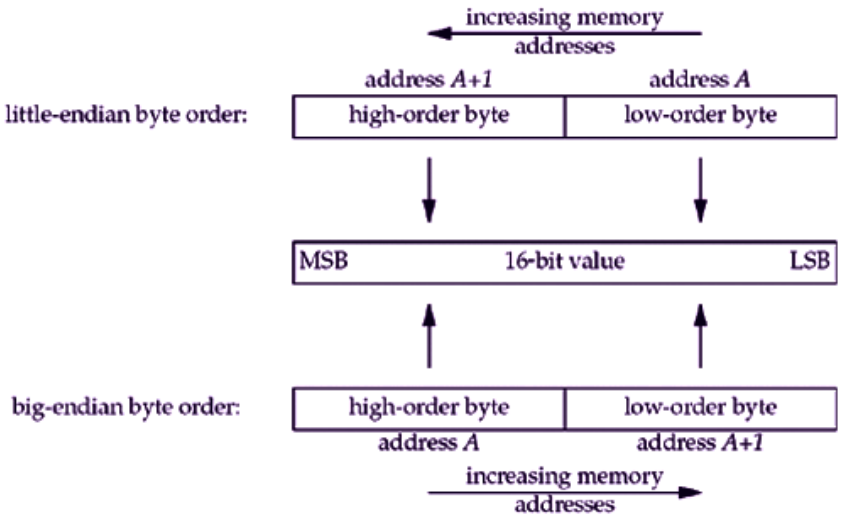


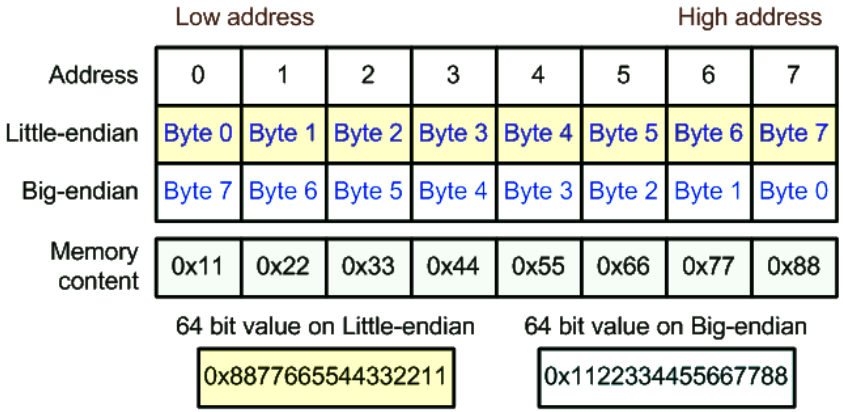
Byte Ordering Functions

Two ways to store the bytes in memory:

1. **little-endian byte order:** with the lower-order byte at the starting address.
2. **big-endian byte order:** with the higher-order byte at the starting address.

Let us consider a 16-bit integer that is made up of 2 bytes.



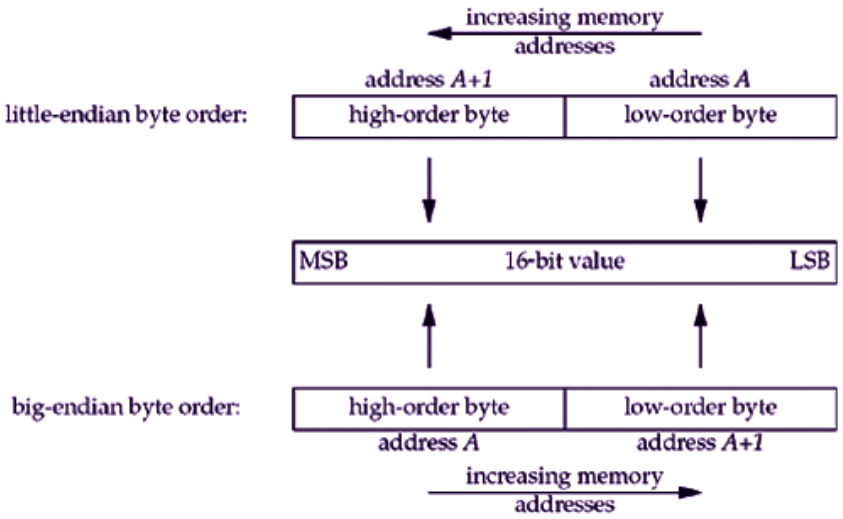


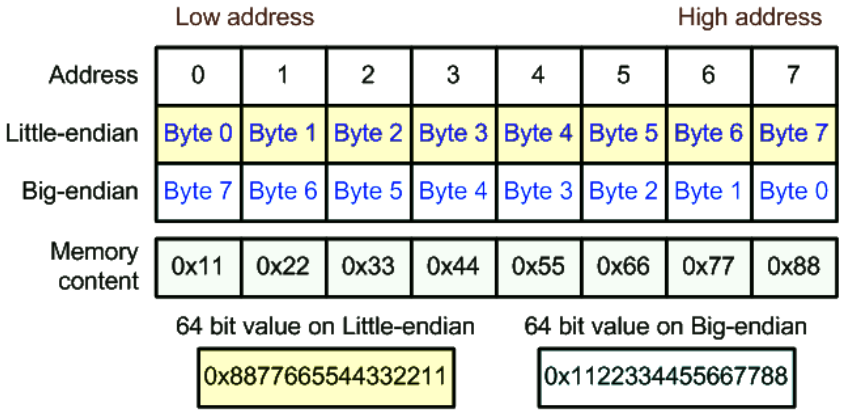
Byte Ordering Functions

Two ways to store the bytes in memory:

1. **little-endian byte order:** with the lower-order byte at the starting address.
2. **big-endian byte order:** with the higher-order byte at the starting address.

Let us consider a 16-bit integer that is made up of 2 bytes.





Note: There is no standard between these two byte ordering. So, systems use either formats.

Host Byte Order and Network Byte Order

Host byte order: byte ordering used by a given system.

Host Byte Order and Network Byte Order

Host byte order: byte ordering used by a given system.

Network byte order:

- Networking protocols define network byte order.
- Network byte order is big-endian byte order.
- The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of the multibytes fields will be transmitted.

