

Socket Programming

Kameswari Chebrolu

Reference: Beej's Guide to Network Programming
this basically is related to the application layer.. how to generate packets etc etc.



Quote

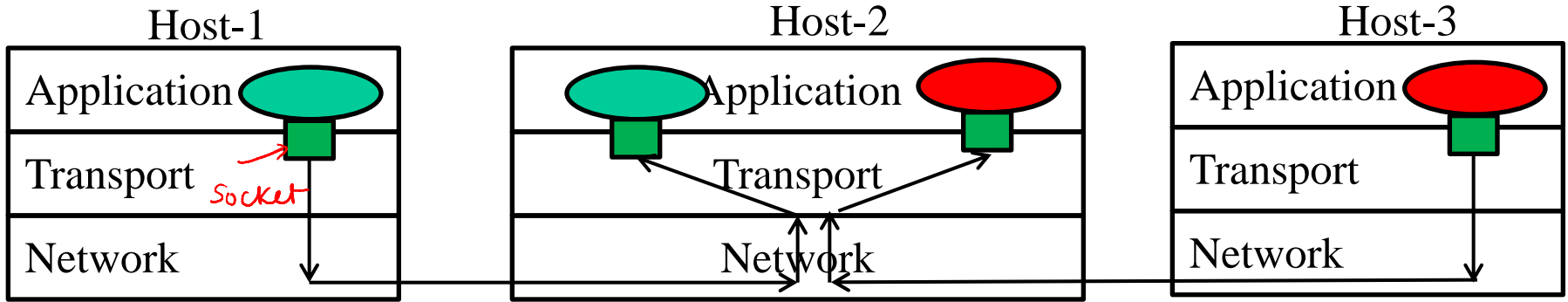
I hear and I forget

I see and I remember

I do and I understand

-- Chinese Proverb

Multiplexing/Demultiplexing



applications interact with Transport layer using Sockets... the important things here are src, dst ports, ip address

Demultiplexing: Deliver segments to the right socket

Multiplexing: Assemble segments such that they get delivered to right socket

src & Dest
IP

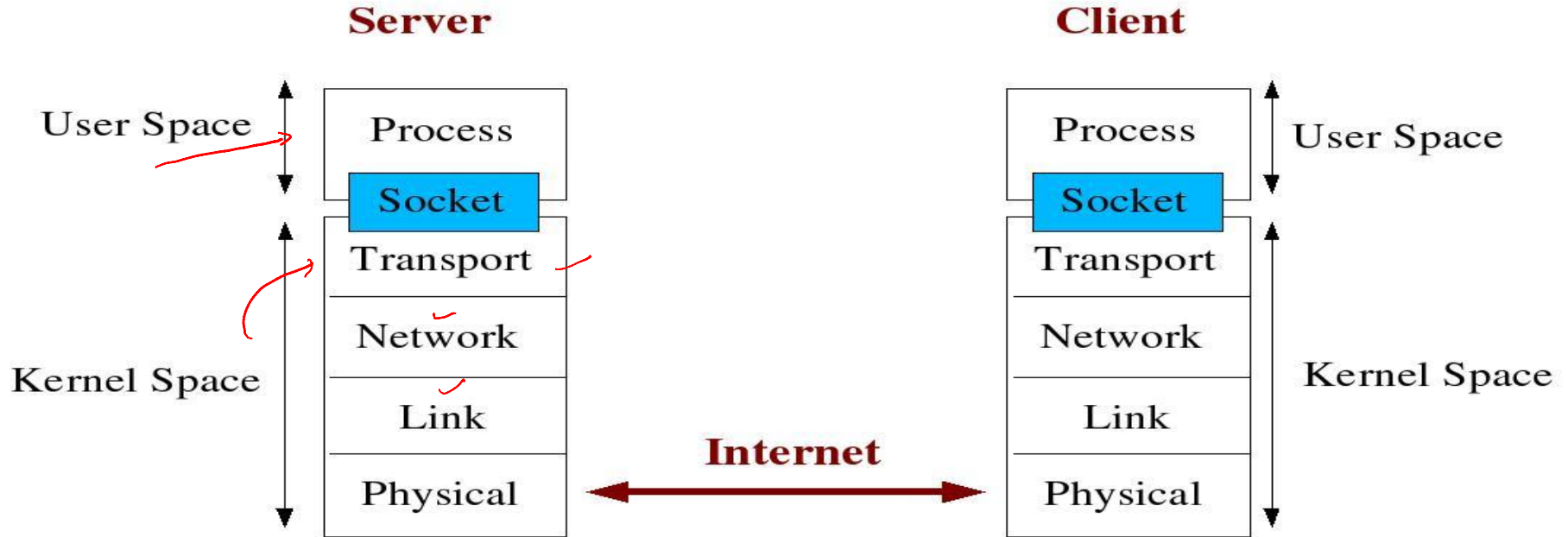
Source Port	Destination Port
Other fields in header	
Application Data	

Transport Layer Segment

What is a socket?

