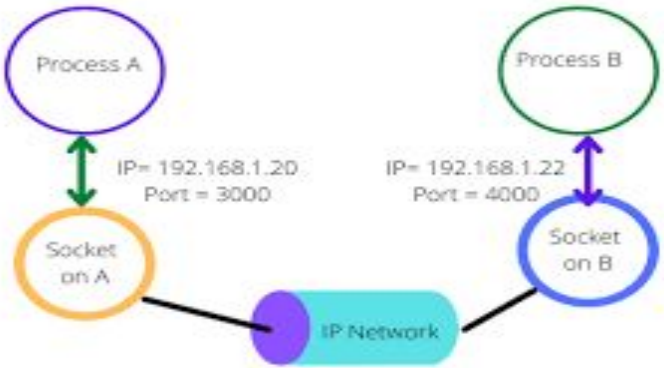
Socket Programming in C

CN Tutorial 3rd Feb,2024

- Saket Biju

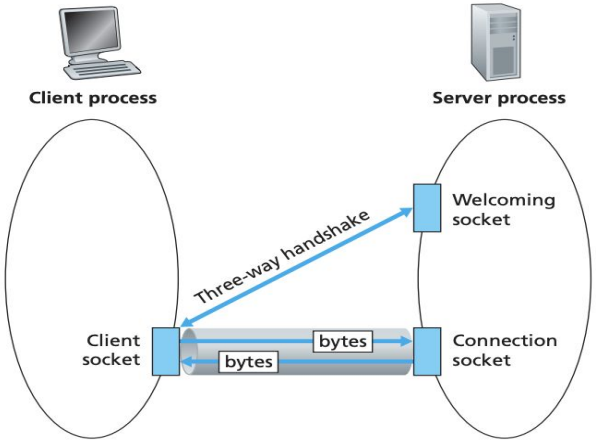
What is a socket?

- Well, basically it’s a link between your application and the underlying network
- Associated with a file descriptor



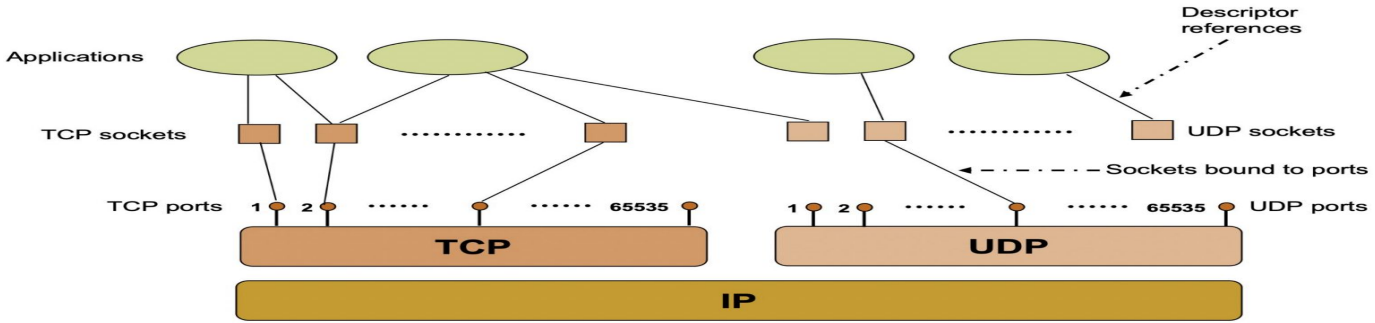
Types of sockets

1. Stream Sockets(TCP)
    a. Reliable
    b. 2-way connected
    c. In-order
    d. Error-free
    e. Slower
    f. ssh, http, ftp
2. Datagram Sockets(UDP)
    a. Unreliable
    b. Connectionless
    c. May be out of order
    d. Error-free
    e. Faster
    f. dhcp, video streaming



Addressing a socket

- IP Addresses
  - IPv4, IPv6
  - 127.0.0.1 → loopback address(localhost)
  - 0.0.0.0 → listens to all available interfaces
- Port Numbers
  - 16-bit numbers
  - 0-1023 are usually reserved and unusable for users
- TCP Socket(src ip, dst ip, src port, dst port)
- UDP Socket(dst ip, dst port)