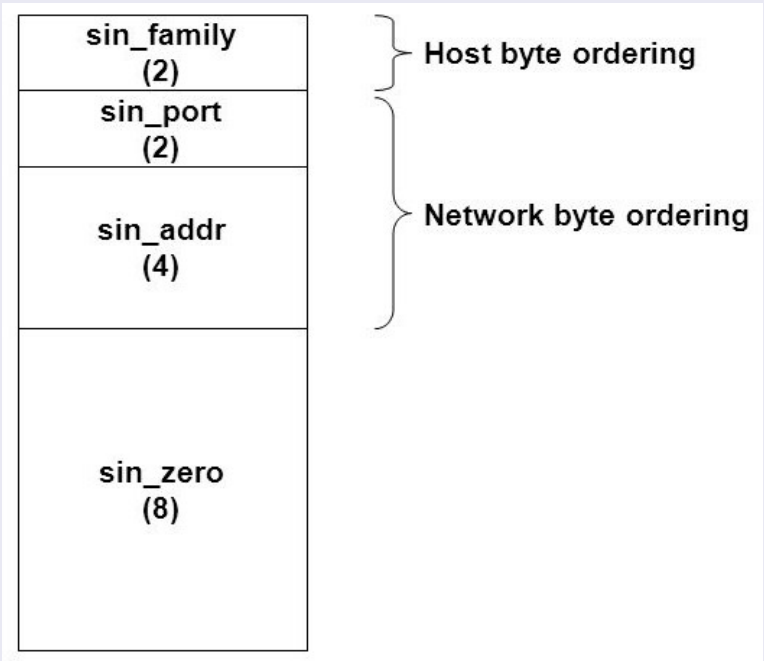
Host Byte Order and Network Byte Order

Host byte order: byte ordering used by a given system.

Network byte order:

- Networking protocols define network byte order.
- Network byte order is big-endian byte order.
- The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of the multibytes fields will be transmitted.

Byte ordering of `sockaddr_in` structure



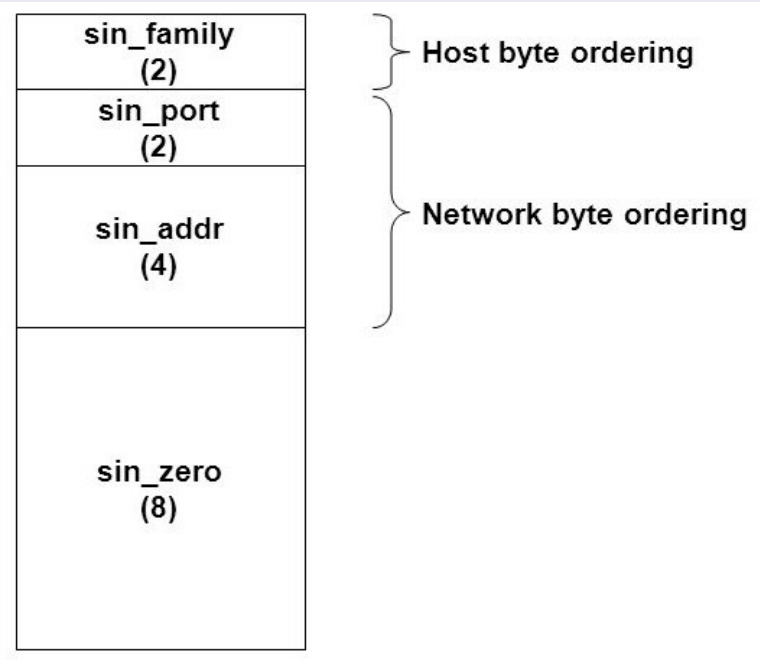
Host Byte Order and Network Byte Order

Host byte order: byte ordering used by a given system.

Network byte order:

- Networking protocols define network byte order.
- Network byte order is big-endian byte order.
- The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of the multibytes fields will be transmitted.

Byte ordering of `sockaddr_in` structure



So, appropriate functions must be called to convert between host and network byte order.

Conversion Functions

Converts between **host byte order** and **network byte order**

- `h` - host byte order
- `n` - network byte order
- `s` - short (2 bytes), converts port numbers
- `l` - long (4 bytes), converts IP addresses

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

uint16_t htons(unit16_t host16bitvalue);
uint32_t htonl(unit32_t host32bitvalue);

                    Both return: value in network byte order

uint16_t ntohs(unit16_t net16bitvalue);
uint32_t ntohl(unit32_t net32bitvalue);

                    Both return: value in host byte order
```

Byte Manipulation Functions

Two groups of functions:

- BSD provided: names begin with **b** (for byte)
- ANSI C provided: names begin with **m** (for memory)

As per 4.2BSD

```
#include<strings.h>

void bzero(void *dest, size_t nbytes);

void bcopy(const void *src, void *dest, size_t nbytes);

int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);

                    Returns: 0 if equal, nonzero if unequal
```

Note: Berkeley-derived Functions

- **`bzero`** sets the specified number of bytes to 0 in the destination.

Note: Berkeley-derived Functions

- **`bzero`** sets the specified number of bytes to 0 in the destination.
- **`bcopy`** moves the specified number of bytes from the source to the destination.

Note: Berkeley-derived Functions

- **`bzero`** sets the specified number of bytes to 0 in the destination.
- **`bcopy`** moves the specified number of bytes from the source to the destination.
- **`bcomp`** compares two arbitrary byte strings. The return value is zero if the two byte strings are identical; otherwise, it is nonzero.

Byte Manipulation Functions: ANSI C

- names begin with **m** (for memory) from 4.2BSD
- ANSI C provided:

```
#include<string.h>

void *memset(void *dest,int c, size_t len);

void *memcpy(void *dest, const void *src, size_t nbytes);

int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);