- Socket: An interface between an application process and transport layer
 - The application process can send/receive messages to/from another application process (local or remote) via a socket
- In Unix jargon, a socket is a file descriptor – an integer associated with an open file
- Types of Sockets: **Internet Sockets**, unix sockets, X.25 sockets etc
 - Internet sockets characterized by IP Address (4 bytes), port number (2 bytes)

Socket Description



Types of Internet Sockets

- Stream Sockets (SOCK_STREAM)
 - Connection oriented
 - Rely on TCP to provide reliable two-way connected communication
- Datagram Sockets (SOCK_DGRAM)
 - Rely on UDP
 - Connection is unreliable

Byte Ordering

- Two types of “Byte ordering”
 - Big-Endian (Network Byte Order): High-order byte of the number is stored in memory at the lowest address
 - Little-Endian: Low-order byte of the number is stored in memory at the lowest address
 - Some hosts use this ordering
 - Network stack (TCP/IP) expects Network Byte Order

Byte Ordering

- Conversions:
 - htons() - Host to Network Short
 - htonl() - Host to Network Long
 - ntohs() - Network to Host Short
 - ntohl() - Network to Host Long

Connectionless Protocol

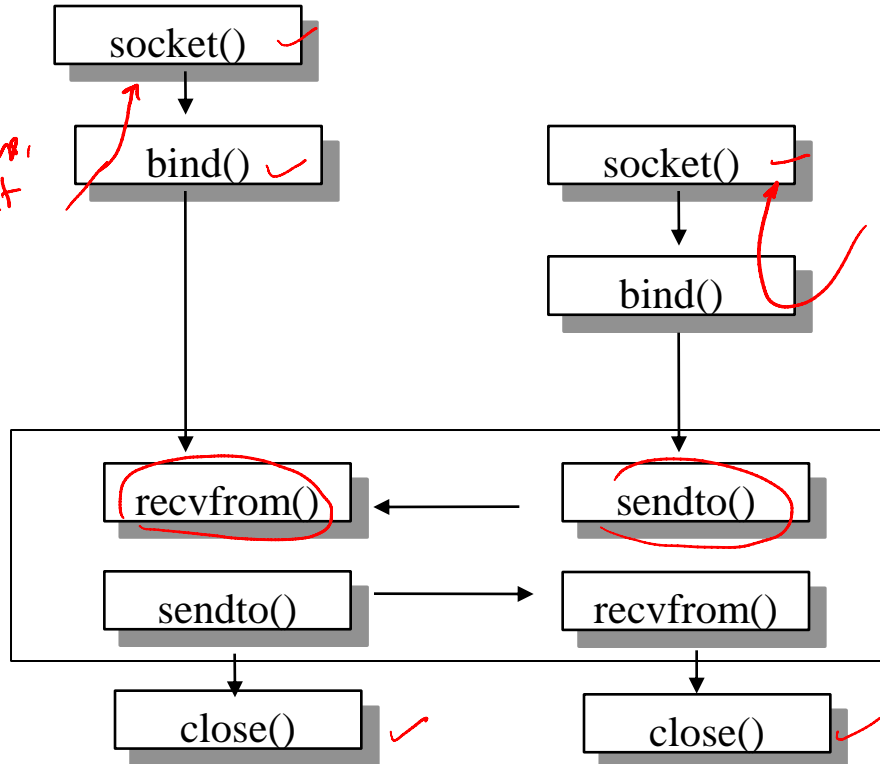
Server

Client

UDP

we bind the ip,port to a socket.. this gives an identity to the socket.

