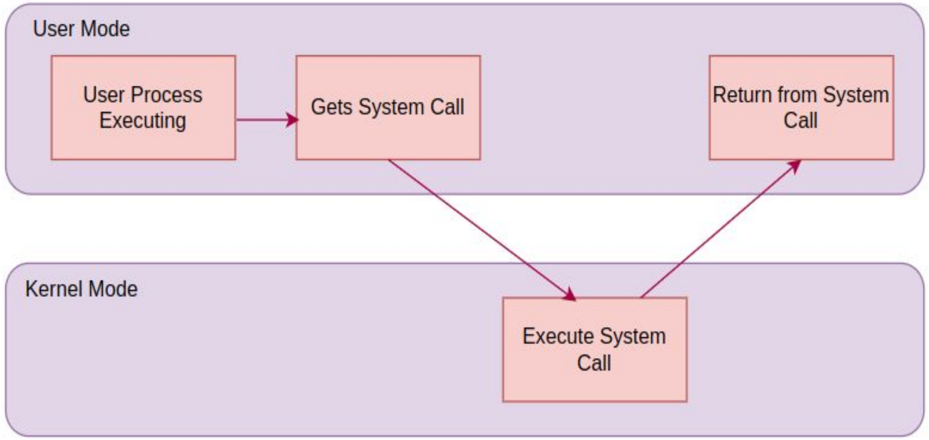
Socket Programming using blocking system calls (Server Client Architecture)

Design Lab (CS69202)

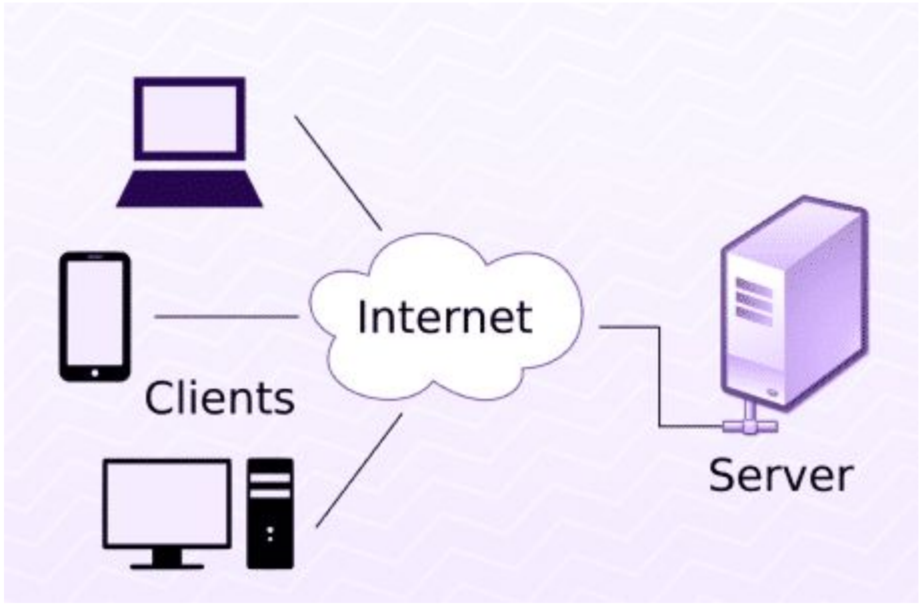
22 January, 2025

Blocking System Calls



Blocking system calls halt the execution of a program until a specific event occurs, such as receiving data or establishing a connection. This simplicity can lead to performance issues when handling many clients.