        Returns: 0 if equal, < 0 or > 0  if unequal
```

Note: ANSI C-derived Functions

- **`memset`** sets the specified number of bytes to the value **`c`** in the destination.

Note: ANSI C-derived Functions

- **`memset`** sets the specified number of bytes to the value **`c`** in the destination.

- **`memcpy`** moves the specified number of bytes from the source to the destination. **`memcpy`** is similar to **`bcopy`**, but the order of the two arguments is swapped. **`bcopy`** correctly handles overlapping fields, while the behavior of the **`memcpy`** is undefined if the source and destination overlap. The ANSCI C **`memmove`** must be used when the fields overlap.

Note: ANSI C-derived Functions

- **`memset`** sets the specified number of bytes to the value **`c`** in the destination.

- **`memcpy`** moves the specified number of bytes from the source to the destination. **`memcpy`** is similar to **`bcopy`**, but the order of the two arguments is swapped. **`bcopy`** correctly handles overlapping fields, while the behavior of the **`memcpy`** is undefined if the source and destination overlap. The ANSCI C **`memmove`** must be used when the fields overlap.

- **`memcmp`** compares two arbitrary byte strings and returns 0 if they are identical. If not identical, the return value is either greater than 0 or less than 0, depending on whether the first unequal byte pointed to by **`ptr1`** is greater than or less than the corresponding byte pointed to by **`ptr2`**. The comparison is done assuming the two unequal bytes are `unsigned chars`.

Address Conversion Function

Convert Internet addresses between ASCII string (what human prefer to use) and network byte ordered binary values (values that are stored in socket address structures).

1. **`inet_addr`**, and **`inet_aton`** functions convert an IPv4 addresses from a dotted-decimal string (e.g. `206.234.56.78`) to its 32-bit network byte ordered binary value. **`inet_ntoa`** function does the reverse.

2. functions for both IPv4 & IPv6 :**`inet_pton`**, **`inet_ntop`**

inet_aton, inet_addr, and inet_ntoa functions

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

int inet_aton(const char *strptr, struct in_addr *addptr);
                    Returns: 1 if string was valid, 0 on error

in_addr_t inet_addr(const char *strptr);
                  Returns: 32-bit binary network byte ordered
                           IPv4 address; INADDR_NONE if error

char *inet_ntoa(struct in_addr inaddr);
                     Returns: pointer to dotted-decimal string
```

inet_aton, inet_addr, and inet_ntoa functions

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

int inet_aton(const char *strptr, struct in_addr *addptr);
                    Returns: 1 if string was valid, 0 on error

in_addr_t inet_addr(const char *strptr);
                  Returns: 32-bit binary network byte ordered
                           IPv4 address; INADDR_NONE if error

char *inet_ntoa(struct in_addr inaddr);
                     Returns: pointer to dotted-decimal string
```

In inet_aton, and inet_ntoa:

inet_aton, inet_addr, and inet_ntoa functions

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

int inet_aton(const char *strptr, struct in_addr *addptr);
                    Returns: 1 if string was valid, 0 on error

in_addr_t inet_addr(const char *strptr);
                  Returns: 32-bit binary network byte ordered
                           IPv4 address; INADDR_NONE if error

char *inet_ntoa(struct in_addr inaddr);
                     Returns: pointer to dotted-decimal string
```

In **inet_aton**, and **inet_ntoa**:

- **a**- ASCII (ASCII string)

inet_aton, inet_addr, and inet_ntoa functions

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

int inet_aton(const char *strptr, struct in_addr *addptr);
                    Returns: 1 if string was valid, 0 on error

in_addr_t inet_addr(const char *strptr);
                  Returns: 32-bit binary network byte ordered
                           IPv4 address; INADDR_NONE if error

char *inet_ntoa(struct in_addr inaddr);
                     Returns: pointer to dotted-decimal string
```

In **inet_aton**, and **inet_ntoa**:

- **a**- ASCII (ASCII string)
- **n** - Network(network byte ordered binary values)

`inet_aton`, `inet_addr`, and `inet_ntoa` Descriptions

`inet_aton`: coverts C charcter string pointed to by **`strptr`** into its 32-bit binary betwork byte ordered value, which is strored through the pointer **`addptr`**. If successful, 1 is returned; otherwise, 0 is returned.

```
struct sockaddr_in servaddr;
inet_aton(argv[1], &servaddr.sin_addr);
```

`inet_aton`, `inet_addr`, and `inet_ntoa` Descriptions

`inet_aton`: coverts C charcter string pointed to by **`strptr`** into its 32-bit binary betwork byte ordered value, which is strored through the pointer **`addptr`**. If successful, 1 is returned; otherwise, 0 is returned.

```
struct sockaddr_in servaddr;
inet_aton(argv[1], &servaddr.sin_addr);
```

`inet_addr`: same conversion as like **`inet_aton`**, returning the 32-bit binary network byte ordered value. The function returns the constant **`INADDR_NONE`** (typically 32 one-bits) on an error.

```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr=inet_addr(argv[1]);
```

`inet_aton`, `inet_addr`, and `inet_ntoa` Descriptions

`inet_aton`: coverts C charcter string pointed to by **`strptr`** into its 32-bit binary betwork byte ordered value, which is strored through the pointer **`addptr`**. If successful, 1 is returned; otherwise, 0 is returned.

```
struct sockaddr_in servaddr;
inet_aton(argv[1], &servaddr.sin_addr);
```

`inet_addr`: same conversion as like **`inet_aton`**, returning the 32-bit binary network byte ordered value. The function returns the constant **`INADDR_NONE`** (typically 32 one-bits) on an error.

```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr=inet_addr(argv[1]);
```

`inet_ntoa`: converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string.

```
struct sockaddr_in servaddr;
printf("IP::%s\n", inet_ntoa(servaddr.sin_addr));
```

inet_pton, and inet_ntop functions

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

int inet_pton(int family, const char *strptr, void *addptr);
          Returns: 1 if OK,
                   0 if input not a valid presentation format,
                  -1 on error

const char *inet_ntop(int family, const void *addrptr,
                                  char *strptr, size_t len);

                            Returns: pointer to result if OK,
                                     NULL on error
```

inet_pton, and inet_ntop functions

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

int inet_pton(int family, const char *strptr, void *addptr);
          Returns: 1 if OK,
                   0 if input not a valid presentation format,
                  -1 on error

const char *inet_ntop(int family, const void *addrptr,
                                  char *strptr, size_t len);

                            Returns: pointer to result if OK,
                                     NULL on error
```

inet_pton, and inet_ntop functions

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

int inet_pton(int family, const char *strptr, void *addptr);
          Returns: 1 if OK,
                   0 if input not a valid presentation format,
                  -1 on error

const char *inet_ntop(int family, const void *addrptr,
                                  char *strptr, size_t len);

                              Returns: pointer to result if OK,
                                       NULL on error
```

In inet_pton, and inet_ntop:

- p - Presentation (ASCII string)

inet_pton, and inet_ntop functions

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

int inet_pton(int family, const char *strptr, void *addptr);
          Returns: 1 if OK,
                   0 if input not a valid presentation format,
                  -1 on error

const char *inet_ntop(int family, const void *addrptr,
                                  char *strptr, size_t len);

                              Returns: pointer to result if OK,
                                       NULL on error
```

In inet_pton, and inet_ntop:

- p - Presentation (ASCII string)
- n - Numeric (network byte ordered binary values)

inet_pton, and inet_ntop functions

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

int inet_pton(int family, const char *strptr, void *addptr);
          Returns: 1 if OK,
                   0 if input not a valid presentation format,
                  -1 on error

const char *inet_ntop(int family, const void *addrptr,
                                  char *strptr, size_t len);

                              Returns: pointer to result if OK,
                                       NULL on error
```

In **inet_pton**, and **inet_ntop**:

p - Presentation (ASCII string)

n - Numeric (network byte ordered binary values)

family - `AF_INET` or `AF_INET6`. If **family** is not supported, both functions return an error with `errno` set to `EAFNOSUPPORT`.

