Network Programming

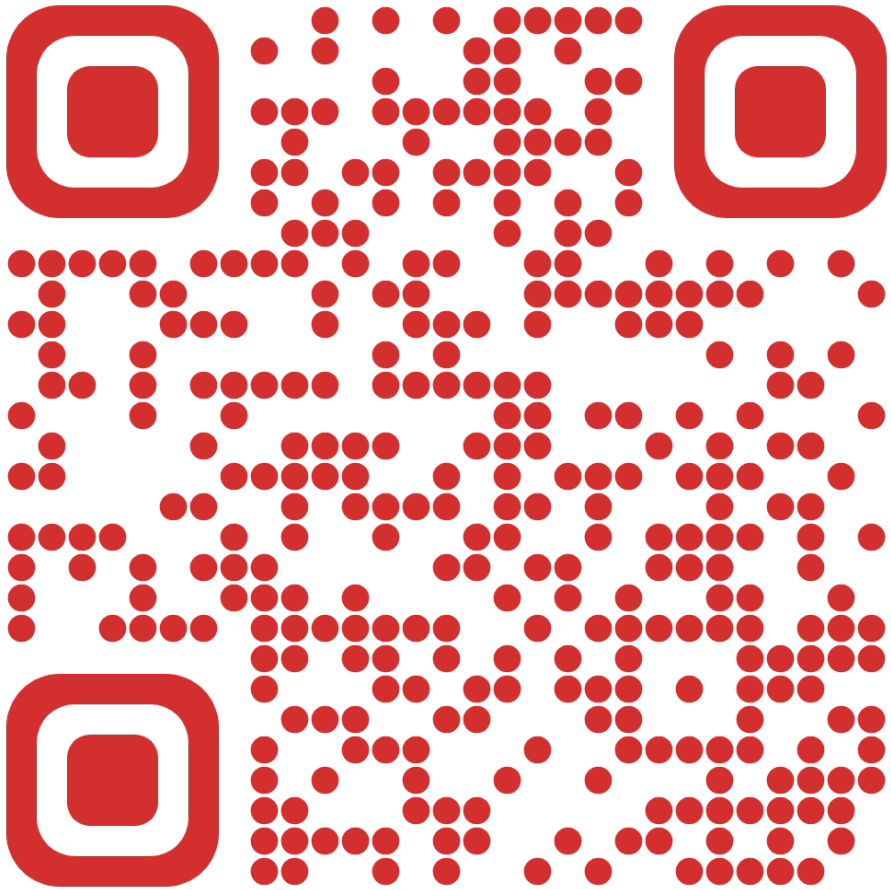
Socket Programming

Rathindra Nath Dutta

Senior Research Fellow
Advanced Computing & Microelectronics Unit
Indian Statistical Institute, Kolkata

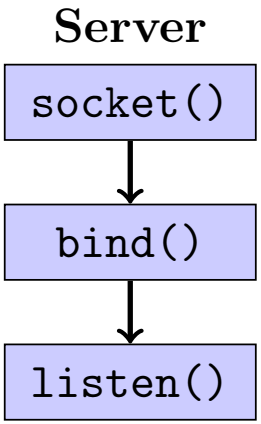
October 9, 2023

https://www.isical.ac.in/~rathin_r/uploads/CN/



WEB PAGE

Socket Programming Using C



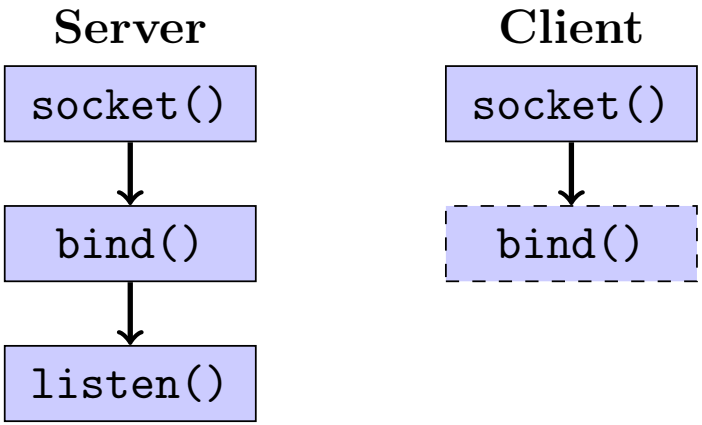
`socket()` creates and returns a socket descriptor representing an endpoint for communications