Byte ordering

- Some computers use Big-Endian while others use Little-Endian
- However Network Byte Order is fixed to be Big-Endian
- Need to convert from host byte order to network byte order and vice-versa
- htons(), htonl(), ntohs(), ntohl()
- htons(5100) ⇒ htons(0x13EC) ⇒ 0xEC13

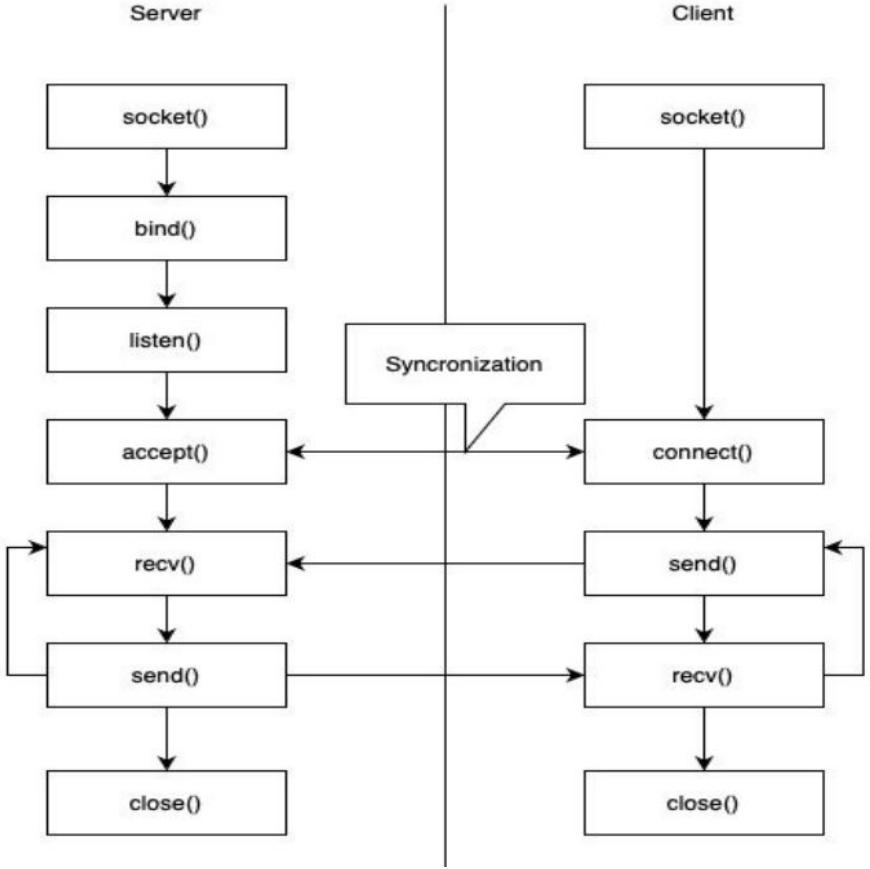
Client-server communication

- Server
  - Passively waits for client connections and requests
  - Usually has a fixed ip and port# associated with the socket
  - Passive socket
- Client
  - Initiates the communication with the server
  - Should be aware of server ip and port#
  - Active socket