Introduction to Server Client Architecture

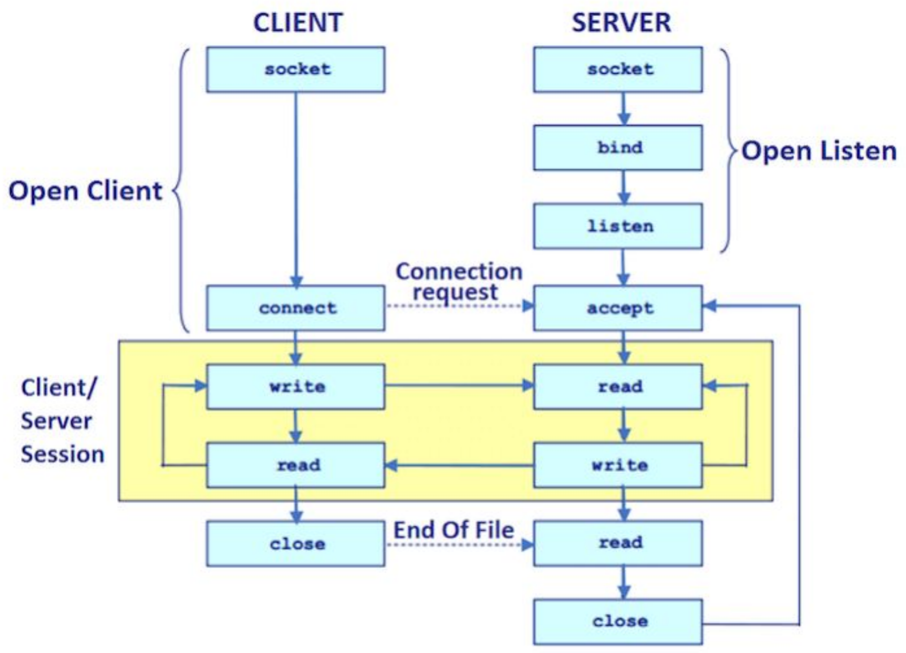


Introduction to Server Client Architecture

Server	Client
A server is a program that listens for requests from clients and responds accordingly. It acts as a centralized point of service.	A client is a program that initiates a request to a server and receives a response. Clients consume services provided by the server.

Network Programming

Network Programming : Client - Server Architecture



TCP (Transmission Control Protocol) Client-Server

- In this section we illustrate the use of sockets for inter-process communication across the network.
- We show the communication between a server process and a client process.

Network Programming : Server side

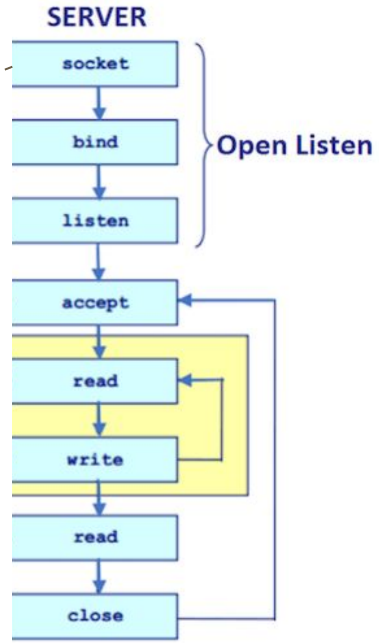
- Since many server processes may be running in a system, we identify the desired server process by a "port number". In the code snippets, we choose port number 6000 for our server process.

- To create a TCP server process, we first need to open a "socket" using the socket( ) system call.

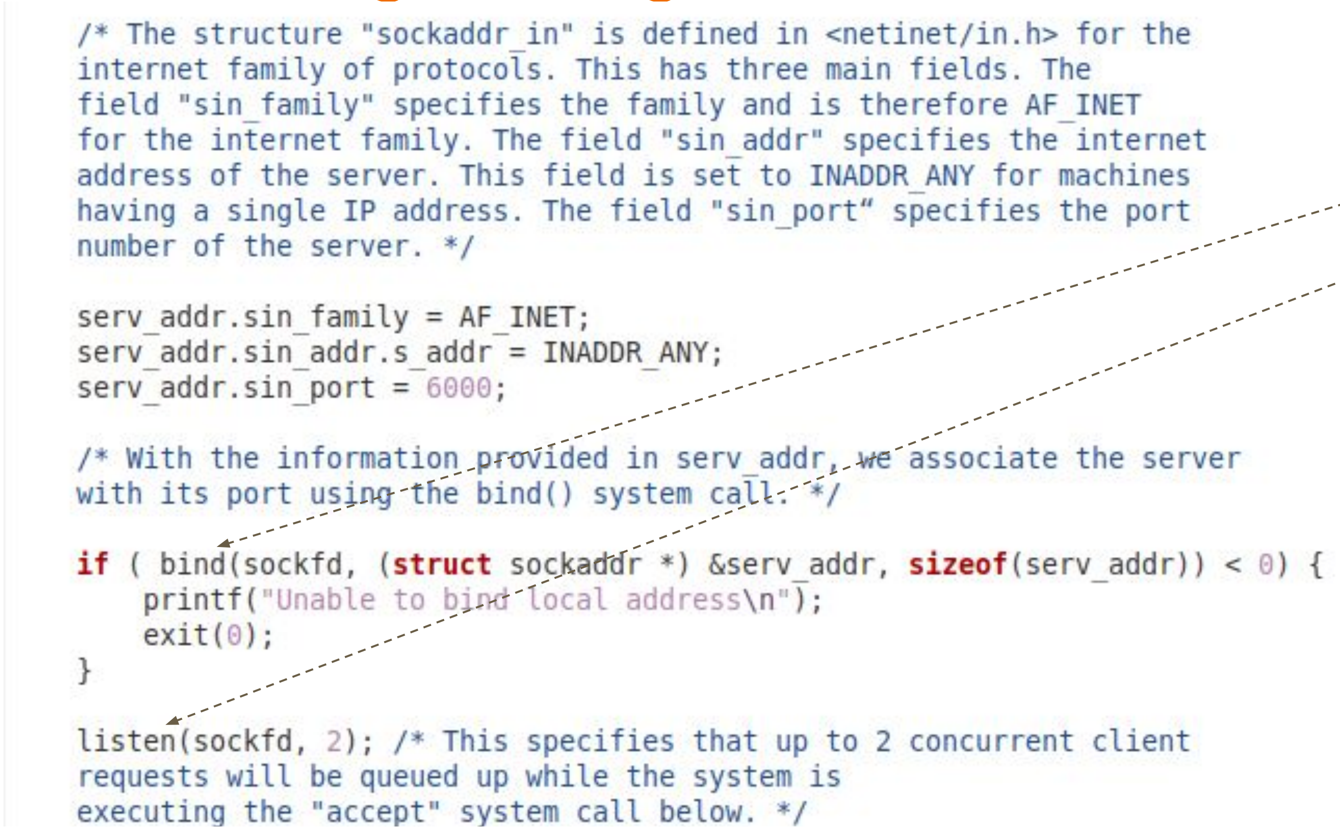
- The following three files must be included for network programming -> <sys/socket.h>, <netinet/in.h> and <arpa/inet.h>