Servers must bind a unique name to a socket descriptor using `bind()` to make it accessible from the network

`listen()` call shows willingness to accept client connection requests

NB: a socket cannot actively initiate any connection requests after a `listen()` call

Socket Programming Using C

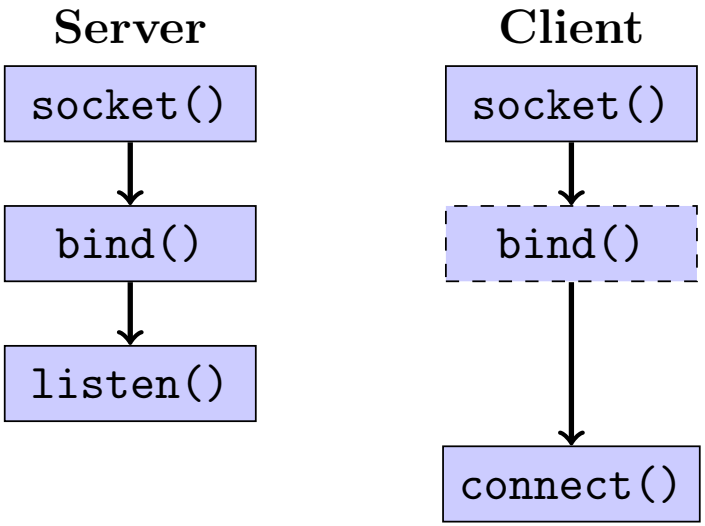


Meanwhile on the client side:

A socket file descriptor is created similarly

(Optionally) the client tries to bind it

Socket Programming Using C



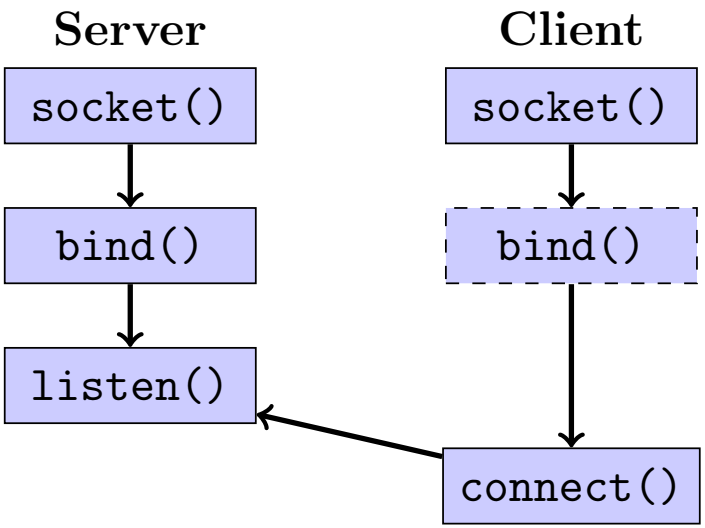
Meanwhile on the client side:

A socket file descriptor is created similarly

(Optionally) the client tries to bind it

The client invokes a `connect()` request on the stream socket to establish a connection to the server

Socket Programming Using C



Meanwhile on the client side:

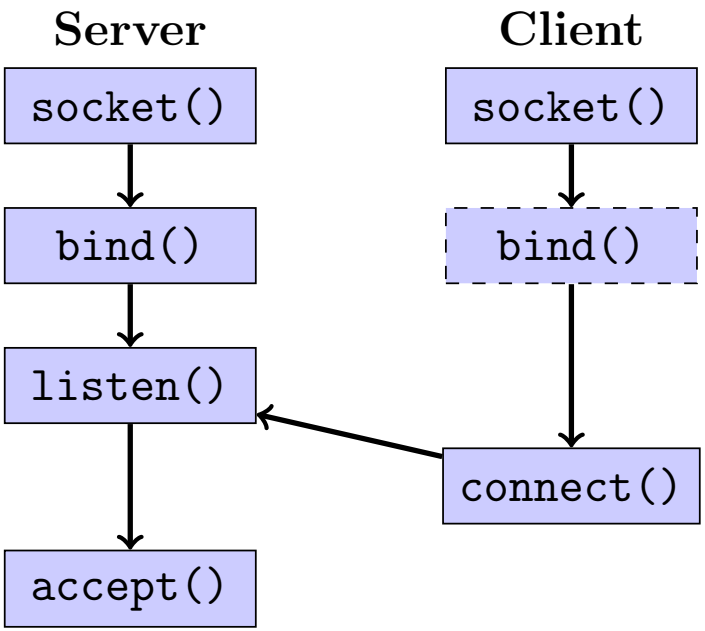
A socket file descriptor is created similarly

(Optionally) the client tries to bind it

The client invokes a `connect()` request on the stream socket to establish a connection to the server

The request arrives at the server

Socket Programming Using C



Meanwhile on the client side:

A socket file descriptor is created similarly

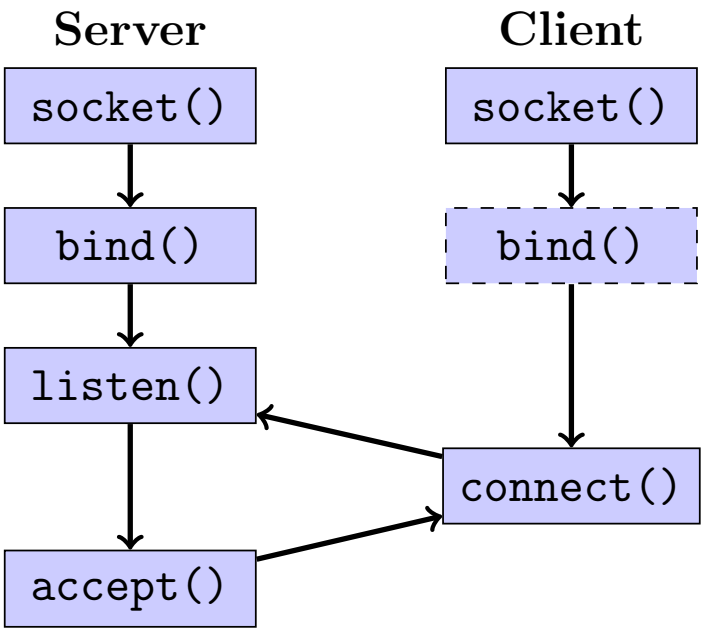
(Optionally) the client tries to bind it

The client invokes a `connect()` request on the stream socket to establish a connection to the server

The request arrives at the server

Server may choose to honor that request via `accept()`

Socket Programming Using C



Meanwhile on the client side:

A socket file descriptor is created similarly

(Optionally) the client tries to bind it

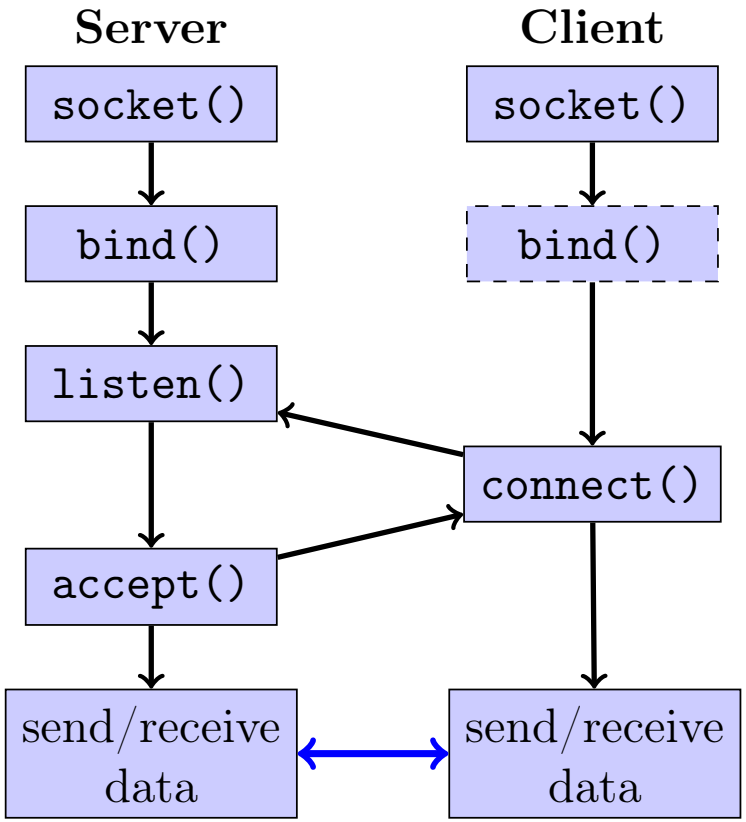
The client invokes a `connect()` request on the stream socket to establish a connection to the server

The request arrives at the server

Server may choose to honor that request via `accept()`

The client is informed that the connection request has been accepted

Socket Programming Using C



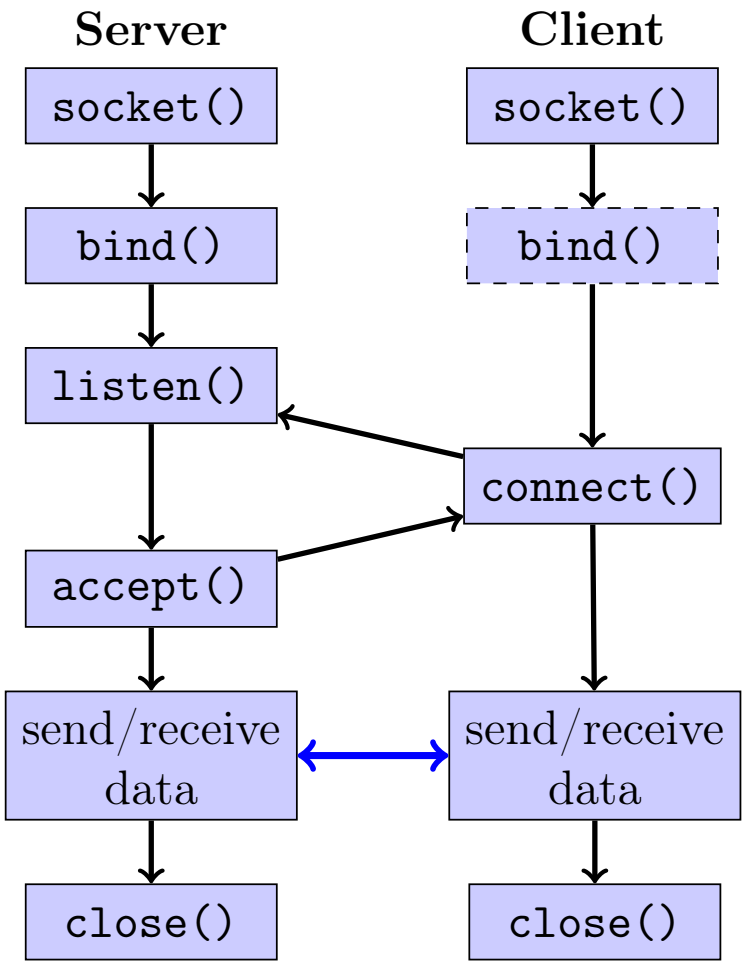
Now client and server can communicate with each other