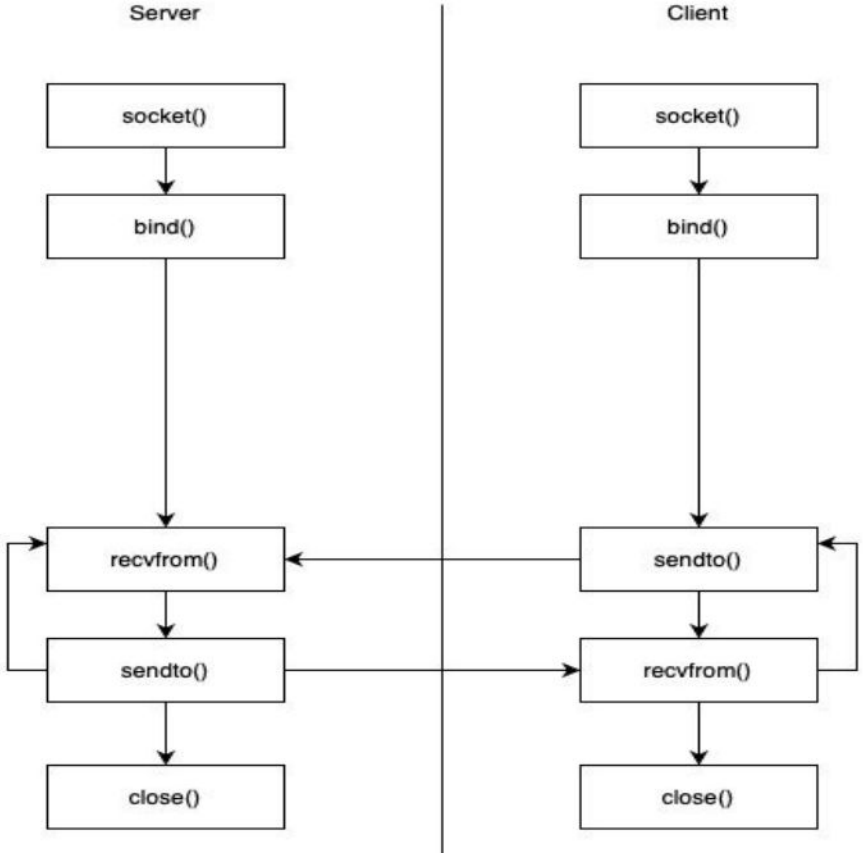
System calls (very important)

1. Socket
2. Bind
3. Listen
4. Accept
5. Connect
6. Send
7. Recv
8. Close

Client-Server Communication (TCP)



Client-Server Communication (UDP)



1. SOCKET

```
int sockfd = socket(int domain, int type, int protocol);
```

- sockfd is the unique socket descriptor(-1 on error)
- domain is the address family
  - AF_INET: IPv4
  - AF_INET6: IPv6
  - AF_UNIX: Local communication
- type specifies stream or datagram
  - SOCK_STREAM: TCP
  - SOCK_DGRAM: UDP
- protocol
  - IPPROTO_TCP, IPPROTO_UDP
  - Usually **set to 0** to choose the default

SOME IMPORTANT (DATA)STRUCTS!

```
struct sockaddr_in {
    __uint8_t       sin_len;
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct  in_addr sin_addr;
    char            sin_zero[8];
};
```

```
struct sockaddr_in6 {
    __uint8_t       sin6_len;      /* length of this struct(sa_family_t) */
    sa_family_t     sin6_family;   /* AF_INET6 (sa_family_t) */
    in_port_t       sin6_port;     /* Transport layer port # (in_port_t) */
    __uint32_t      sin6_flowinfo; /* IP6 flow information */
    struct in6_addr sin6_addr;     /* IP6 address */
    __uint32_t      sin6_scope_id; /* scope zone index */
};
```

```
struct sockaddr {
    __uint8_t       sa_len;
    sa_family_t     sa_family;
    char            sa_data[14];
};
```

```
struct in_addr {
    in_addr_t s_addr;
};
```

```
typedef struct in6_addr {
    union {
        __uint8_t   __u6_addr8[16];
        __uint16_t  __u6_addr16[8];
        __uint32_t  __u6_addr32[4];
    } __u6_addr;                    /* 128-bit IP6 address */
} in6_addr_t;
```

2. BIND

```
int status = bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

- status holds -1 if error
- sockfd is the socket descriptor
- my_addr is a pointer to struct sockaddr that stores ip and port
- addrlen stores the size of the my_addr struct

USEFUL FUNCTIONS

`htons(uint_16)`

- converts host byte order to network byte order of short integer

`ntohs(uint_16)`

- converts network byte order to host byte order of short integer

- Similarly, `htonl(uint_32)` and `ntohl(uint_32)` for long
- Used when specifying ip or port numbers to sockadd_in

BIND EXAMPLE