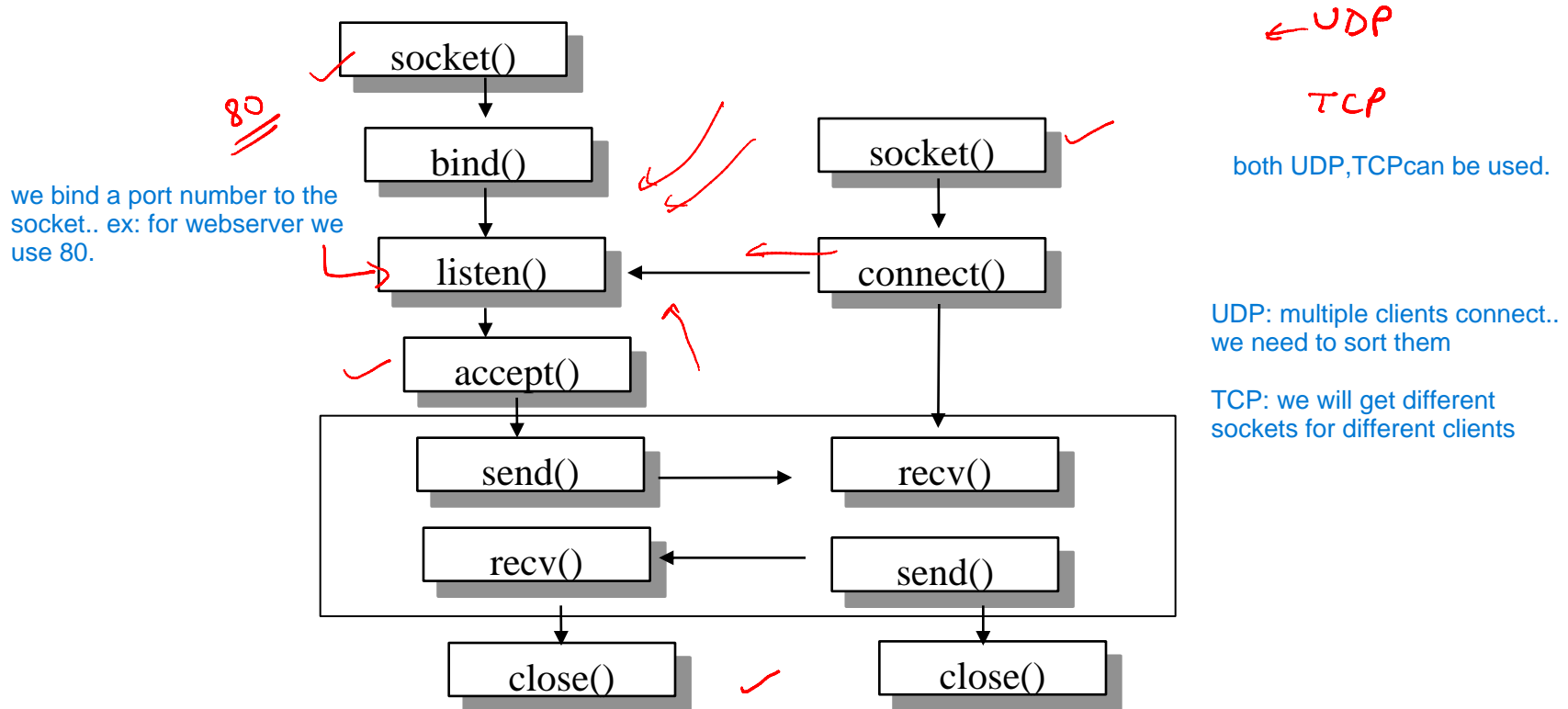
IP address, port



Connection Oriented Protocol

Server

Client



socket() -- Get the file descriptor

- int socket(int domain, int type, int protocol);
 - domain should be set to PF_INET
 - type can be SOCK_STREAM or SOCK_DGRAM
↳ TCP↳ UDP
 - set protocol to 0 to have socket choose the correct protocol based on type
 - socket() returns a socket descriptor for use in later system calls or -1 on error

```
[ int sockfd;  
  sockfd = socket (PF_INET, SOCK_STREAM, 0);
```

bind() - what port am I on?

- Used to associate a socket with a port on the local machine
 - The port number is used by the kernel to match an incoming packet to a process) de-multiplexing
- int bind(int sockfd, struct sockaddr *my_addr, int addrlen)
 - sockfd is the socket descriptor returned by socket()
 - my_addr is pointer to struct sockaddr that contains information about your IP address and port
 - addrlen is set to sizeof(struct sockaddr)
 - returns -1 on error

bind() - failure

- All ports below 1024 are reserved
- You can use ports above 1024 upto 65535 provided there are not already in use
- Re-running a server may result in bind failure
 - Why? Socket still around in kernel using the port
 - Solution: Wait a minute or two or use function setsockopt() to clear the socket

Socket Structures

- struct sockaddr: Holds socket address information for many types of sockets

not only internet sockets , but many other types of sockets

```
struct sockaddr {  
    ✓ unsigned short  sa_family;    //address family AF_XXX  
    ✓ unsigned short  sa_data[14]; //14 bytes of protocol addr  
}
```

if we are using internet
family we use INET

INET

↑

Socket Structures

- struct sockaddr_in: A parallel structure that makes it easy to reference elements of the socket address

```
struct sockaddr_in {  
    short int  
    ✓ unsigned short int  
    struct in_addr  
    unsigned char  
}
```

host byte order

```
sin_family; // set to AF_INET  
sin_port; - 2 // Port number  
[ sin_addr; → 4 // Internet address  
sin_zero[8]; //set to all zeros
```

sin_family can be in a host byte order because it is not sent in the packets, but port, ip are actually sent in packets.. so Net-Byte-Order

- sin_port and sin_addr must be in **network byte order**

Populating the structure

```
struct in_addr {  
    unsigned long s_addr; // that's 32-bit long, or 4 bytes  
};
```

now how to populate these structs?