Network Programming : Server side

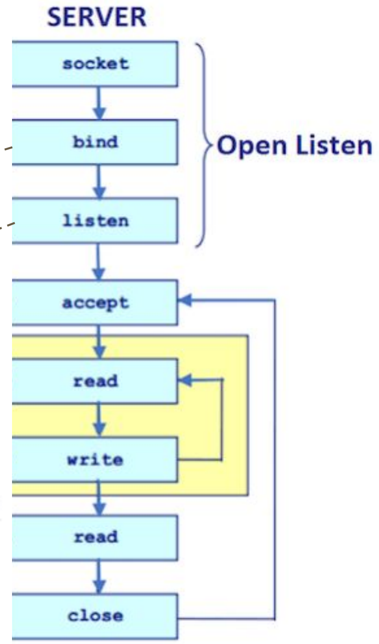
```
main( )
{
    int sockfd, newsockfd; /* Socket descriptors */
    int clilen;
    struct sockaddr_in cli_addr, serv_addr;
    int i;
    char buf[100]; /* A buffer for communication */

    /* The following system call opens a socket. The first parameter
    indicates the family of the protocol to be followed. For internet
    protocols we use AF_INET. For TCP sockets the second
    parameter is SOCK_STREAM. The third parameter is set to 0
    for user applications. */

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0) {
        printf("Cannot create socket\n");
        exit(0);
    }
```