We can use `read()`, `write()` APIs for stream I/O

There are socket specific APIs[1] such as `send()`, `recv()`, `sendto()`, `recvfrom()`, `sendmsg()`, `recvmsg()`

[1] See man pages: https://linux.die.net/man/2/send and https://linux.die.net/man/2/recv

Socket Programming Using C



Finally, when a server or client wants to stop operations, it issues a `close()` call to release any system resources acquired by the socket

The socket() API

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Return value: On success, a file descriptor (some positive number) for the new socket is returned. On error, -1 is returned

Parameters:
`domain`: specifies a protocol family (communication domain), e.g.

protocol family	description
AF_UNIX	Local inter-process communication
AF_INET	Remote communication via IPv4 Internet Protocol
AF_INET6	Remote communication via IPv6 Internet Protocol

`type`: specifies the communication semantics, e.g.

common types	description	default protocol
SOCK_STREAM	sequenced, reliable, two-way, connection-oriented byte streams	TCP
SOCK_DGRAM	connectionless, unreliable messages of a fixed maximum length	UDP

`protocol`: specifies a particular protocol to be used (0 implies default)

[1] See man pages: https://man7.org/linux/man-pages/man2/socket.2.html and https://man7.org/linux/man-pages/man7/socket.7.html

The bind() API

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

Binds the address specified by `addr` to the socket referred to by the file descriptor `sockfd`

Return value: On success, zero is returned. On error, -1 is returned

Parameters:
`sockfd`: a socket file descriptor created with `socket()`

`addr`: a pointer to an address structure
actual structure depends on the socket address family

`addrlen`: specifies the size, in bytes, of the address structure `addr`

[1] See man page: https://man7.org/linux/man-pages/man2/bind.2.html

Address Structure for AF_INET

```c
#include <sys/socket.h>
#include <netinet/in.h>
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order*/
    struct in_addr sin_addr;   /* internet address */
};
/* Internet address */
struct in_addr {
    uint32_t       s_addr; /* address in network byte order */
};
```

special addresses	IP	description
INADDR_LOOPBACK	127.0.0.1	localhost
INADDR_ANY	0.0.0.0	any address for binding
INADDR_BROADCAST	255.255.255.255	any host[2]

[1] See the man page: https://man7.org/linux/man-pages/man7/ip.7.html

[2] INADDR_BROADCAST has the same effect on bind as INADDR_ANY for historical reasons

Binding a Socket to an Address

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0); // TCP socket
char IP[] = "127.0.0.1";
int portno = 54321;

struct sockaddr_in sock_addr;
bzero((char *)&sock_addr, sizeof(sock_addr)); // clear

sock_addr.sin_family = AF_INET;
sock_addr.sin_port = htons(portno);
sock_addr.sin_addr.s_addr = inet_addr(IP); // client
// sock_addr.sin_addr.s_addr = INADDR_LOOPBACK // localhost
// sock_addr.sin_addr.s_addr = INADDR_ANY; // server