- int inet_aton(const char *cp, struct in_addr *inp);

#define MYPORT 80

```
struct sockaddr_in my_addr;  
my_addr.sin_family = AF_INET; ✓  
my_addr.sin_port = htons(MYPORT);  
inet_aton("10.0.0.5", &(my_addr.sin_addr));  
memset(&(my_addr.sin_zero), '\0', 8);
```

ascii field to network

– inet_aton() gives non-zero on success; zero on failure

- To convert binary IP to string: `inet_noa()` network to ascii
`printf("%s", inet_ntoa(my_addr.sin_addr));`
- `my_addr.sin_port = 0;` //choose an unused port at random
- `my_addr.sin_addr.s_addr = INADDR_ANY;` //use my IP adr

Example

int sockfd;

struct sockaddr_in my_addr;

sockfd = socket(PF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET; // host byte order

my_addr.sin_port = htons(MYPORT); // short, network byte order

my_addr.sin_addr.s_addr = inet_addr("10.0.0.1");

memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

/****** Code needs error checking. Don't forget to do that *****/

sendto() and recvfrom() - DGRAM style

→ UDP SOCK-DGRAM

- int sendto(int sockfd, const void *msg, int len, int flags, const struct sockaddr *to, int tolen);

600 bytes

- sockfd: socket descriptor you want to send data to
- msg is pointer to the data you want to send
- to is a pointer to a struct sockaddr which contains the destination IP and port
- tolen is sizeof(struct sockaddr)

we may have sent 500, but it only could have sent 400.. and the function returns the same. Its ur job to keep track of it and send the remaining later.. the application layer should take care of those


- Set flags to zero
- Function returns the number of bytes actually sent or -1 on error