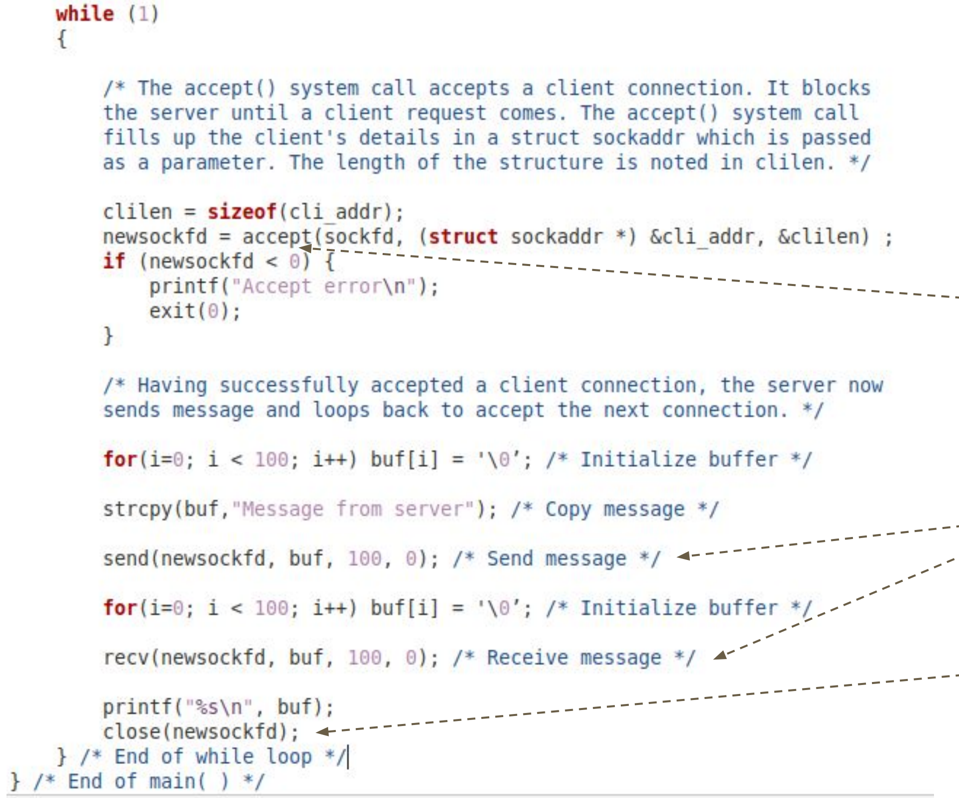
Network Programming : Server side



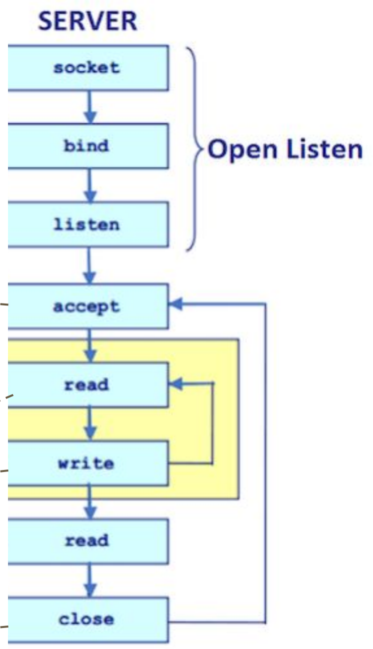
```
/* The structure "sockaddr_in" is defined in <netinet/in.h> for the
internet family of protocols. This has three main fields. The
field "sin_family" specifies the family and is therefore AF_INET
for the internet family. The field "sin_addr" specifies the internet
address of the server. This field is set to INADDR_ANY for machines
having a single IP address. The field "sin_port" specifies the port
number of the server. */

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = 6000;

/* With the information provided in serv_addr, we associate the server
with its port using the bind() system call. */

if ( bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
    printf("Unable to bind local address\n");
    exit(0);
}

listen(sockfd, 2); /* This specifies that up to 2 concurrent client
requests will be queued up while the system is
executing the "accept" system call below. */
```



