

TATA STEEL LIMITED



Project Report on:-

**HANDSHAKING OF PROCESS DATA
BETWEEN TWO SYSTEM USING NETWORK
PROTOCOL**

(1 June 2023-1 July 2023)

REFERENCE NO.-VT20231525

SUBMITTED BY: ANKIT KISHORE SHARAN
(KIIT,Bhubaneshwar)

GUIDED BY: Mr. Rajiv Kumar (Senior Manager, ADC,
Automation Division, TATA Steel)

ACKNOWLEDGEMENT

This work carries with it the kind support, inspiration and guidance by various people at various levels, to whom I am grateful and sincerely indebted.

To start with, I wish to record my deepest gratitude to my guide **Mr. Rajiv Kumar (Senior Manager, ADC, Automation Division, TATA Steel)** who led me inspiringly and motivated throughout, along with his scrupulous guidance at all time. His keen interest in the discussions has benefited us to the extent that cannot be spanned in words. Finally, I thank Mr Shakil Ahmed, General Manager (SNTI), for all the help and resources that were made available to me.

I am also thankful to all the respected individuals who were directly or indirectly involved in the successful completion of my technical project. Lastly, I want to thank all my friends that have given me advice and encouragement in completing my project throughout this journey. Thank you very much to all and may God bless you.

ANKIT SHARAN

ABSTRACT

The abstract of handshaking process data between two systems using a network protocol refers to the high-level summary of the procedure involved in establishing a connection, exchanging data, and confirming successful transmission between two networked systems. Handshaking, in this context, refers to the mutual agreement and synchronization between the systems to ensure reliable data transfer. The process typically involves the following steps:

1. **Connection Establishment:** The two systems initiate a connection using a network protocol such as TCP/IP. This involves the exchange of control messages to establish a reliable communication channel.
2. **Handshake Initialization:** Once the connection is established, the systems exchange specific handshaking messages to negotiate parameters, establish synchronization, and confirm the readiness to exchange data.
3. **Data Exchange:** After successful handshaking, the systems can begin the transfer of process data. This data may include various types of information, such as sensor readings, control commands, or any other relevant data for the particular application.
4. **Acknowledgment:** Upon receiving the data, the receiving system sends an acknowledgment message to the sending system, confirming the successful reception of the data. This helps ensure data integrity and provides feedback to the sender.
5. **Error Handling:** If an error occurs during data transmission or reception, error handling mechanisms defined by the network protocol are invoked. These mechanisms may include retransmission of lost or corrupted data packets to maintain the integrity of the data transfer.
6. **Termination:** Once the data exchange is complete or when either system decides to end the communication, termination messages are exchanged, and the connection is closed gracefully.

The abstract of handshaking process data between two systems using a network protocol encapsulates the overall sequence of events involved in establishing, maintaining, and terminating a reliable communication link for exchanging process data between two networked systems.

TABLE OF CONTENTS

1)Introduction.....	5
2)What is Handshaking?.....	6
3)Process Data.....	7
4)Handshaking of Process Data.....	8
5)Client Server architecture.....	9
6)Implementation of Client-Server architecture.....	10-15
7)Connection-Oriented AND Connectionless Network	
7.1)TCP/IP MODEL.....	16-20
7.2)UDP MODEL.....	20-23
8)Network Protocol.....	24
9)Data Transmission Over the Network	
9.1)PARALLEL DATA TRANSMISSION.....	25
9.2)SERIAL DATA TRANSMISSION.....	25-26
10)Conclusion.....	27