[500 bytes → 100
400 bytes

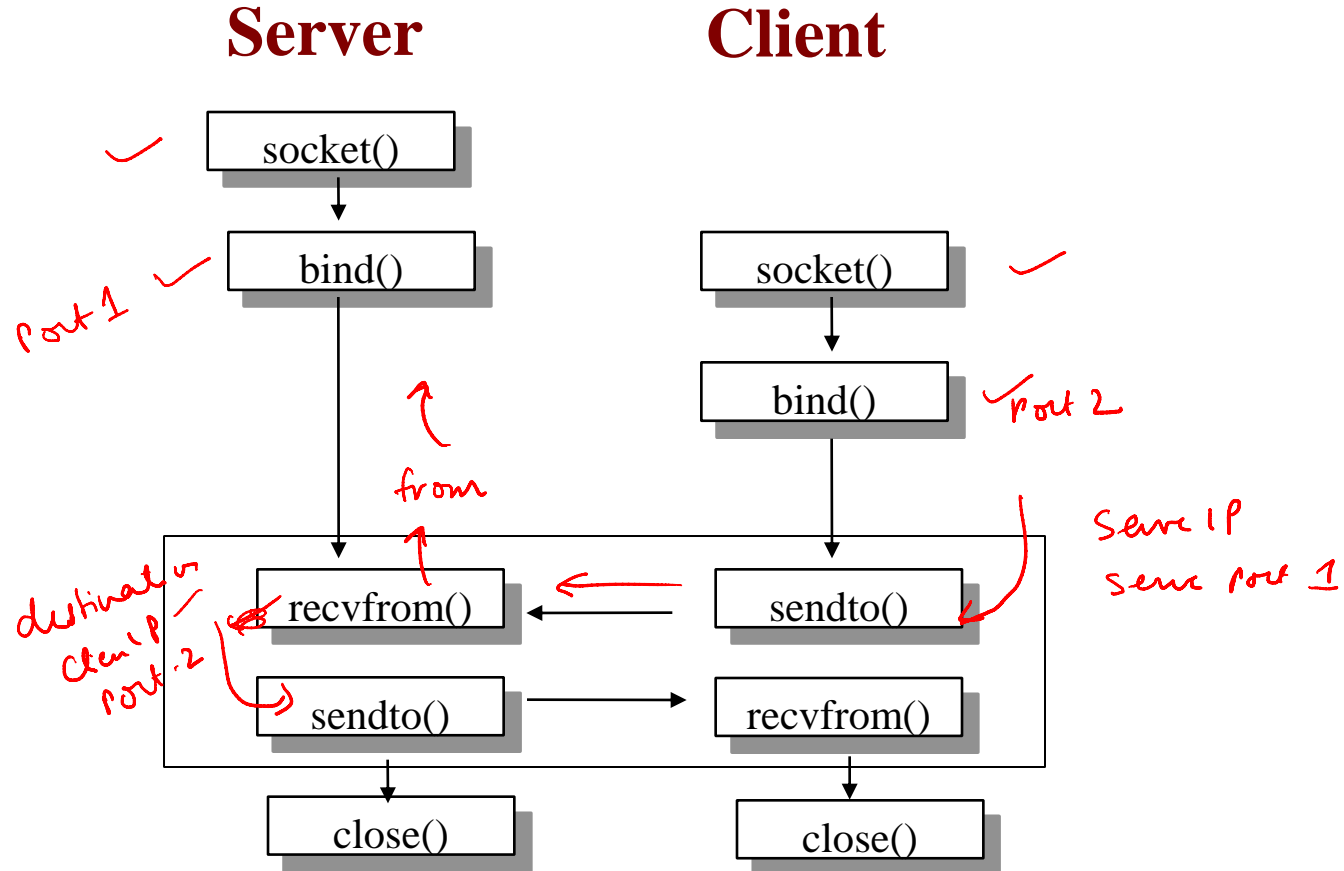
sendto() and recvfrom() - DGRAM style

- `int recvfrom(int sockfd, void *buf, int len, int flags, struct sockaddr *from, int *fromlen);`
 - *sockfd*: socket descriptor to read from
 - *buf*: buffer to read the information from
 - *len*: maximum length of the buffer
 - *flags* set to zero
 - *from* is a pointer to a local struct sockaddr that will be filled with IP address and port of the originating machine
 - *fromlen* will contain length of address stored in *from*
 - Returns the number of bytes received or -1 on error

close() - Bye Bye!

- int close(int sockfd);
 - Closes connection corresponding to the socket descriptor and frees the socket descriptor
 - Will prevent any more sends and receives