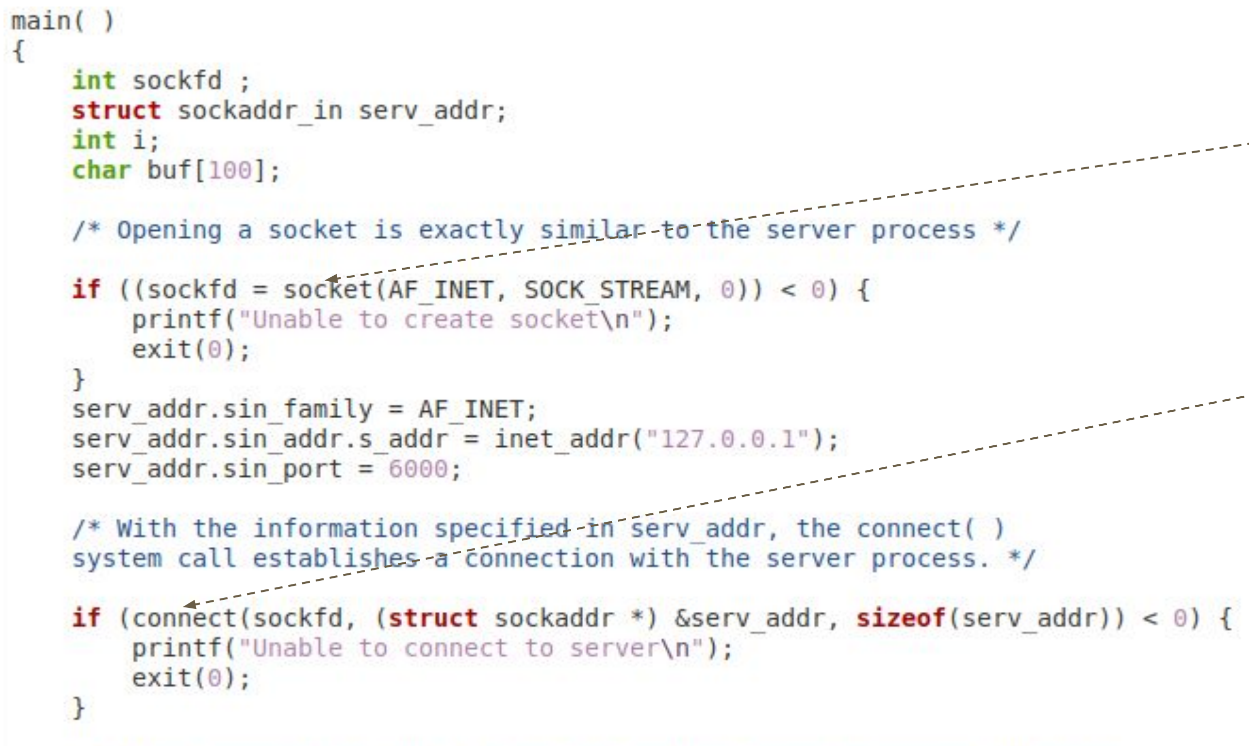
Network Programming : Server side



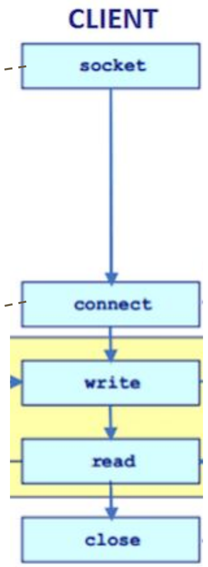
```
    while (1)
    {

        /* The accept() system call accepts a client connection. It blocks
        the server until a client request comes. The accept() system call
        fills up the client's details in a struct sockaddr which is passed
        as a parameter. The length of the structure is noted in clilen. */

        clilen = sizeof(cli_addr);
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen) ;
        if (newsockfd < 0) {
            printf("Accept error\n");
            exit(0);
        }

        /* Having successfully accepted a client connection, the server now
        sends message and loops back to accept the next connection. */

        for(i=0; i < 100; i++) buf[i] = '\0'; /* Initialize buffer */

        strcpy(buf,"Message from server"); /* Copy message */

        send(newsockfd, buf, 100, 0); /* Send message */