Introduction

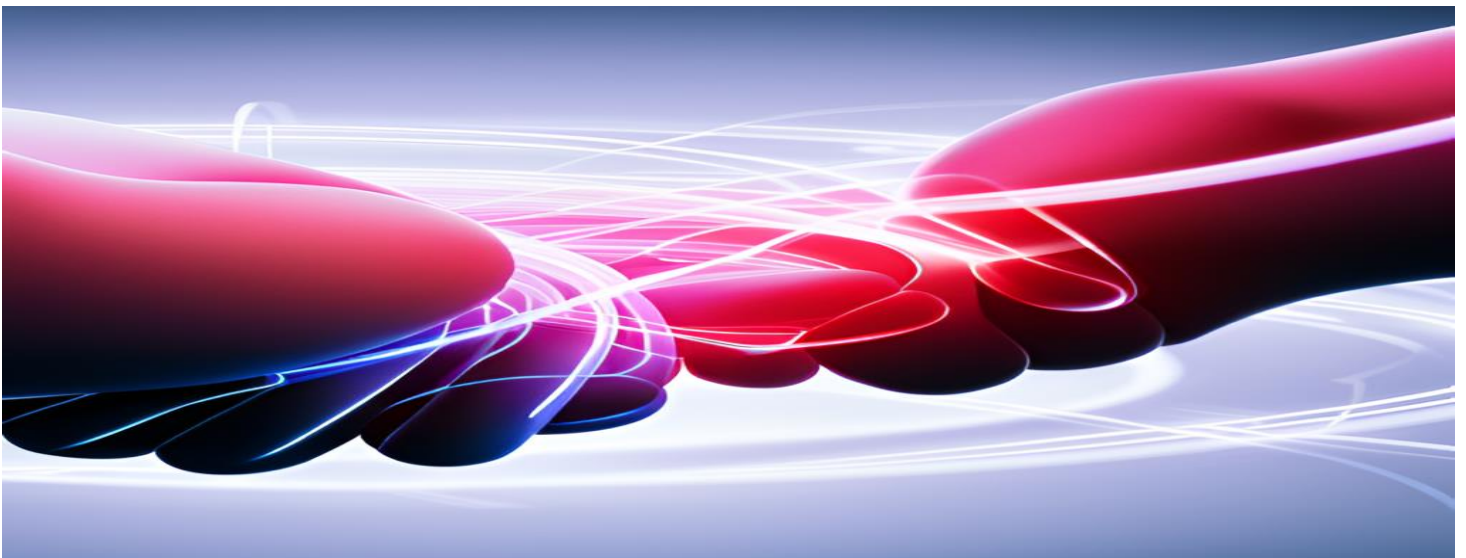
Welcome to my presentation on handshaking of process data between two systems using network protocol. In today's interconnected world, the exchange of information is critical to the functioning of many systems. Handshaking is a fundamental concept in computer networks that enables this exchange to happen smoothly and efficiently.

In this presentation, we will explore what handshaking is, the importance of process data in computer networks, the role of network protocols, and how handshaking is used to exchange process data between two systems using network protocol.

What is Handshaking?

Handshaking is a communication process used in computer networks to establish a connection between two systems. It involves a series of messages sent between the two systems to negotiate the terms of the connection and ensure that both systems are ready to communicate.

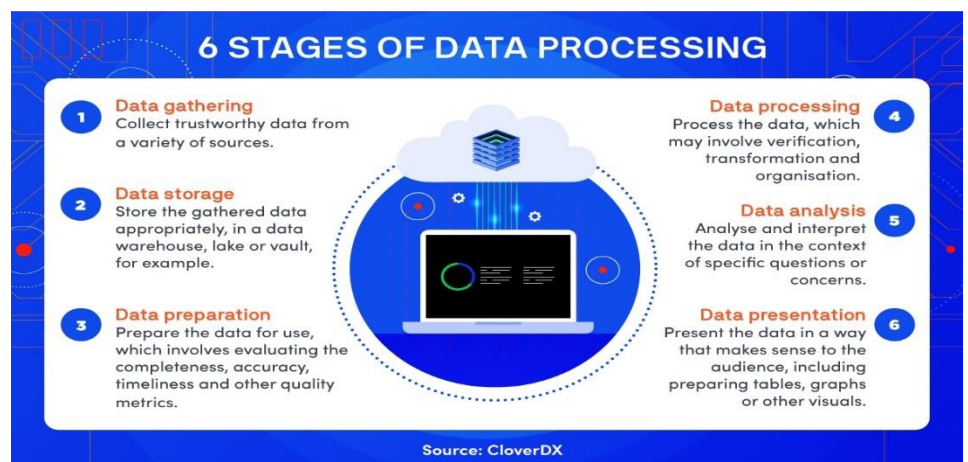
During handshaking, each system sends a message to the other indicating its readiness to communicate and the protocols it supports. The two systems then agree on the protocols they will use for the exchange of data.



Process Data

Process data refers to the information generated by a system as it carries out its tasks. This data can be used to monitor the performance of the system and make adjustments as needed. In computer networks, process data is exchanged between systems to enable them to work together effectively.