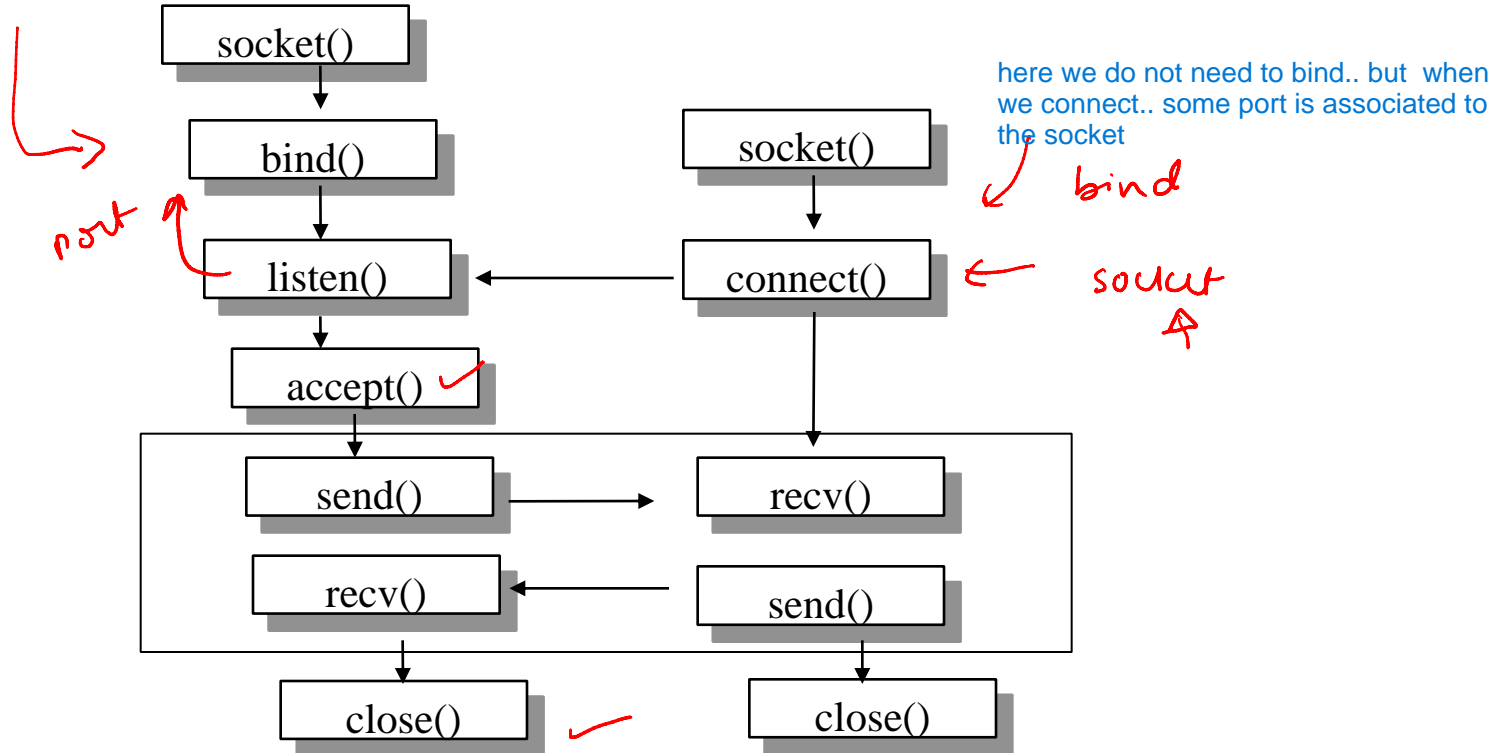
Connectionless Protocol



Connection Oriented Protocol

Server

Client



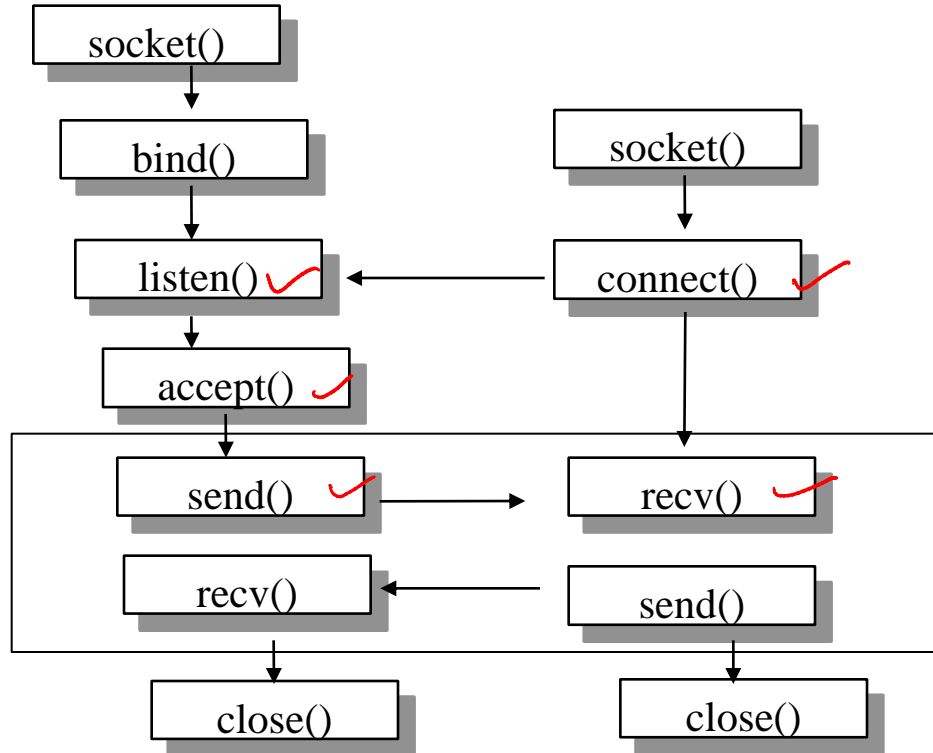
Break



Connection Oriented Protocol

Server

Client



connect() - Hello!

- Connects to a remote host
- int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)
 - sockfd is the socket descriptor returned by socket()
 - serv_addr is pointer to struct sockaddr that contains information on destination IP address and port
 - addrlen is set to sizeof(struct sockaddr)
 - returns -1 on error
- No need to bind(), kernel will choose a port

src, src IP
random

tuple
↳ socket

Example

```
#define DEST_IP  "10.2.44.57"
```

```
#define DEST_PORT 5000
```

```
main(){
```

```
    int sockfd;
```

```
    struct sockaddr_in dest_addr; // will hold the destination addr
```

```
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

```
    dest_addr.sin_family = AF_INET; // host byte order
```

```
    dest_addr.sin_port = htons(DEST_PORT); // network byte order
```

```
    dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
```

```
    memset(&(dest_addr.sin_zero), '\0', 8); // zero the rest of the struct
```

```
    connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
```

```
    /***** Don't forget error checking *****/
```

✓ client random src port

listen() - Call me please!

- Waits for incoming connections
- `int listen(int sockfd, int backlog);`
 - sockfd is the socket file descriptor returned by `socket()`
 - backlog is the number of connections allowed on the incoming queue
 - `listen()` returns -1 on error
 - Need to call bind() before you can listen()

accept() - Thank you for calling !

- accept() gets the pending connection on the port you are listen()ing on
- int accept(int sockfd, void *addr, int *addrlen);
 - sockfd is the listening socket descriptor
 - information about incoming connection is stored in addr which is a pointer to a local struct sockaddr_in
 - addrlen is set to sizeof(struct sockaddr_in)
 - accept returns a new socket file descriptor to use for this accepted connection and -1 on error *↪ send/recv data*

Example

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

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

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

server side

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