`inet_pton`, and `inet_ntop` Descriptions

`inet_pton`: convert the string pointed to by **`strptr`**, storing the binary result through the pointer **`addrptr`**. If successful, the return value is 1. If the input string is not a valid presentation format for the specified family, 0 is returned.

```
struct sockaddr_in servaddr;
inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
```

`inet_pton`, and `inet_ntop` Descriptions

`inet_pton`: convert the string pointed to by **`strptr`**, storing the binary result through the pointer **`addrptr`**. If successful, the return value is 1. If the input string is not a valid presentation format for the specified family, 0 is returned.

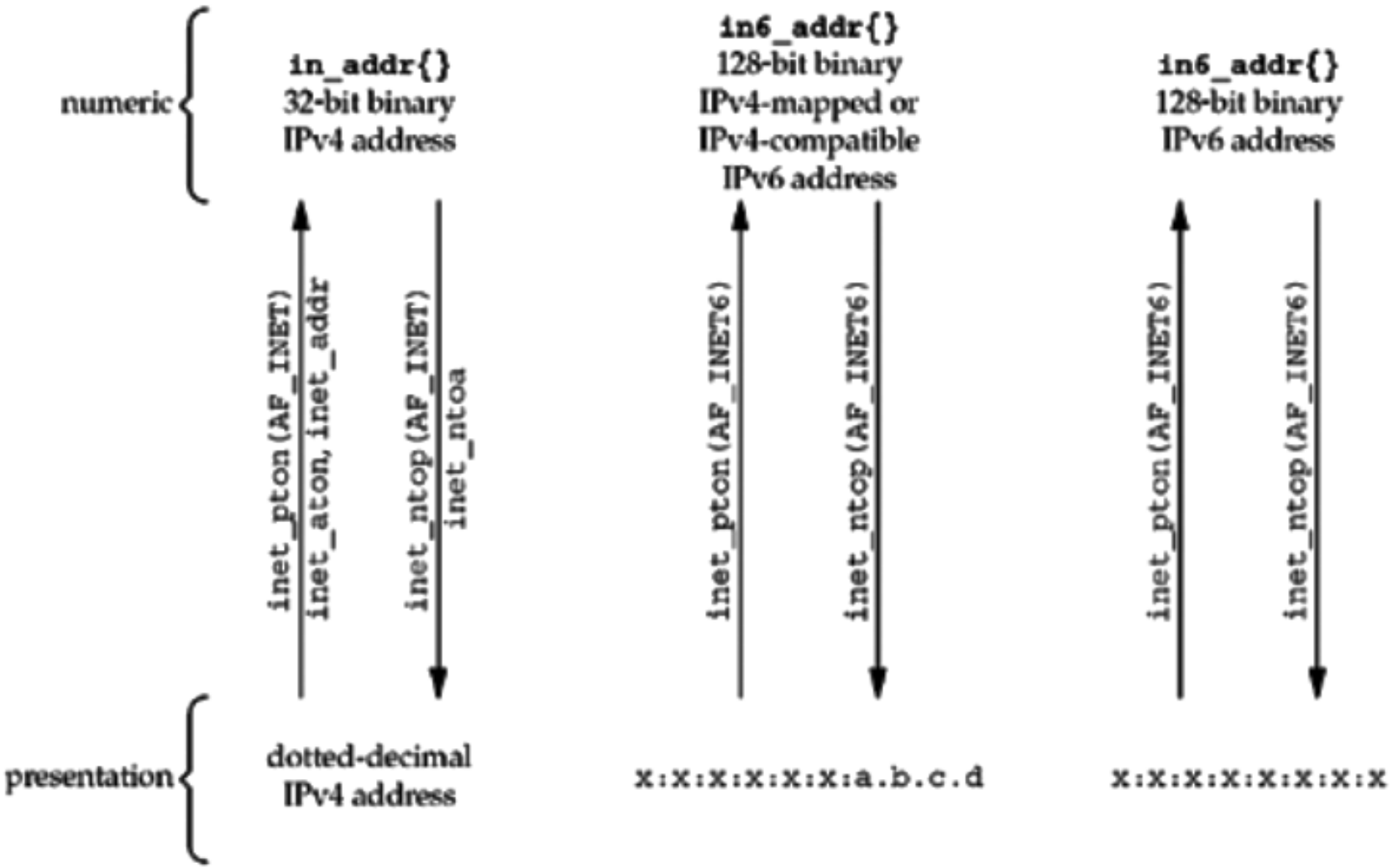
```
struct sockaddr_in servaddr;
inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
```

`inet_ntop`: does the reverse conversion, from numeric (**`addrptr`**) to presentation (**`strptr`**).