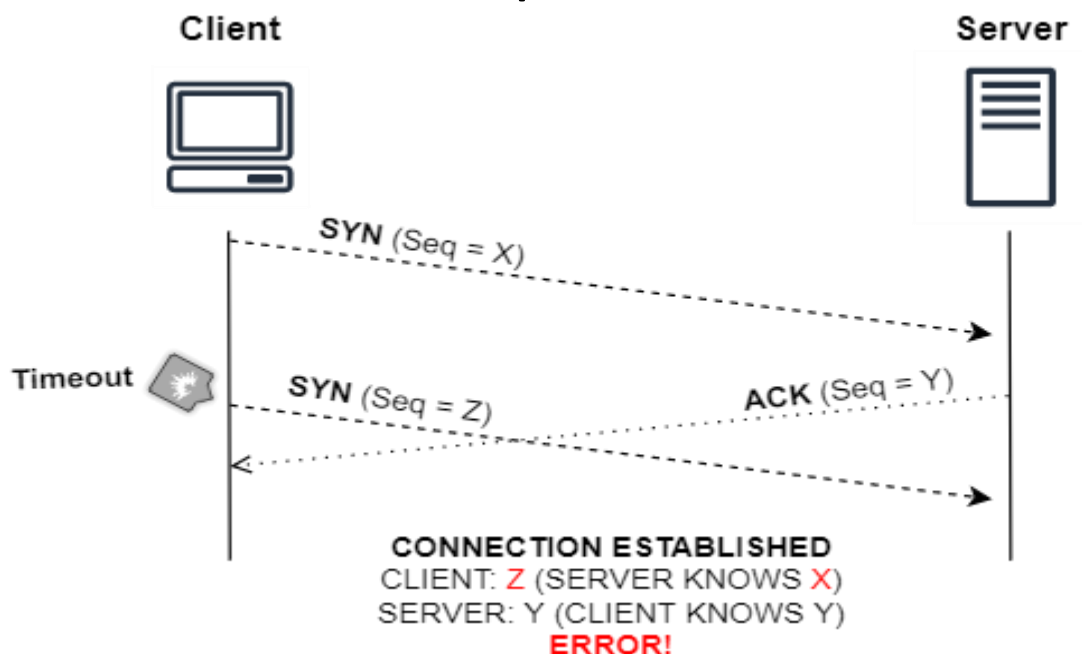
For example, in a manufacturing plant, process data might include information about the temperature, pressure, and flow rate of materials as they move through the production process. This data can be used to optimize the plant's performance and ensure it operates at peak efficiency.



Handshaking of Process Data

Handshaking is used to exchange process data between two systems using network protocol. During handshaking, the two systems negotiate the terms of the connection and agree on the protocols they will use to exchange data.

Once the connection has been established, the systems can begin to exchange process data. This data can be used to monitor the performance of the systems and make adjustments as needed to optimize their operation.

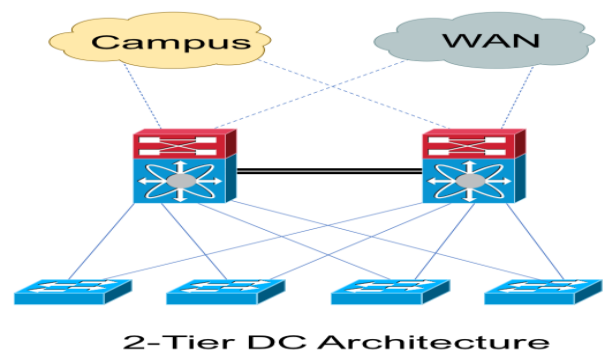
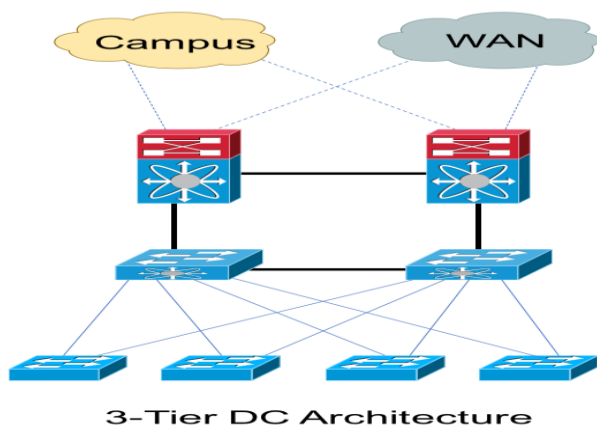


Client Server architecture

Client-server architecture is a computing model in which many clients (remote processors) request and receive service from a centralized server (host computer). There are two types of following Client server Architecture.