bind(sockfd, (struct sockaddr*)&sock_addr, sizeof(sock_addr))
```

[1] `htons()`: converts an unsigned short integer from host byte order to network byte order

[3] `inet_addr()`: converts a IPv4 host address string written in dotted decimal notation, into binary data in network byte order; require `#include <arpa/inet.h>`

The `listen()` API

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Marks the socket referred to by `sockfd` as a passive socket, i.e, a socket to be used to accept incoming connection requests using `accept()`

Return value: On success, zero is returned. On error, -1 is returned

Parameters:
`sockfd`: file descriptor after `bind()` to a socket type, e.g. `SOCK_STREAM`

`backlog`: defines the maximum queue length[2] of pending connections
if a connection request arrives when the queue is full, the client may receive an error with an indication of `ECONNREFUSED` or,
if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt succeeds.

[1] See the man page: https://man7.org/linux/man-pages/man2/listen.2.html

[2] If the `backlog` value is greater than the value in `/proc/sys/net/core/somaxconn`, then it is silently capped to that value. Since Linux 5.4, the default in this file is 4096; in earlier kernels, the default value is 128

The accept() API

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

Used with connection-oriented socket types (e.g. `SOCK_STREAM`)
It extracts the first connection request on the queue of pending connections for the listening socket `sockfd`
Creates a new connected socket, and returns a new file descriptor for it

The newly created socket is not in the listening state
The original socket `sockfd` remains unaffected

Return value: On success, returns a file descriptor for the accepted socket (a nonnegative integer). On error, -1 is returned

[1] See the man page: https://man7.org/linux/man-pages/man2/accept.2.html

The accept() API

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

Parameters:

`sockfd`: a file descriptor of a listening socket

`addr`: a pointer to an address structure of the peer
actual structure depends on the socket address family

`addrlen`: a call-by-address argument; initialized to contain the size (in bytes) of the structure pointed to by `addr`; on return it will contain the actual size of the peer address

[1] See the man page: https://man7.org/linux/man-pages/man2/accept.2.html

A typical connection accept mechanism

```
struct sockaddr_in cli_addr;
int cli_addr_len = sizeof(cli_addr);
int accepted_sockfd = accept(sockfd,
        (struct sockaddr *)&cli_addr, &cli_addr_len);
```

Perform I/O on this `accepted_sockfd`

A typical connection accept mechanism

```
struct sockaddr_in cli_addr;
int cli_addr_len = sizeof(cli_addr);
int accepted_sockfd = accept(sockfd,
        (struct sockaddr *)&cli_addr, &cli_addr_len);
```

Perform I/O on this `accepted_sockfd`

Let us now study `server1.c`

Exercise 1: Displaying Client Info

Modify the Echo Server program: `server1.c`
To Print `telnet` client's IP and Port address

[1] Use `ifconfig -a` or `ip addr` to get local IP address

[2] Use `netstat -na | grep <portno>` to get status of that port

Exercise 1: Displaying Client Info

Modify the Echo Server program: server1.c
To Print `telnet` client's IP and Port address

Solution

```
printf("IP address is: %s\n", inet_ntoa(cli_addr.sin_addr));
printf("port is: %d\n", (int) ntohs(cli_addr.sin_port));
```

Simply uncomment the lines 50 and 51 in server1.c

[1] Use `ifconfig -a` or `ip addr` to get local IP address

[2] Use `netstat -na | grep <portno>` to get status of that port

[3] `inet_ntoa()` converts the Internet host address given in network byte order, to a string in IPv4 dotted-decimal notation

[4] `ntohs()` converts the given unsigned short integer from network byte order to host byte order

The connect() API

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

Connects the socket referred to by the file descriptor `sockfd` to the address specified by `addr`

Return value: On success, zero is returned. On error, -1 is returned

Parameters:
`sockfd`: a socket file descriptor created with `socket()`

`addr`: a pointer to an address structure
actual structure depends on the socket address family

`addrlen`: specifies the size, in bytes, of the address structure `addr`

[1] See man page: https://man7.org/linux/man-pages/man2/connect.2.html

Creating a Client

A typical connection request mechanism:

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0); // TCP socket

struct sockaddr_in serv_addr;
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr=inet_addr("127.0.0.1"); //server IP
serv_addr.sin_port = htons(54321); // server port

connect(sockfd, (struct sockaddr *)&serv_addr,
            sizeof(serv_addr))

// read/write using the sockfd
```

Creating a Client

A typical connection request mechanism:

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0); // TCP socket

struct sockaddr_in serv_addr;
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr=inet_addr("127.0.0.1"); //server IP
serv_addr.sin_port = htons(54321); // server port

connect(sockfd, (struct sockaddr *)&serv_addr,
            sizeof(serv_addr))