```
#define MYPORT 3490 // the port users will be connecting to
```

```
#define BACKLOG 10 // pending connections queue will hold
```

```
main(){
```

```
    int sockfd, new_fd; // listen on sock_fd, new connection on new_fd
```

```
    struct sockaddr_in my_addr; // my address information
```

```
    struct sockaddr_in their_addr; // connector's address information
```

```
    int sin_size;
```

✓ TCP

```
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
```


```
my_addr.sin_family = AF_INET;      // host byte order
my_addr.sin_port = htons(MYPORT);  // short, network byte order
my_addr.sin_addr.s_addr = INADDR_ANY; // auto-fill with my IP
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

// don't forget your error checking for these calls:

bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
listen(sockfd, BACKLOG);

sin_size = sizeof(struct sockaddr_in);

new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
```

 client info

send() and recv() - Let's talk!

- The two functions are for communicating over stream sockets or connected datagram sockets.
bind, listen, accept
- `int send(int sockfd, const void *msg, int len, int flags);`
 - `sockfd` is the socket descriptor you want to send data to (got from `accept()`)
 - `msg` is a pointer to the data you want to send
 - `len` is the length of that data in bytes
 - set `flags` to 0 for now
 - `send()` returns the number of bytes actually sent (may be less than the number you told it to send) or -1 on error

Example

```
char *msg = “hello!”;
```

```
int len, bytes_sent;
```

```
.....
```

✓

]

```
len = strlen(msg);
```

```
bytes_sent = send(sockfd, msg, len, 0);
```

↖

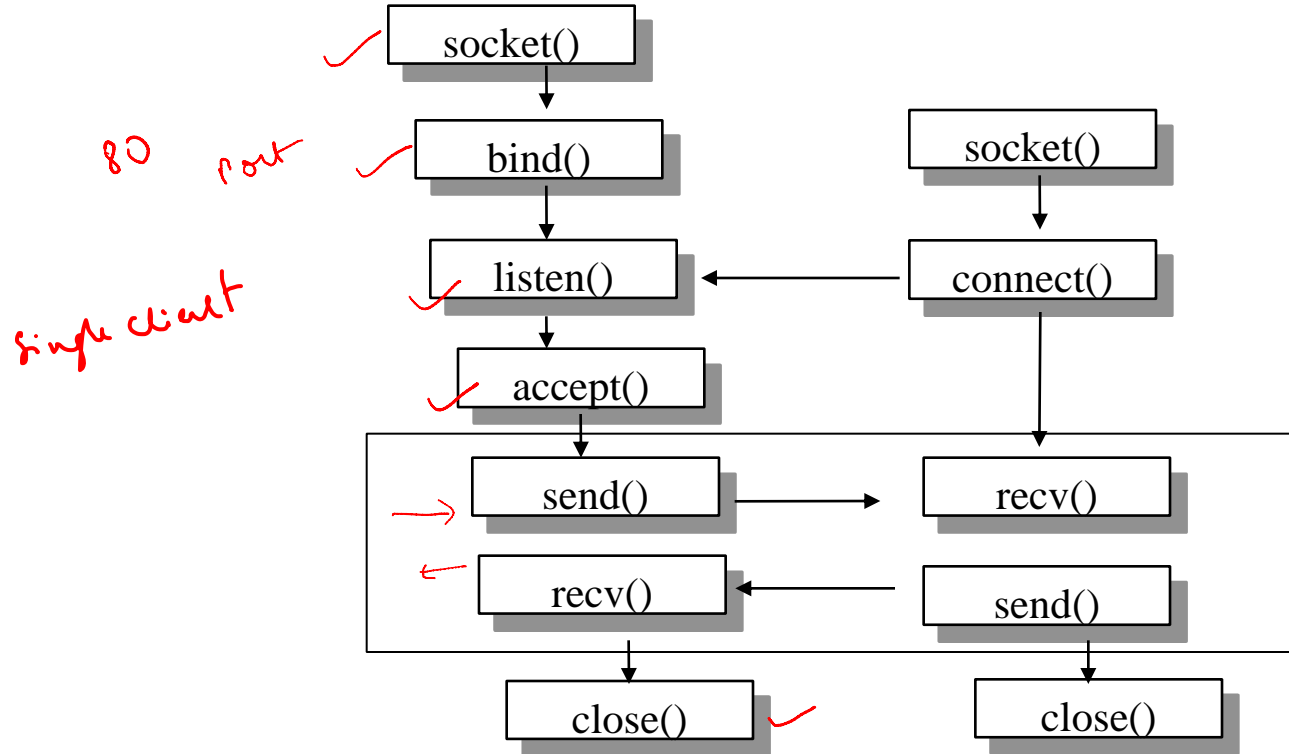
send() and recv() - Let's talk!

- `int recv(int sockfd, void *buf, int len, int flags);`
 - sockfd is the socket descriptor to read from
 - buf is the buffer to read the information into
 - len is the maximum length of the buffer
 - set flags to 0 for now
 - `recv()` returns the number of bytes actually read into the buffer or -1 on error
 - If `recv()` returns 0, the remote side has closed connection on you

Connection Oriented Protocol

Server


Client



Break



Miscellaneous Routines

- int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);


- Will tell who is at the other end of a connected stream socket and store that info in *addr*

- int gethostname(char *hostname, size_t size);


- Will get the name of the computer your program is running on and store that info in hostname

Miscellaneous Routines

- Provides DNS service: `struct hostent *gethostbyname(const char *name);`