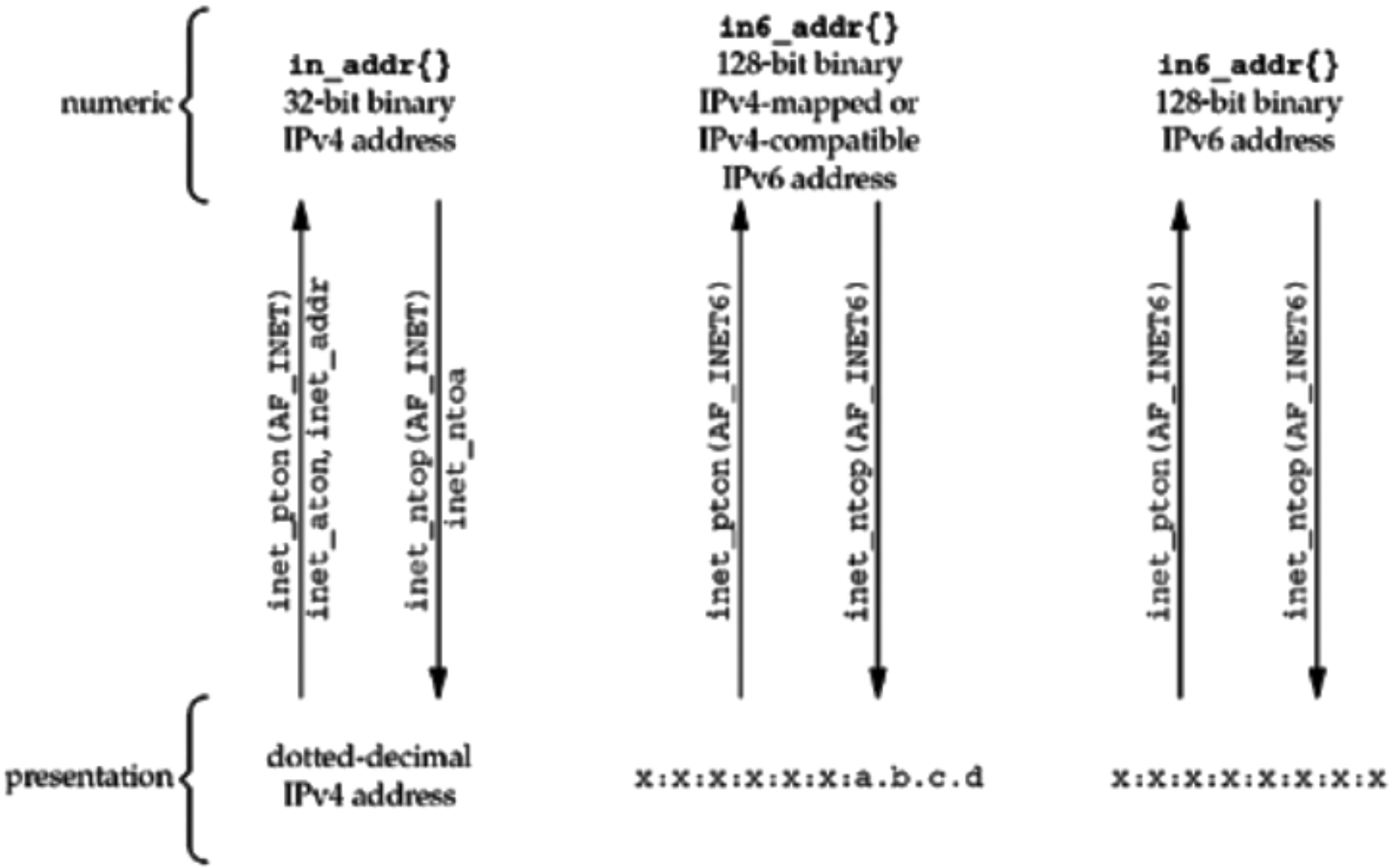
```
#define INET_ADDRSTRLEN 16
struct sockaddr_in servaddr;
char strptr[INET_ADDRSTRLEN];
inet_ntop(AF_INET, &servaddr.sin_addr, strptr,
                                   INET_ADDRSTRLEN);
printf("IP::%s\n", strptr);
```

Note: If `len` is too small to hold the resulting presentation format, including the terminating null, a null pointer ie returned and `errno` is set to `ENOSPC`.

Summary of Address Conversion Functions



Summary of Address Conversion Functions



Elementary **Socket** Functions

(a) To perform network I/O, a process must call the *socket* function, specifying the type of communication protocol desired.

(b) **Communication protocol:** TCP using IPV4, UDP using IPV4, Unix domain stram protocol, etc.

(c) **Baisc functions:**

Elementary **Socket** Functions

(a) To perform network I/O, a process must call the *socket* function, specifying the type of communication protocol desired.

(b) **Communication protocol:** TCP using IPV4, UDP using IPV4, Unix domain stram protocol, etc.

(c) **Baisc functions:**

1. `socket`
2. `connect`
3. `bind`
4. `listen`
5. `accept`
6. `close`
7. `getsockname` and `getpeername`
8. socket I/O functions: `send, recv, read, and write` etc.
9. ⋮

`Socket` Function

`socket()` - creates an endpoint for communication and returns a descriptor.

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

int socket(int family, int type, int protocol);

                    Returns: non-negative descriptor if OK,
                             -1 on error
```

Socket Function

socket() - creates an endpoint for communication and returns a descriptor.

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

int socket(int family, int type, int protocol);

                        Returns: non-negative descriptor if OK,
                                 -1 on error
```

family:

family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing sockets
AF_KEY	key socket

`Socket` Function

`socket()` - creates an endpoint for communication and returns a descriptor.

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

int socket(int family, int type, int protocol);

                    Returns: non-negative descriptor if OK,
                             -1 on error
```

`family:`

family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing sockets
AF_KEY	key socket

`type:`

type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

Protocol **`family`** constants and **`type`** of socket for **`socket`** function

Protocol of Sockets for `AF_INET` or `AF_INET6`

The **`protocol`** argument to the socket function should bet set to the specific protocol type given in the below table or 0 to select the system's default for the given combination of **`family`** and **`type`**.

Protocol of Sockets for `AF_INET` or `AF_INET6`

The **`protocol`** argument to the socket function should bet set to the specific protocol type given in the below table or 0 to select the system's default for the given combination of **`family`** and **`type`**.

`protocol`	**Description**
`IPPROTO_TCP`	TCP transport protocol
`IPPROTO_UDP`	UDP transport protocol
`IPPROTO_SCTP`	SCTP transport protocol

Table: Protocol of sockets

Protocol of Sockets for AF_INET or AF_INET6

The **protocol** argument to the socket function should bet set to the specific protocol type given in the below table or 0 to select the system's default for the given combination of **family** and **type**.

protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

Table: Protocol of sockets

Example

```
#include<sys/socket.h>
int sockfd;
sockfd=socket(AF_INET, SOCK_STREAM, 0);
```

ON success, the **socket** function returns a small non-negative integer value stored on **sockfd**.

Combinations of `family` and `type` for the `socket` Function

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP | SCTP	TCP | SCTP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_SEQPACKET	SCTP	SCTP	Yes		
SOCK_RAW	IPv4	IPV6		Yes	Yes

Combinations of `family` and `type` for the `socket` Function

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP | SCTP	TCP | SCTP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_SEQPACKET	SCTP	SCTP	Yes		
SOCK_RAW	IPv4	IPV6		Yes	Yes

Note:

- Not all combinations of socket `family` and `type` are valid.
- Table above shows the valid combinations along with the actual protocols that are valid.
- The boxes marked “Yes” are valid but do not have handy acronums.
- The blank boxes are not supported.
- Ref. Text Book for **`AF_xxx`** versus **`PF_xxx`**

`connect` Function

`socket()` - initiate a connection on a socket.

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

int connect(int sockfd, const struct sockaddr *servaddr,
                                        socklen_t addrlen);

                    Returns:   0 if OK,
                              -1 on error
```

`connect` Function

`socket()` - initiate a connection on a socket.