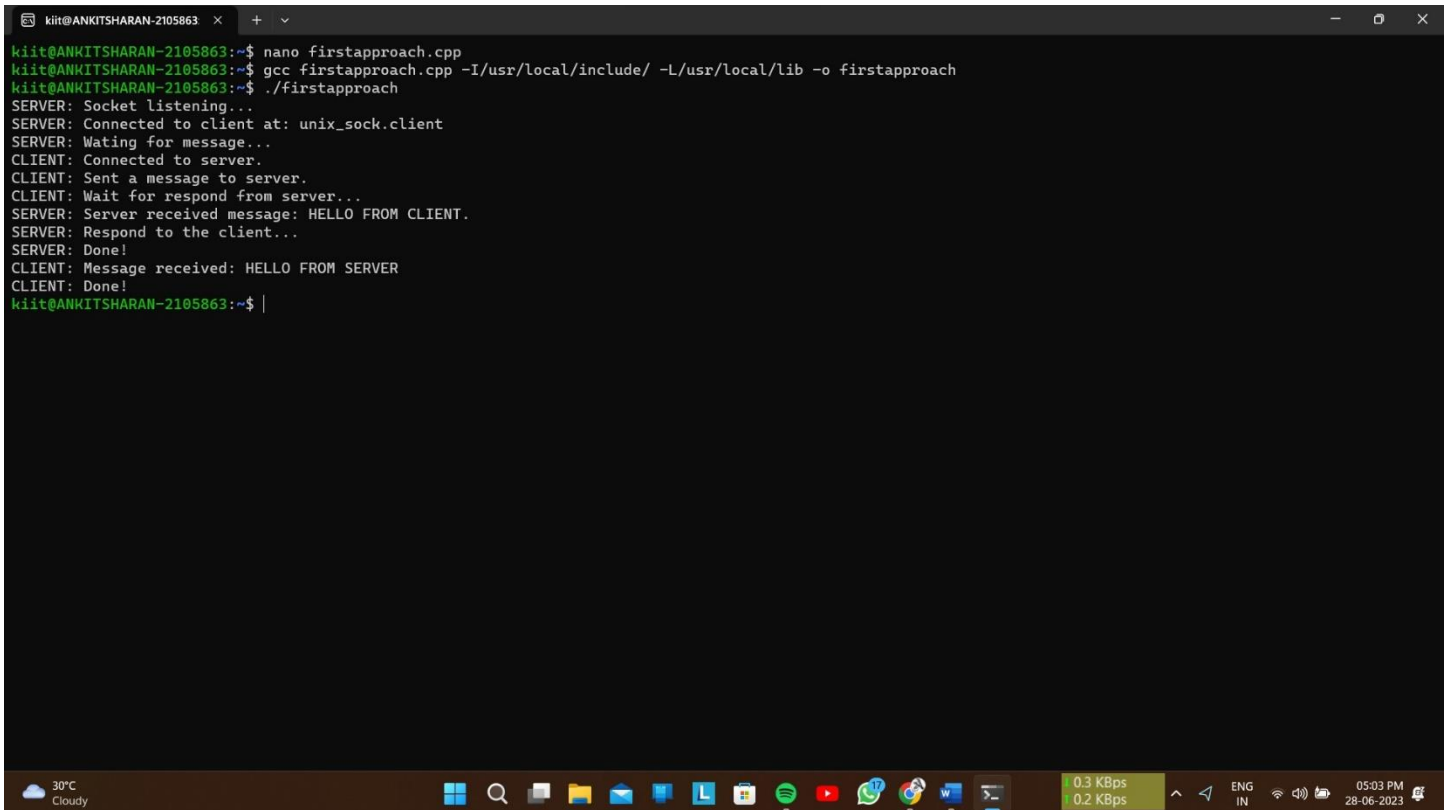
In a **two-tier** client-server architecture, the application logic is either buried inside the user interface on the client or within the database on the server (or both). With two-tier client/server architectures, the user system interface is usually located in the user's desktop environment and the database management services are usually in a server that is a more powerful machine that services many clients.

In contrast, a **three-tier** architecture introduces a middle tier between the user system interface client environment and the database management server environment. This middle tier can be implemented in a variety of ways, such as transaction processing monitors, message servers, or application servers. In three-tier architecture, the application logic or process lives in the middle-tier and is separated from the data and the user interface.



Implementation of Client Server architecture:

FIRST APPROACH



```
kiit@ANKITSHARAN-2105863:~$ nano firstapproach.cpp
kiit@ANKITSHARAN-2105863:~$ gcc firstapproach.cpp -I/usr/local/include/ -L/usr/local/lib -o firstapproach
kiit@ANKITSHARAN-2105863:~$ ./firstapproach
SERVER: Socket listening...
SERVER: Connected to client at: unix_sock.client
SERVER: Waiting for message...
CLIENT: Connected to server.
CLIENT: Sent a message to server.
CLIENT: Wait for respond from server...
SERVER: Server received message: HELLO FROM CLIENT.
SERVER: Respond to the client...
SERVER: Done!
CLIENT: Message received: HELLO FROM SERVER
CLIENT: Done!
kiit@ANKITSHARAN-2105863:~$
```

CODE:-

****/Two-way communication between process server and client*/***

```
#include <unistd.h> // for fork()

#include <stdio.h> // for printf

#include <stdlib.h> // for exit()

#include <sys/socket.h>

#include <sys/un.h> // socket in Unix

// for print error message

#include <string.h>

#include <errno.h>
```

```

#define SERVER_SOCK_PATH "unix_sock.server"

#define CLIENT_SOCK_PATH "unix_sock.client"

#define SERVER_MSG "HELLO FROM SERVER"

#define CLIENT_MSG "HELLO FROM CLIENT"

int main(int argc, char **argv)
{
    struct sockaddr_un server_addr;

    struct sockaddr_un client_addr;

    memset(&server_addr, 0, sizeof(server_addr));

    memset(&client_addr, 0, sizeof(client_addr));

    int rc;

    // for simplicity, we will assign a fixed size for buffer of the message

    char buf[256];

    // create two processes of client and server

    pid_t pid = fork();

    //----- SERVER PROCESS

    if (pid != 0)

    {

        // maximum number of client connections in queue

        int backlog = 10;

        /* Open the server socket with the SOCK_STREAM type */

        int server_sock = socket(AF_UNIX, SOCK_STREAM, 0);

        if (server_sock == -1)

        {

            printf("SERVER: Error when opening server socket.\n");

            exit(1);

        }

        /* Bind to an address on file system */

        // similar to other IPC methods, domain socket needs to bind to a file system

        // so that client know the address of the server to connect to

        server_addr.sun_family = AF_UNIX;