// read/write using the sockfd
```

Let us now study client1.c

Testing the Client Program

Download `client1.c` and `server1.c`

Open a terminal for server process
`gcc server1.c -o server && ./server`
leave this window open

Open another terminal for client process
`gcc client1.c -o client && ./client`
send "quit" to stop

Exercise 2

Modify `client1.c` and `server1.c` such that

- server now takes an optional command-line argument specifying the port address; if no argument is given it uses the default 54321

Exercise 2

Modify `client1.c` and `server1.c` such that

- server now takes an optional command-line argument specifying the port address; if no argument is given it uses the default 54321
- client takes two arguments from command-line; the first one is for the server IP address (in dotted decimal notation) and the second one for the server port address

Exercise 2

Modify `client1.c` and `server1.c` such that

- server now takes an optional command-line argument specifying the port address; if no argument is given it uses the default 54321
- client takes two arguments from command-line; the first one is for the server IP address (in dotted decimal notation) and the second one for the server port address
- ⋆ client can also accept any hostname specified in `/etc/hosts` as a server address in the first argument

Sending an Integer over a Socket

- Socket communication uses byte stream
- Integer (or anything) needs to interpreted as raw bytes[1]
- Always write machine independent codes:
  use `htonl()`, `ntohl()`, `uint32_t` or similar things[2]

[1] The process is known as **Serialization**/**Deserialization**

[2] See this discussion: https://stackoverflow.com/questions/9140409/transfer-integer-over-a-socket-in-c

Sending an Integer over a Socket

- Socket communication uses byte stream
- Integer (or anything) needs to interpreted as raw bytes[1]
- Always write machine independent codes:
  use `htonl()`, `ntohl()`, `uint32_t` or similar things[2]

```
uint32_t number_to_send = 10000; // Put your value
uint32_t converted_num = htonl(number_to_send);
write(socketfd, &converted_num, sizeof(uint32_t));
```

[1] The process is known as **Serialization**/**Deserialization**

[2] See this discussion: https://stackoverflow.com/questions/9140409/transfer-integer-over-a-socket-in-c

Sending an Integer over a Socket

- Socket communication uses byte stream
- Integer (or anything) needs to interpreted as raw bytes[1]
- Always write machine independent codes:
  use `htonl()`, `ntohl()`, `uint32_t` or similar things[2]

```
uint32_t number_to_send = 10000; // Put your value
uint32_t converted_num = htonl(number_to_send);
write(socketfd, &converted_num, sizeof(uint32_t));

uint32_t read_num, num;
read(socketfd, &read_num, sizeof(uint32_t));
num = ntohl(read_num); // get the actual value
```

[1] The process is known as **Serialization**/**Deserialization**

[2] See this discussion: https://stackoverflow.com/questions/9140409/transfer-integer-over-a-socket-in-c

Sending an Integer over a Socket

- Socket communication uses byte stream
- Integer (or anything) needs to interpreted as raw bytes[1]
- Always write machine independent codes:
  use `htonl()`, `ntohl()`, `uint32_t` or similar things[2]

```
uint32_t number_to_send = 10000; // Put your value
uint32_t converted_num = htonl(number_to_send);
write(socketfd, &converted_num, sizeof(uint32_t));