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

int connect(int sockfd, const struct sockaddr *servaddr,
                                        socklen_t addrlen);

                    Returns:   0 if OK,
                              -1 on error
```

- **`sockfd`** is a socket descriptor returned by the **`socket`** function.
- The second argument is a pointer to generic socket address structure.
- Third argument is the size of socket address structure.

connect Function

socket() - initiate a connection on a socket.

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

int connect(int sockfd, const struct sockaddr *servaddr,
                                        socklen_t addrlen);

                    Returns:   0 if OK,
                              -1 on error
```

- **sockfd** is a socket descriptor returned by the **socket** function.
- The second argument is a pointer to generic socket address structure.
- Third argument is the size of socket address structure.

Example:

```
struct sockaddr_in servaddr;
socklen_t addrlen;
addrlen=sizeof(sockaddr_in);
/* fill in servaddr{} :: IP address and port number*/
connect(int sockfd, (struct sockaddr *)&servaddr, addrlen);
```

Note:: In case of TCP socket

The **`connect`** function initiates TCP's three-way handshake. The function returns only when the connection is established or *error* occurs. There are several different *error* returns possible.

Note:: In case of TCP socket

The **`connect`** function initiates TCP's three-way handshake. The function returns only when the connection is established or *error* occurs. There are several different *error* returns possible.

`ETIMEDOUT`: If the client TCP receives no response to its **SYN** segment.

`ECONNREFUSED`: If the server's response to the client's **SYN** ia s reset (**RST**), this indicates that no process is waiting for connections on the server host at the port specified(i.e., the server process is probably not running). This kind of error is a *hard error*.

`EHOSTUNREACH` or

`ENETUNREACH`: If the client's **SYN** elicits an ICMP "destination unreachable" from some intermediate router. It is considered a *soft error*.

`bind` Function

- assigns a local protocol address to a socket.
- with the Internet protocols, the **protocol address**={a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number}.
- server calls `bind`.

`bind` Function

- assigns a local protocol address to a socket.
- with the Internet protocols, the **protocol address**={a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number}.
- server calls `bind`. The client does not have to call `bind`?.

```
#include<sys/socket.h>
int bind(int sockfd, const struct sockaddr *servaddr,
                                        socklen_t addrlen);

                    Returns:   0 if OK,
                              -1 on error
```

`bind` Function

- assigns a local protocol address to a socket.
- with the Internet protocols, the **protocol address**={a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number}.
- server calls `bind`. The client does not have to call `bind`?.

```
#include<sys/socket.h>
int bind(int sockfd, const struct sockaddr *servaddr,
                                        socklen_t addrlen);

                    Returns:   0 if OK,
                              -1 on error
```

- **`sockfd`** is a socket descriptor returned by the **`socket`** function.
- The second argument is a pointer to generic socket address structure.
- Third argument is the size of socket address structure.

Example: `bind` Function

```
struct sockaddr_in servaddr;
socklen_t addrlen;
addrlen=sizeof(sockaddr_in);
/* fill in servaddr{} :: IP address and port number*/
bind(sockfd, (struct sockaddr *)&servaddr, addrlen);
```

Result: Specifying IP address and/or Port Number to `bind`

Process specifies		Result
IP address	**Port**	
Wildcard	0	Kernel chooses IP address, and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Loacl IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

Result: Specifying IP address and/or Port Number to `bind`

Process specifies		Result
IP address	**Port**	
Wildcard	0	Kernel chooses IP address, and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Loacl IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

with IPV4: The *wildcard* address is specified by the constant `INADDR_ANY`, whose value is 0.

```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
```

with IPv6: different than IPV4.

```
struct sockaddr_in6 serv;
serv.sin6_addr=in6addr_any; /*wildcard*/
```

`listen` Function

`listen()` - called only by a TCP server. Listen for connections on a socket.

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

int listen(int sockfd, int backlog);

                    Returns:  0 if OK,
                             -1 on error
```

`listen` Function

`listen()` - called only by a TCP server. Listen for connections on a socket.

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

int listen(int sockfd, int backlog);

                    Returns:  0 if OK,
                             -1 on error
```

- **`sockfd`** is a socket descriptor returned by the **`socket`** function.
- The second argument is an integer value specifies the maximum number of connections the kernel should queue for this socket.

`listen` Function

`listen()` - called only by a TCP server. Listen for connections on a socket.

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

int listen(int sockfd, int backlog);

                    Returns:  0 if OK,
                             -1 on error
```

- **`sockfd`** is a socket descriptor returned by the **`socket`** function.
- The second argument is an integer value specifies the maximum number of connections the kernel should queue for this socket.

Example:

```
int sockfd;
listen(sockfd, 5);
```

Note::: `listen` Function

The **`listen`** function is called only by a TCP server and it performs two actions:

Note::: `listen` Function

The **`listen`** function is called only by a TCP server and it performs two actions:

- When a socket is created by the `socket` function, it is assumed to be an active socket(e.g. a client socket that will issue a `connect`). The `listen` function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket(socket—-listening socket).

Note::: `listen` Function

The **`listen`** function is called only by a TCP server and it performs two actions:

- When a socket is created by the `socket` function, it is assumed to be an active socket(e.g. a client socket that will issue a `connect`). The `listen` function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket(socket—-listening socket).
- In terms of the TCP state transition diagram, the call to `listen` moves the socket from the CLOSED state to the LISTEN state.

Note::: `listen` Function

The **`listen`** function is called only by a TCP server and it performs two actions:

- When a socket is created by the `socket` function, it is assumed to be an active socket(e.g. a client socket that will issue a `connect`). The `listen` function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket(socket—-listening socket).
- In terms of the TCP state transition diagram, the call to `listen` moves the socket from the CLOSED state to the LISTEN state.
- The second argument specifies the maximum number of connections the kernel should queue for this socket.

More on `backlog` (The second argument to `listen` function)

Kernel maintains two queues for a given listening socket

More on `backlog` (The second argument to `listen` function)

Kernel maintains two queues for a given listening socket

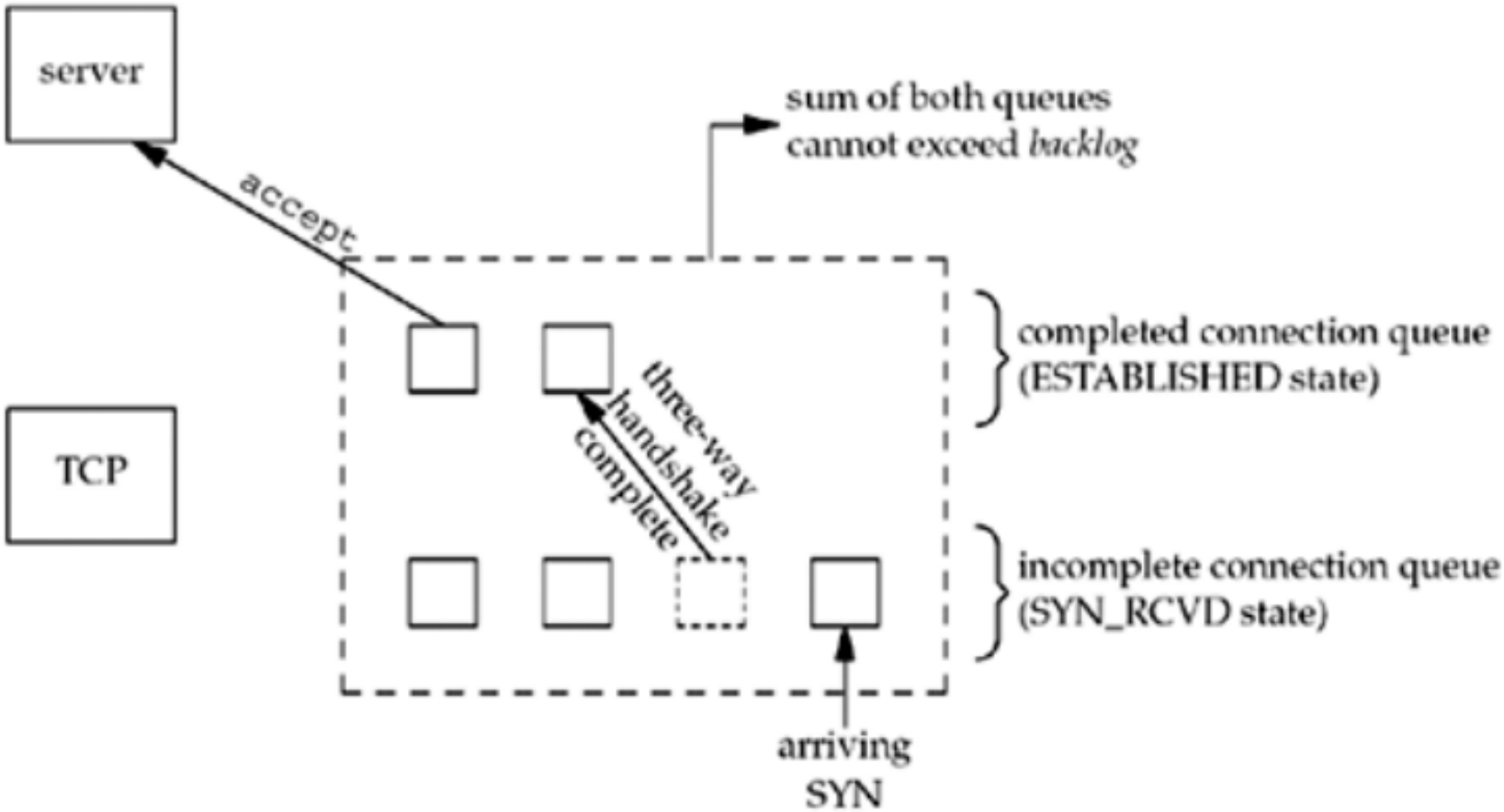
- An incomplete connection queue: which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way hanshake. These sockets are in the SYN_rcvd state.

More on `backlog` (The second argument to `listen` function)

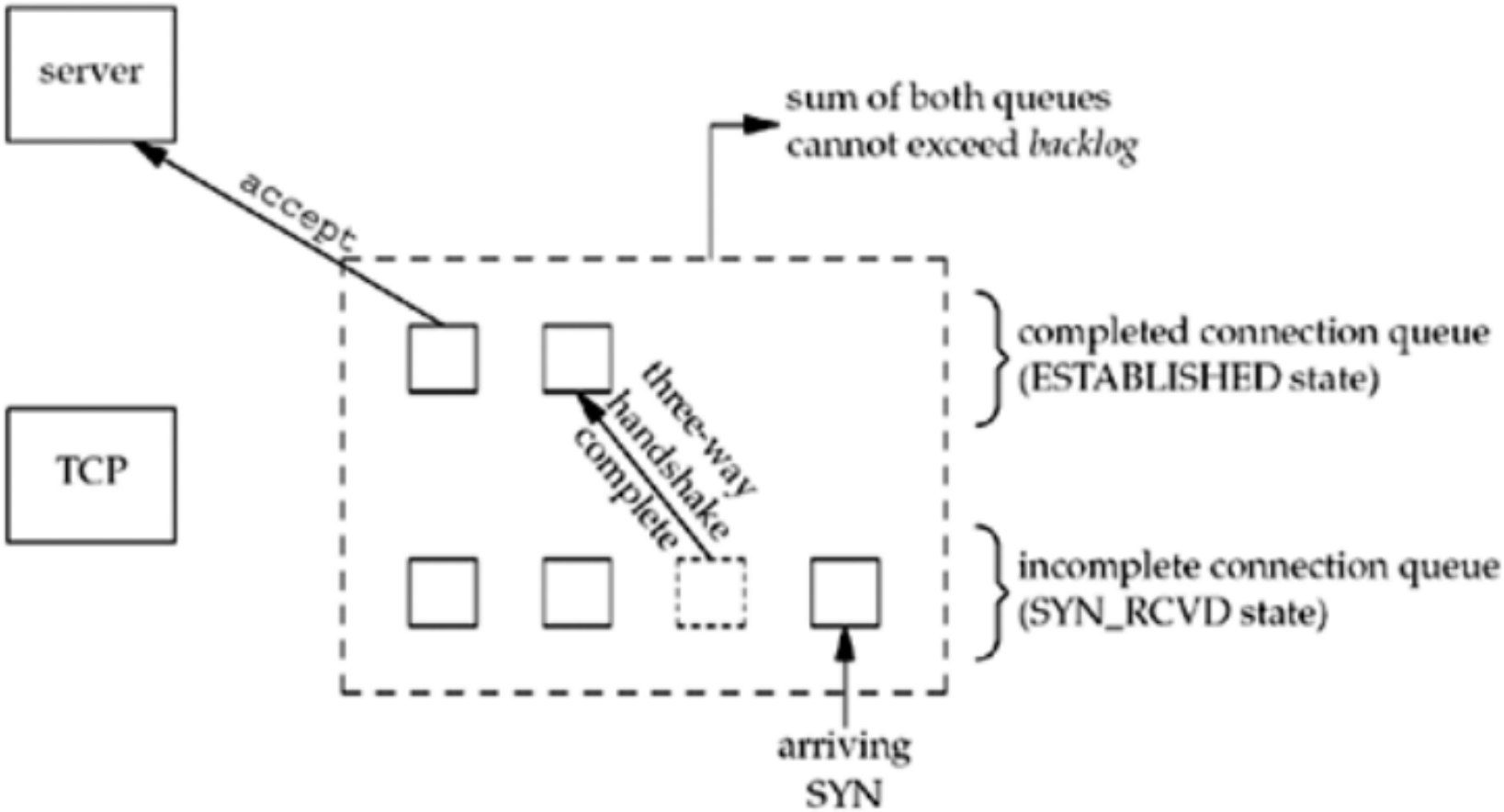
Kernel maintains two queues for a given listening socket

- An incomplete connection queue: which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way hanshake. These sockets are in the SYN_rcvd state.
- A completed connection queue: which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state.

Two queues maintained by TCP for listening socket



Two queues maintained by TCP for listening socket



`accept` Function

`accept()` - called by a TCP server. Accept a connection on a socket.

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

int accept(int sockfd, const struct sockaddr *cliaddr,
                                        socklen_t *addrlen);

                    Returns:  non-negative descriptor if OK,
                                    -1 on error
```