```

```

strcpy(server_addr.sun_path, SERVER_SOCKET_PATH);

int len = sizeof(server_addr);

// unlink the file before bind, unless it can't bind

unlink(SERVER_SOCKET_PATH);

rc = bind(server_sock, (struct sockaddr *)&server_addr, len);

if (rc == -1)
{
    printf("SERVER: Server bind error: %s\n", strerror(errno));

    close(server_sock);

    exit(1);
}

/* Listen and accept client connection */

// set the server in the "listen" mode and maximum pending connected clients in queue

rc = listen(server_sock, backlog);

if (rc == -1)
{
    printf("SERVER: Listen error: %s\n", strerror(errno));

    close(server_sock);

    exit(1);
}

printf("SERVER: Socket listening...\n");

int client_fd = accept(server_sock, (struct sockaddr *) &client_addr, (socklen_t *)&len);

if (client_fd == -1)
{
    printf("SERVER: Accept error: %s\n", strerror(errno));

    close(server_sock);

    close(client_fd);

    exit(1);
}

printf("SERVER: Connected to client at: %s\n", client_addr.sun_path);

printf("SERVER: Waiting for message...\n");

/* Listen to client */

```

```

memset(buf, 0, 256);

int byte_rcv = recv(client_fd, buf, sizeof(buf), 0);

if (byte_rcv == -1)

{

    printf("SERVER: Error when receiving message: %s\n", strerror(errno));

    close(server_sock);

    close(client_fd);

    exit(1);

}

else

    printf("SERVER: Server received message: %s.\n", buf);

/* Response to client */

printf("SERVER: Respond to the client...\n");

memset(buf, 0, 256);

strcpy(buf, SERVER_MSG);

rc = send(client_fd, buf, strlen(buf), 0);

if (rc == -1)

{

    printf("SERVER: Error when sending message to client.\n");

    close(server_sock);

    close(client_fd);

    exit(1);

}

printf("SERVER: Done!\n");

close(server_sock);

close(client_fd);

remove(SERVER_SOCKET_PATH);

}

//----- CLIENT PROCESS

else

{

    /* Open a client socket (same type as the server) */

```

```

int client_sock = socket(AF_UNIX, SOCK_STREAM, 0);

if (client_sock == -1)

{

    printf("CLIENT: Socket error: %s\n", strerror(errno));

    exit(1);

}

/* Bind client to an address on file system */

client_addr.sun_family = AF_UNIX;

strcpy(client_addr.sun_path, CLIENT_SOCK_PATH);

int len = sizeof(client_addr);

unlink (CLIENT_SOCK_PATH);

rc = bind(client_sock, (struct sockaddr *)&client_addr, len);

if (rc == -1)

{

    printf("CLIENT: Client binding error. %s\n", strerror(errno));

    close(client_sock);

    exit(1);

}

/* Set server address and connect to it */

server_addr.sun_family = AF_UNIX;

strcpy(server_addr.sun_path, SERVER_SOCK_PATH);

rc = connect(client_sock, (struct sockaddr *)&server_addr, len);

if (rc == -1)

{

    printf("CLIENT: Connect error. %s\n", strerror(errno));

    close(client_sock);

    exit(1);

}

printf("CLIENT: Connected to server.\n");

/* Send message to server */

memset(buf, 0, sizeof(buf));

strcpy(buf, CLIENT_MSG);