        for(i=0; i < 100; i++) buf[i] = '\0'; /* Initialize buffer */

        recv(newsockfd, buf, 100, 0); /* Receive message */

        printf("%s\n", buf);
        close(newsockfd);
    } /* End of while loop */
} /* End of main( ) */
```



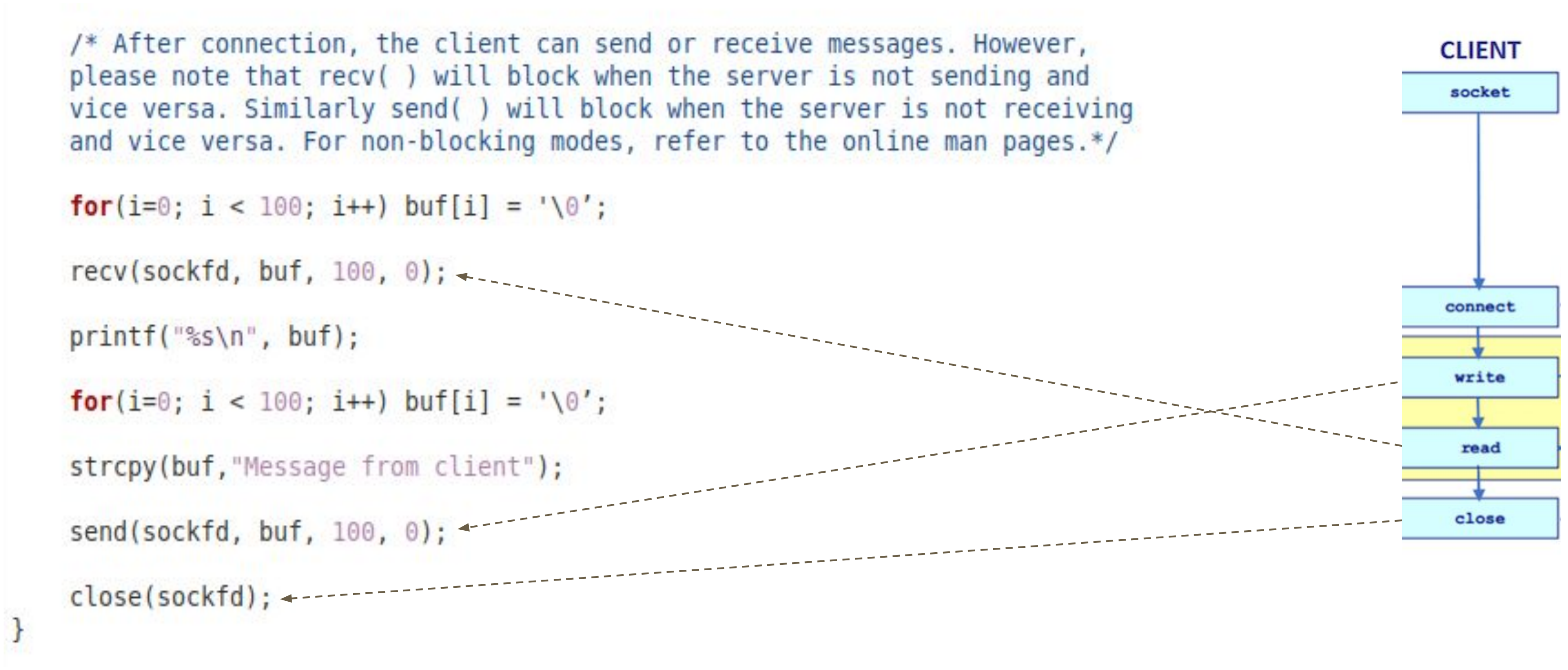
Network Programming : Client side



```
main( )
{
    int sockfd ;
    struct sockaddr_in serv_addr;
    int i;
    char buf[100];

    /* Opening a socket is exactly similar to the server process */

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("Unable to create socket\n");
        exit(0);
    }
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    serv_addr.sin_port = 6000;

    /* With the information specified in serv_addr, the connect( )
    system call establishes a connection with the server process. */

    if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
        printf("Unable to connect to server\n");
        exit(0);
    }
```