`accept` Function

`accept()` - called by a TCP server. Accept a connection on a socket.

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

int accept(int sockfd, const struct sockaddr *cliaddr,
                                        socklen_t *addrlen);

                    Returns:  non-negative descriptor if OK,
                                  -1 on error
```

- **`sockfd`** is a socket descriptor returned by the **`socket`** function.
- The second argument is a pointer to generic socket address structure.
- Third argument is a pointer the size of socket address structure.

accept Function

accept() - called by a TCP server. Accept a connection on a socket.

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

int accept(int sockfd, const struct sockaddr *cliaddr,
                                        socklen_t *addrlen);

                    Returns:  non-negative descriptor if OK,
                                    -1 on error
```

- **sockfd** is a socket descriptor returned by the **socket** function.
- The second argument is a pointer to generic socket address structure.
- Third argument is a pointer the size of socket address structure.

Example:

```
struct sockaddr_in cliaddr;
socklen_t addrlen; int fd;
addrlen=sizeof(sockaddr_in);
/* fill in servaddr{} :: IP address and port number*/
fd=accept(int sockfd, (struct sockaddr *)&cliaddr, &addrlen);
```

`close` Function

`close()` - used to close a socket and terminate a TCP connection.

```
#include<unistd.h>

int close(int sockfd);

                    Returns:   0 if OK,
                              -1 on error
```

`close` Function

`close()` - used to close a socket and terminate a TCP connection.

```
#include<unistd.h>

int close(int sockfd);

                    Returns:   0 if OK,
                              -1 on error
```

- **`sockfd`** is a socket descriptor returned by the **`socket`** function.

`close` Function

`close()` - used to close a socket and terminate a TCP connection.

```
#include<unistd.h>

int close(int sockfd);

                    Returns:   0 if OK,
                              -1 on error
```

- **`sockfd`** is a socket descriptor returned by the **`socket`** function.

Example:

```
int sockfd;
close(sockfd);
```

Kernel to choose a port number for our socket

getsockname()

1. To obtain the value of the ephemeral port assigned by the kernal, call *getsockname* to return the prorocol address.
2. protocol address is the combination of ip address along with a port number. Also called socket address.
3. ***getsockname()* signature :**

```
struct sockaddr_in localaddr;
int getsockname(int sockfd, struct sockaddr *localaddr,
                                        socklen_t  *addrlen);
```

getsockname()

Description

1. **getsockname()** returns the current address to which the socket sockfd is bound, in the buffer pointed to by **addr**.

getsockname()

Description

1. **getsockname()** returns the current address to which the socket sockfd is bound, in the buffer pointed to by **addr**.
2. The addrlen argument should be initialized to indicate the amount of space (in bytes) pointed to by **addr**.

getsockname()

Description

1. **getsockname()** returns the current address to which the socket sockfd is bound, in the buffer pointed to by **addr**.
2. The addrlen argument should be initialized to indicate the amount of space (in bytes) pointed to by **addr**.
3. On return it contains the actual size of the socket address.

getsockname()

Description

1. **getsockname()** returns the current address to which the socket sockfd is bound, in the buffer pointed to by **addr**.
2. The addrlen argument should be initialized to indicate the amount of space (in bytes) pointed to by **addr**.
3. On return it contains the actual size of the socket address.

Return value

1. On success, zero is returned.

getsockname()

Description

1. **getsockname()** returns the current address to which the socket sockfd is bound, in the buffer pointed to by **addr**.
2. The addrlen argument should be initialized to indicate the amount of space (in bytes) pointed to by **addr**.
3. On return it contains the actual size of the socket address.

Return value

1. On success, zero is returned.
2. On error, -1 is returned, and errno is set appropriately.

getpeername Function

getpeername() - Used to get the foreign protocol address associated with a socket.

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

int getpeername(int sockfd, const struct sockaddr *peeraddr,
                                        socklen_t *addrlen);

                    Returns:   0 if OK,
                              -1 on error
```

getpeername Function

getpeername() - Used to get the foreign protocol address associated with a socket.

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

int getpeername(int sockfd, const struct sockaddr *peeraddr,
                                        socklen_t *addrlen);

                    Returns:   0 if OK,
                              -1 on error
```

- **sockfd** is a socket descriptor returned by the **socket** function.
- The second argument is a pointer to generic socket address structure.
- Third argument is a pointer the size of socket address structure.

getpeername Function

getpeername() - Used to get the foreign protocol address associated with a socket.

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

int getpeername(int sockfd, const struct sockaddr *peeraddr,
                                        socklen_t *addrlen);

                    Returns:   0 if OK,
                              -1 on error
```

- **sockfd** is a socket descriptor returned by the **socket** function.
- The second argument is a pointer to generic socket address structure.
- Third argument is a pointer the size of socket address structure.

Example:

```
struct sockaddr_in peeraddr;
socklen_t addrlen;
addrlen=sizeof(sockaddr_in);
/* fill in servaddr{} :: IP address and port number*/
getpeername(int sockfd, (struct sockaddr *)&peeraddr, &addrlen);
```

Just a Begin....