```
int sockfd;
struct sockaddr_in addrport;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
addrport.sin_family = AF_INET;
addrport.sin_port = htons(5100);
addrport.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(sockid, (struct sockaddr *) &addrport, sizeof(addrport))!= -1) {
...}
```

USAGE OF BIND

- While receiving incoming connections, the sender has to know the specific port at which the socket listens to.
- During sending, it is usually not necessary.

USEFUL FUNCTIONS

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

- af: Address Family – AF_INET, AF_INET6
- src: IP address string in human readable format
- dst: Pointer to the location where the binary representation of IP address

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

- af: Address Family – AF_INET, AF_INET6
- src: Pointer to the binary representation of the IP address
- dst: Pointer to the buffer where the resulting string representation will be stored
- size: The size of the buffer pointed to by ***dst***

Refer Usage in Example Client-Server Program!!

SOME HELPFUL FUNCTIONS :)

Add the below snippet to your code for reusing the port once disconnected.

```
int yes = 1;
if(setsockopt(server_socket_fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1){
    perror("setsockopt");
    exit(1);
}
```

3. LISTEN

```
int status = listen(int sockfd, int queue);
```

- status is -1 on error
- sockfd is the socket descriptor
- queue denotes the limit on the number of incoming connections that can queue up until it is accept()'d.
- Only used by servers to accept() new connections

4. CONNECT

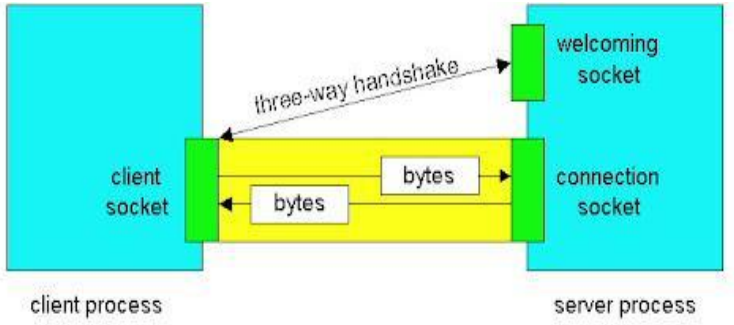
```
int status = connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

- status is -1 on error
- sockfd is the socket descriptor
- serv_addr contains destination ip and port#
- addrlen stores the size of **serv_addr**
- It is a blocking call

5. ACCEPT

```
int new_sockfd = accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- new_sockfd is a new socket descriptor returned by accept to use for all send() and recv(), returns -1 on error
- sockfd is the socket descriptor(which listens for incoming requests)
- addr stores the incoming connection's details like ip and port#
- addrlen stores the size of addr
- It is a blocking call



6. SEND

```
int send_count = send(int sockfd, const void *msg, int len, int flags);
```

- msg specifies the data(notice it is void*)
- len indicates the length of the message
- flags usually set to 0
- send_count returns the number of bytes sent, -1 on error
- send_count <= len
- if send_count < len, you should manually handle the resending
- blocks if send buffer full

7. RECV

```
int recv_count = recv(int sockfd, void *buf, int len, int flags);
```

- buf specifies the location to store the incoming data(again void*)
- len indicates the size of the buffer **buf**
- flags usually set to 0
- recv_count returns the number of bytes read into the buffer, -1 on error
- blocks if receive buffer empty

8. close

```
int status = close(sockfd);
```

- sockfd is the socket descriptor
- status is -1 if error
- closes a connection if stream socket

SENDTO AND RECVFROM

- For sending datagrams
- Lookup the syntax yourselves

Additional system calls

```
int status = getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

⇒ Stores the sockaddr_info of the socket **sockfd** in **addr**

MULTITHREADING

- Necessary when multiple clients interact with server simultaneously
- Main thread will listen to incoming connections
- New threads will be allocated for interacting with each client
- Necessary when client gets blocked by multiple calls
- Semaphores may be used to solve Critical Section Problem

SOME USEFUL STRING AND FILE FUNCTIONS

- strcat
- **strtok**
- strcpy
- **sprintf**
- strlen
- **fgets**
- fscanf
- fprintf

*Bolded functions will be of very much use if you learn their working properly

Error-checking

- It’s always a good practice to have error checking codes.
- Time consumed initially will be saved multifold later while debugging…BELIEVE ME!!!
- Go through the code and understand how error checking is being carried out.

USEFUL REFERENCES

https://beej.us/guide/bgnet/pdf/bgnet_usl_c_1.pdf