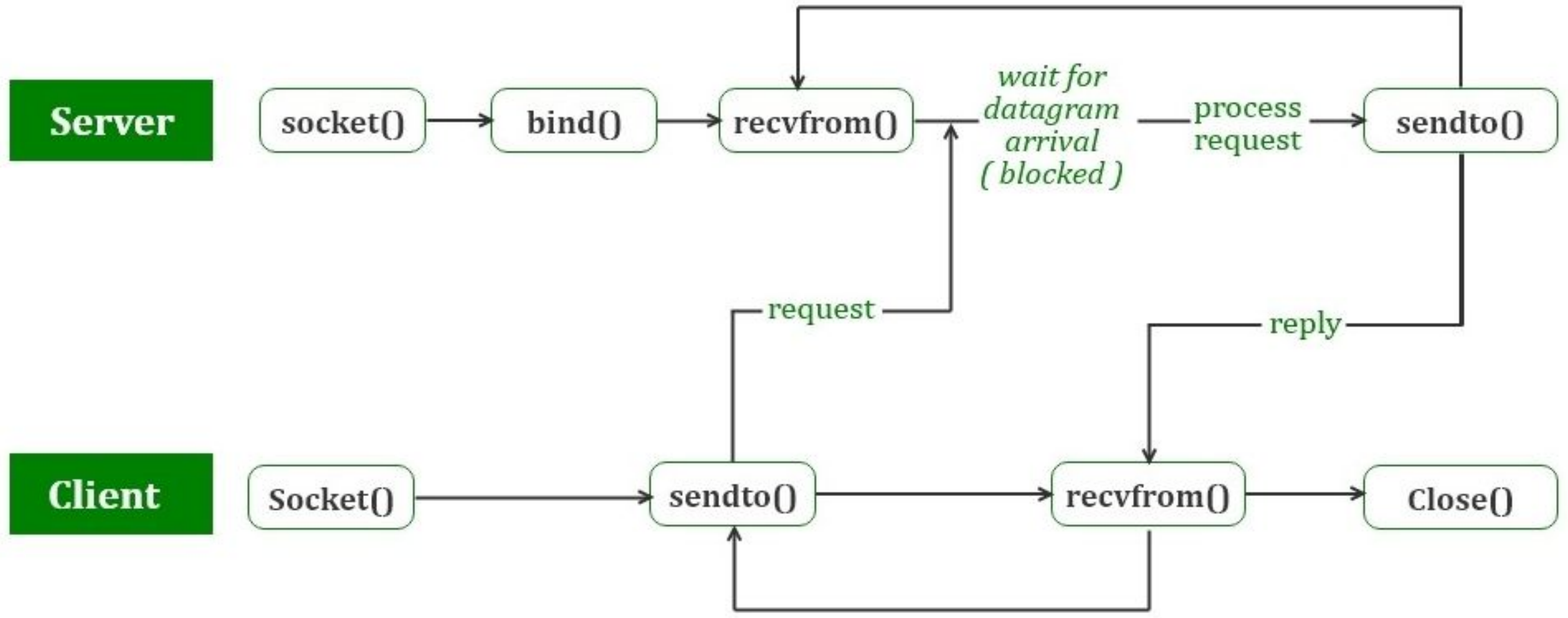


Network Programming : Client side

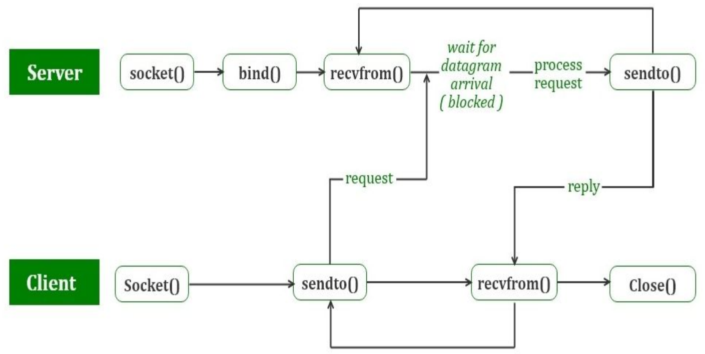


```
    /* After connection, the client can send or receive messages. However,
    please note that recv( ) will block when the server is not sending and
    vice versa. Similarly send( ) will block when the server is not receiving
    and vice versa. For non-blocking modes, refer to the online man pages.*/

    for(i=0; i < 100; i++) buf[i] = '\0';

    recv(sockfd, buf, 100, 0);

    printf("%s\n", buf);

    for(i=0; i < 100; i++) buf[i] = '\0';

    strcpy(buf,"Message from client");

    send(sockfd, buf, 100, 0);

    close(sockfd);
}
```

UDP (User Datagram Protocol) Sockets



UDP (User Datagram Protocol) Sockets



TCP is a **connection-oriented** protocol, where as UDP is a **connection-less** protocol.

- Does not have any connect( ) from client and any accept( ) from server (because no connection is established)

- Instead each message sent using sendto( ) must specify the destination IP address.

References

- Beej's Guide to Network Programming (http://www.cs.columbia.edu/~danr/courses/6761/Fall00/hw/pa1/6761-sockhelp.pdf)
- Tutorial - https://nikhilroxtomar.medium.com/tcp-client-server-implementation-in-c-idiot-developer-52509a6c1f59

it's not finished yet !!!

From Bytes to Objects

- TCP/UDP transport raw bytes over the network
- What if we want to send/receive objects?
- Also, TCP does not preserve message boundaries
  - Can’t differentiate where one object starts and where other ends
  - One recv() may return half a message or multiple messages stuck together
- So we need:
  - **Framing:** where does one message end?
  - **Encoding:** how do we represent fields?

Serialization

- Serialization is way we represent fields as bytes
- TCP forces us to think about how to know how many bytes belong to one message
- So every real protocol has a “message boundary” strategy
- Common boundary strategies:
  - Length-prefix: [len][payload] (most common for binary protocols)
  - Delimiter: **\n** or **\r\n\r\n** (common in text protocols like HTTP headers)
  - Fixed-size: message size is constant (rare but simple)
- Since a message can arrive in chunks we need to accumulate it until we get the full message