```

```

rc = send(client_sock, buf, sizeof(buf), 0);

if (rc == -1)

{

    printf("CLIENT: Send error. %s\n", strerror(errno));

    close(client_sock);

    exit(1);

}

printf("CLIENT: Sent a message to server.\n");

/* Listen to the response from server */

printf("CLIENT: Wait for respond from server...\n");

memset(buf, 0, sizeof(buf));

rc = recv(client_sock, buf, sizeof(buf), 0);

if (rc == -1)

{

    printf("CLIENT: Recv Error. %s\n", strerror(errno));

    close(client_sock);

    exit(1);

}

else

    printf("CLIENT: Message received: %s\n", buf);

printf("CLIENT: Done!\n");

close(client_sock);

remove(CLIENT_SOCK_PATH);

}

return 0;

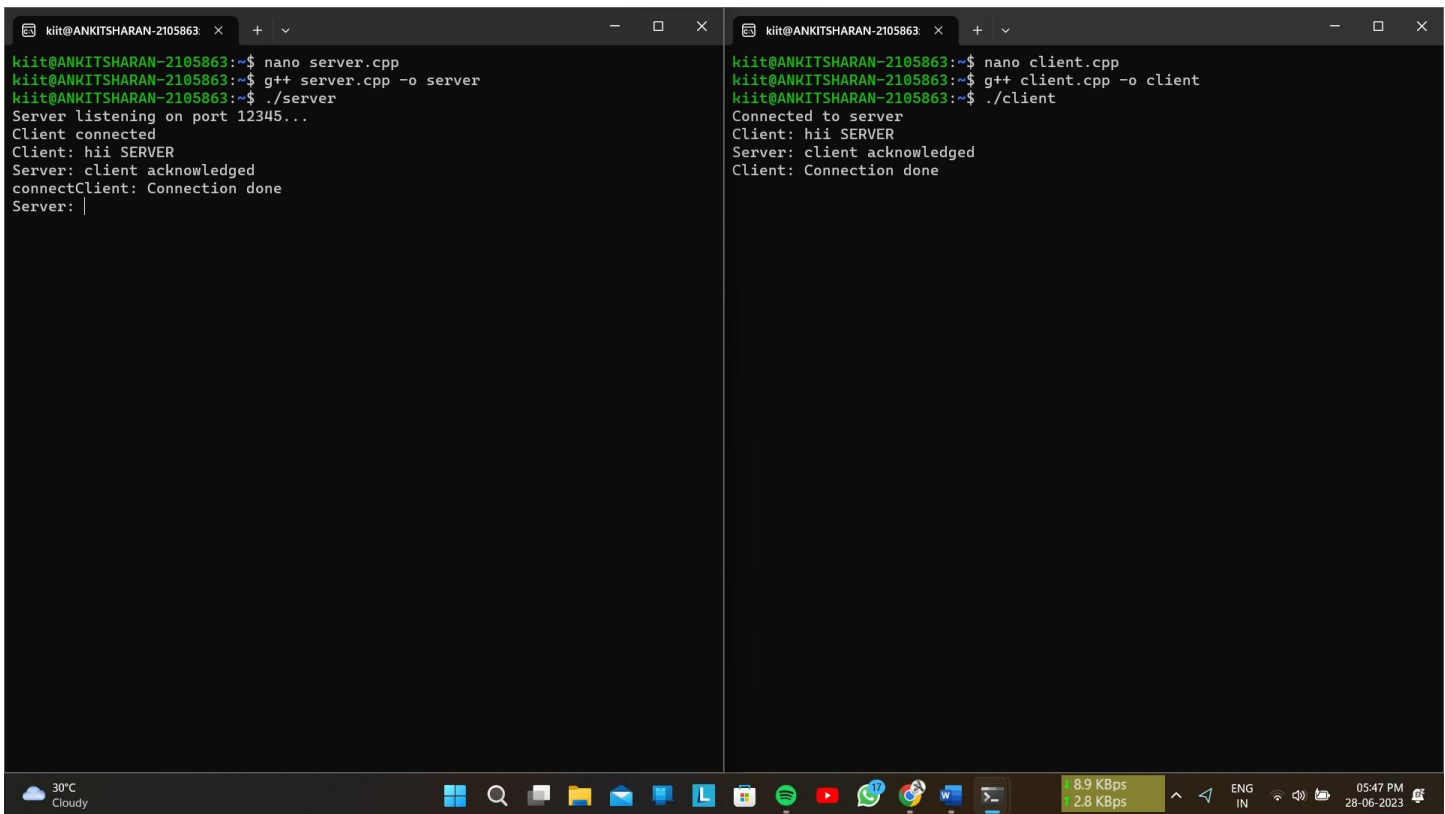
}

```

Connection Oriented AND Connectionless Network

- Connection-oriented networks require an initial setup phase to establish a connection, while connectionless networks do not.

TCP/IP MODEL



```
kiit@ANKITSHARAN-2105863:~$ nano server.cpp
kiit@ANKITSHARAN-2105863:~$ g++ server.cpp -o server
kiit@ANKITSHARAN-2105863:~$ ./server
Server listening on port 12345...
Client connected
Client: hii SERVER
Server: client acknowledged
connectClient: Connection done
Server: |

kiit@ANKITSHARAN-2105863:~$ nano client.cpp
kiit@ANKITSHARAN-2105863:~$ g++ client.cpp -o client
kiit@ANKITSHARAN-2105863:~$ ./client
Connected to server
Client: hii SERVER
Server: client acknowledged
Client: Connection done
```