```
struct hostent {  
    char *h_name;           //official name of host  
    char **h_aliases;       //alternate names for the host  
    int h_addrtype;         //usually AF_INET  
    int h_length;           //length of the address in bytes  
    char **h_addr_list;     //array of network addresses for the host  
}  
#define h_addr h_addr_list[0]
```

- Example Usage:

```
struct hostent *h;  
h = gethostbyname("www.iitb.ac.in");  
printf("Host name : %s \n", h->h_name);  
printf("IP Address: %s\n", inet_ntoa(*((struct in_addr *)h->h_addr)));
```

Input/Output Multiplexing

client 1 ✓
client 2 ✓

blocking functions --- they will not return unless some event has happened

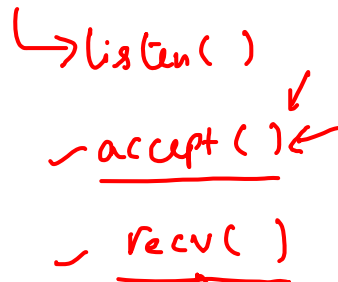
- Some routines like accept(), recv() block

- Make sockets non-blocking

```
sockfd = socket(PF_INET, SOCK_STREAM, 0);  
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

- Polling (consumes CPU time)

- Fork a separate process for each I/O channel ✓
- Threading ✓
- Select system call (HIGHLY RECOMMENDED)



so is we block bcz of accept(), then we will not be able to recv() messages... and if we block bcz of recv() we will not be able to accept() new connections.
The solutions are these...

Select()

→ list of file descriptors

list → listen, client 1, client 2
sockets

- `int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`
 - `numfds`: highest file descriptor + 1
 - `Readfds`, `writefds`, `exceptfds`: set of file descriptors to monitor for read, write and exception operations
 - When `select()` returns, the set of file descriptors is modified to reflect the one that is currently ready
 - Timeout: select returns after this period if it still hasn't found any ready file descriptors

```
struct timeval {  
    int tv_sec; // seconds  
    int tv_usec; // microseconds  
};
```


Useful Macros

- FD_ZERO(fd_set *set)
 - clears a file descriptor set
- FD_SET(int fd, fd_set *set)
 - adds fd to the set
- FD_CLR(int fd, fd_set *set)
 - removes fd from the set
- FD_ISSET(int fd, fd_set *set)
 - tests to see if fd is in the set

Example

```
#define STDIN 0 // file descriptor for standard input
```

```
int main(void) {
```

```
    struct timeval tv;
```

```
    fd_set readfds;
```

```
    tv.tv_sec = 2;
```

```
    tv.tv_usec = 500000;
```

```
    FD_ZERO(&readfds);
```

```
    FD_SET(STDIN, &readfds);
```

```
    // don't care about writefds and exceptfds:
```

```
    select(STDIN+1, &readfds, NULL, NULL, &tv);
```

Example Cont....

```
if (FD_ISSET(STDIN, &readfds))  
    printf("A key was pressed!\n");  
else  
    printf("Timed out.\n");  
return 0;  
}
```

Summary

- Sockets help application process to communicate with each other using standard Unix file descriptors
- Two types of Internet sockets: SOCK_STREAM ^{→ TCP} and SOCK_DGRAM ^{→ UDP}
- Many routines exist to help ease the process of communication

References

- Books:

- Unix Network Programming, volumes 1-2 by W. Richard Stevens.
- TCP/IP Illustrated, volumes 1-3 by W. Richard Stevens and Gary R. Wright

- Web Resources:

- Beej's Guide to Network Programming

(These slides followed 2001 version, there is a 2012 version that includes IPv6)