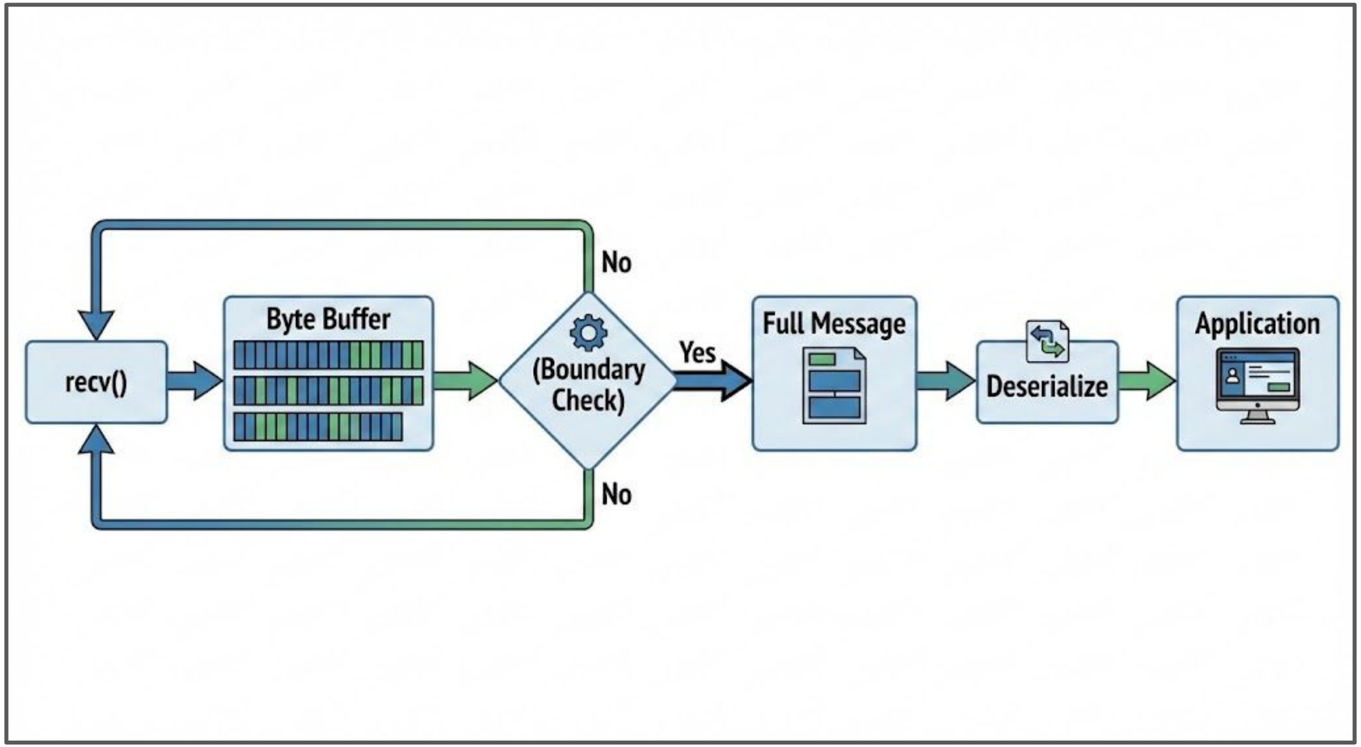
Ways to send Objects

- **Custom format:** We define the exact byte layout (e.g., length + type + fields) and pack/unpack it ourselves
  - Advantages: We get full control over size and speed
  - Disadvantages: More engineering effort and custom encode/decode logic
- **JSON:** Human-readable, easy to log/inspect, and very flexible
  - Advantages: Best for quick prototyping, debugging, and interoperability
  - Disadvantages: Larger payloads + slower parse/serialize compared to binary formats

Ways to send Objects (Continued)

- **Protobuf / FlatBuffers**
  - **Protobuf:** Schema-first format (.proto) with code generation. Messages are serialized bytes that are typically parsed into objects before use
    - Protobuf is very widely adopted, does compact encoding and fast serialization-deserialization
    - It usually requires a parse step (CPU + possible allocations)
  - **Flatbuffers:** Schema-first format that stores data so it can be **read directly** from the byte buffer (often called zero-copy/no-unpack access)
    - Flatbuffers provide very fast reads as we do not need full deserialization
    - On the other hand, message construction in flatbuffers can be more complex

Serialization Flow

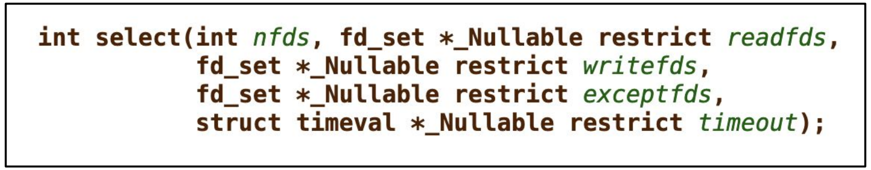


Event Loops: Handling Many Sockets Efficiently

- In the blocking model, **accept()** waits for a client, and **recv()** waits for data
- One slow client can stall progress if we read → process → write sequentially
- We have three concurrency strategies:
    - Process/thread per connection
    - Thread pool
    - Single (or few) threads + event loop (I/O multiplexing)
- In event loop the intuition is:
    - Do not keep blocking on one socket
    - The thread sleeps and when something is ready, the OS will wake it
    - In this way we can process many tasks concurrently

System calls / APIs

- `select()`:



```
int select(int nfds, fd_set *_Nullable restrict readfds,
           fd_set *_Nullable restrict writefds,
           fd_set *_Nullable restrict exceptfds,
           struct timeval *_Nullable restrict timeout);
```

  - Allows a program to monitor multiple file descriptors (FDs)
  - It waits until one or more of the FDs become "ready" for some class of I/O operation
- `poll()`:

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

  - Monitors many FDs and notifies when they become ready
  - `epoll (Linux): epoll_create1, epoll_ctl, epoll_wait`
    - `Level-triggered (default): you keep getting events while the condition holds`
    - `Edge-triggered (EPOLLET): you must read/write until EAGAIN, otherwise you may not get another event`

Non-Blocking Sockets: What & Why

- Blocking I/O: **accept**/**recv**/**send** wait until they can make progress.
- **Non-blocking I/O**: syscalls return immediately if they would block.
- In an event loop, we must never stall the whole loop on one slow socket.
- Non-blocking is the standard model for high-concurrency servers
- Rule of thumb: read/write **only** when the OS says the fd is ready

Turning on Non-Blocking (C/C++)

- Enable via fcntl: set O_NONBLOCK on the socket fd
- Works for:
  - listening socket (affects accept)
  - connected socket (affects recv/send)
- Typical pattern: set non-blocking right after socket() / after accept()

```
#include <fcntl.h>
#include <unistd.h>
#include <cerrno>

int make_non_blocking(int fd) {
   int flags = fcntl(fd, F_GETFL, 0);
   if (flags == -1)
       return -1;
   return fcntl(fd, F_SETFL, flags | O_NONBLOCK);
}
```