CODE:-

Server-side implementation:

```
#include <iostream>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <cstring>

int main() {
```



```

// Create a socket

int serverSocket = socket(AF_INET, SOCK_STREAM, 0);

if (serverSocket == -1) {

    std::cerr << "Failed to create socket\n";

    return 1;

}

// Prepare the sockaddr_in structure

sockaddr_in serverAddress{};

serverAddress.sin_family = AF_INET;

serverAddress.sin_addr.s_addr = INADDR_ANY;

serverAddress.sin_port = htons(12345); // Port number

// Bind

if (bind(serverSocket, reinterpret_cast<struct sockaddr *>(&serverAddress), sizeof(serverAddress)) == -1) {

    std::cerr << "Failed to bind\n";

    return 1;

}

// Listen

if (listen(serverSocket, 5) == -1) {

    std::cerr << "Failed to listen\n";

    return 1;

}

std::cout << "Server listening on port 12345...\n";

// Accept incoming connections

sockaddr_in clientAddress{};

socklen_t clientAddressLength = sizeof(clientAddress);

int clientSocket = accept(serverSocket, reinterpret_cast<struct sockaddr *>(&clientAddress), &clientAddressLength);

if (clientSocket == -1) {

    std::cerr << "Failed to accept connection\n";

    return 1;

}

std::cout << "Client connected\n";

// Receive and send messages

```

```

char buffer[4096];

while (true) {

    memset(buffer, 0, sizeof(buffer));

    // Receive a message from the client

    int bytesRead = recv(clientSocket, buffer, sizeof(buffer), 0);

    if (bytesRead == -1) {

        std::cerr << "Failed to receive message\n";

        break;

    }

    if (bytesRead == 0) {

        std::cout << "Client disconnected\n";

        break;

    }

    std::cout << "Client: " << buffer << std::endl;

    // Send a message to the client

    std::cout << "Server: ";

    std::cin.getline(buffer, sizeof(buffer));

    int bytesSent = send(clientSocket, buffer, strlen(buffer), 0);

    if (bytesSent == -1) {

        std::cerr << "Failed to send message\n";

        break; }}

// Close sockets

close(clientSocket);

close(serverSocket);

return 0;}

```

Client-side implementation:

```

#include <iostream>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <cstring>

int main() {

```

```

// Create a socket

int clientSocket = socket(AF_INET, SOCK_STREAM, 0);

if (clientSocket == -1) {

    std::cerr << "Failed to create socket\n";

    return 1;

}

// Prepare the sockaddr_in structure

sockaddr_in serverAddress{};

serverAddress.sin_family = AF_INET;

serverAddress.sin_port = htons(12345); // Port number

// Convert IP address from string to binary form

if (inet_pton(AF_INET, "127.0.0.1", &serverAddress.sin_addr) <= 0) {

    std::cerr << "Invalid address\n";

    return 1;

}

// Connect to the server

if (connect(clientSocket, reinterpret_cast<const struct sockaddr *>(&serverAddress), sizeof(serverAddress)) == -1) {

    std::cerr << "Failed to connect\n";

    return 1;

}

std::cout << "Connected to server\n";

// Send and receive messages

char buffer[4096];

while (true) {

    // Send a message to the server

    std::cout << "Client: ";

    std::cin.getline(buffer, sizeof(buffer));

    int bytesSent = send(clientSocket, buffer, strlen(buffer), 0);

    if (bytesSent == -1) {

        std::cerr << "Failed to send message\n";

        break;

    }
}

```

```

if (strcmp(buffer, "exit") == 0)

    break;

memset(buffer, 0, sizeof(buffer));

// Receive a message from the server

int bytesRead = recv(clientSocket, buffer, sizeof(buffer), 0);

if (bytesRead == -1) {

    std::cerr << "Failed to receive message\n";

    break;

}

if (bytesRead == 0) {

    std::cout << "Server disconnected\n";

    break;

}

std::cout << "Server: " << buffer << std::endl;

}

// Close socket

close(clientSocket);

return 0;}

```

UDP MODEL

```

kiit@ANKITSHARAN-2105863:~$ nano udp_server.cpp
kiit@ANKITSHARAN-2105863:~$ ./server
Server listening on port 12345...
Client connected
Client: HII SERVER
Server: Client Acknowledged
Client: Connection Done
Server:

kiit@ANKITSHARAN-2105863:~$ nano udp_client.cpp
kiit@ANKITSHARAN-2105863:~$ ./client
Connected to server
Client: HII SERVER
Server: Client Acknowledged
Client: Connection Done