uint32_t read_num, num;
read(socketfd, &read_num, sizeof(uint32_t));
num = ntohl(read_num); // get the actual value
```

- Other types (e.g. `float`[3][4]) can also be sent in similar fashion

[1] The process is known as **Serialization**/**Deserialization**

[2] See this discussion: https://stackoverflow.com/questions/9140409/transfer-integer-over-a-socket-in-c

[3] Sending float: https://stackoverflow.com/questions/38511305/sending-float-values-on-socket-c-c

[4] Serialization (How to Pack Data): https://beej.us/guide/bgnet/html/#serialization

Exercise 3

Modify both the Server and Client programs: `server1.c`, `client1.c`

- client sends an integer n (32 bit) to the server
- server computes $f(n)$ on the received n, where $f(n) = n + 1$
- server returns computed value to the client

Exercise 3

Modify both the Server and Client programs: `server1.c`, `client1.c`

- client sends an integer n (32 bit) to the server
- server computes $f(n)$ on the received n, where $f(n) = n + 1$
- server returns computed value to the client

Solution

Take a look at: `server2.c`, `client2.c`

Serialization of Arbitrary Data Types

- An arbitrary datatype can be defined using a structure
- Apart from endianness, structures introduces padding
- These paddings are machine dependant

Serialization of Arbitrary Data Types

- An arbitrary datatype can be defined using a structure
- Apart from endianness, structures introduces padding
- These paddings are machine dependant
- On possibility is to create a large byte array and manually serialize

```
struct ABC {
    int a; char b;
};
struct ABC s = {10, 'a'};
uint8_t *buff = (uint8_t*)malloc(sizeof(uint32_t)+sizeof(char));
uint32_t a = htonl(s.a);
memcpy(buff, &a, sizeof(uint32_t));
memcpy(buff+sizeof(uint32_t), &s.b, sizeof(char));
```

Serialization of Arbitrary Data Types

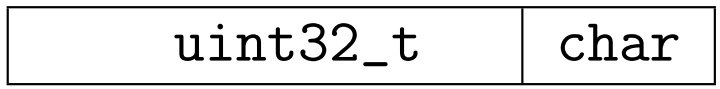
- An arbitrary datatype can be defined using a structure
- Apart from endianness, structures introduces padding
- These paddings are machine dependant
- On possibility is to create a large byte array and manually serialize

```
struct ABC {
    int a; char b;
};
struct ABC s = {10, 'a'};
uint8_t *buff = (uint8_t*)malloc(sizeof(uint32_t)+sizeof(char));
uint32_t a = htonl(s.a);
memcpy(buff, &a, sizeof(uint32_t));
memcpy(buff+sizeof(uint32_t), &s.b, sizeof(char));
```

Serialization of Arbitrary Data Types

- An arbitrary datatype can be defined using a structure
- Apart from endianness, structures introduces padding
- These paddings are machine dependant
- On possibility is to create a large byte array and manually serialize

```
struct ABC {
    int a; char b;
};
struct ABC s = {10, 'a'};
uint8_t *buff = (uint8_t*)malloc(sizeof(uint32_t)+sizeof(char));
uint32_t a = htonl(s.a);
memcpy(buff, &a, sizeof(uint32_t));
memcpy(buff+sizeof(uint32_t), &s.b, sizeof(char));
```



- Better alternative: Google Protocol Buffers[1][2]

[1] See: https://github.com/protobuf-c/protobuf-c

[2] See: https://stackoverflow.com/questions/1577161/passing-a-structure-through-sockets-in-c

Conversing with Multiple Clients

- Run different server process/thread for each new client
  See previous year's materials:
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/Socket_2.pdf#page=15 and
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/codes.php?fname=server3

Conversing with Multiple Clients

- Run different server process/thread for each new client
  See previous year's materials:
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/Socket_2.pdf#page=15 and
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/codes.php?fname=server3

- A better alternative is use `poll()` or `select()` APIs
  See previous year's materials:
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/Socket_3.pdf#page=9 and
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/codes.php?fname=server4

Conversing with Multiple Clients

- Run different server process/thread for each new client
  See previous year's materials:
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/Socket_2.pdf#page=15 and
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/codes.php?fname=server3

- A better alternative is use `poll()` or `select()` APIs
  See previous year's materials:
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/Socket_3.pdf#page=9 and
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/codes.php?fname=server4

- Multicasting over socket
  See previous year's materials:
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/Socket_3.pdf#page=30 and
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/codes.php?fname=server5,
  https://www.isical.ac.in/~rathin_r/uploads/CN/2022/codes.php?fname=client5