```

CODE:-

Server-side implementation:

```
#include <bits/stdc++.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <netinet/in.h>

#define PORT      8080

#define MAXLINE 1024

// Driver code

int main() {

    int sockfd;

    char buffer[MAXLINE];

    const char *hello = "Hello from server";

    struct sockaddr_in servaddr, cliaddr;

    // Creating socket file descriptor

    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {

        perror("socket creation failed");

        exit(EXIT_FAILURE);

    }

    memset(&servaddr, 0, sizeof(servaddr));

    memset(&cliaddr, 0, sizeof(cliaddr));

    // Filling server information

    servaddr.sin_family = AF_INET; // IPv4

    servaddr.sin_addr.s_addr = INADDR_ANY;

    servaddr.sin_port = htons(PORT);

    // Bind the socket with the server address

    if ( bind(sockfd, (const struct sockaddr *)&servaddr,
```

```

        sizeof(servaddr)) < 0 )

    {

        perror("bind failed");

        exit(EXIT_FAILURE);

    }

    socklen_t len;

int n;

    len = sizeof(cliaddr); //len is value/result

    n = recvfrom(sockfd, (char *)buffer, MAXLINE,

                  MSG_WAITALL, ( struct sockaddr *) &cliaddr,

                  &len);

    buffer[n] = '\0';

    printf("Client : %s\n", buffer);

    sendto(sockfd, (const char *)hello, strlen(hello),

            MSG_CONFIRM, (const struct sockaddr *) &cliaddr,

            len);

    std::cout<<"Hello message sent."<<std::endl;

    return 0;

}

```

Client-side implementation:

```

#include <bits/stdc++.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <netinet/in.h>

#define PORT 8080

#define MAXLINE 1024

// Driver code

int main() {

```

```

int sockfd;

char buffer[MAXLINE];

const char *hello = "Hello from client";

struct sockaddr_in servaddr;

// Creating socket file descriptor
if ( ( sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {

    perror("socket creation failed");

    exit(EXIT_FAILURE);

}

memset(&servaddr, 0, sizeof(servaddr));

// Filling server information

servaddr.sin_family = AF_INET;

servaddr.sin_port = htons(PORT);

servaddr.sin_addr.s_addr = INADDR_ANY;

int n;

socklen_t len;

sendto(sockfd, (const char *)hello, strlen(hello),

        MSG_CONFIRM, (const struct sockaddr *) &servaddr,

        sizeof(servaddr));

std::cout<<"Hello message sent."<<std::endl;


n = recvfrom(sockfd, (char *)buffer, MAXLINE,

              MSG_WAITALL, (struct sockaddr *) &servaddr,

              &len);

buffer[n] = '\0';

std::cout<<"Server : "<<buffer<<std::endl;

close(sockfd);

return 0;

}

```

Network Protocol

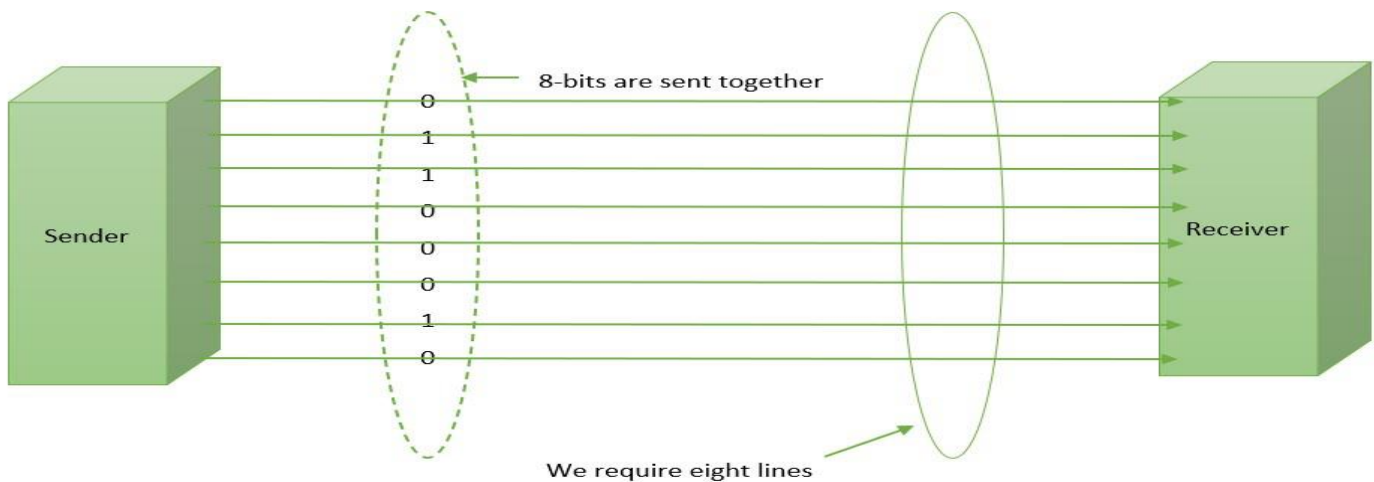
A network protocol is a set of rules and standards that govern the communication between two or more systems on a network. Protocols define how data is transmitted, received, and processed by the systems involved. There are many different network protocols, each designed for a specific purpose. Some common examples include TCP/IP, HTTP, and FTP. These protocols enable systems to communicate with each other over a network, regardless of the hardware or software they are using.



Data Transmission Over the Network

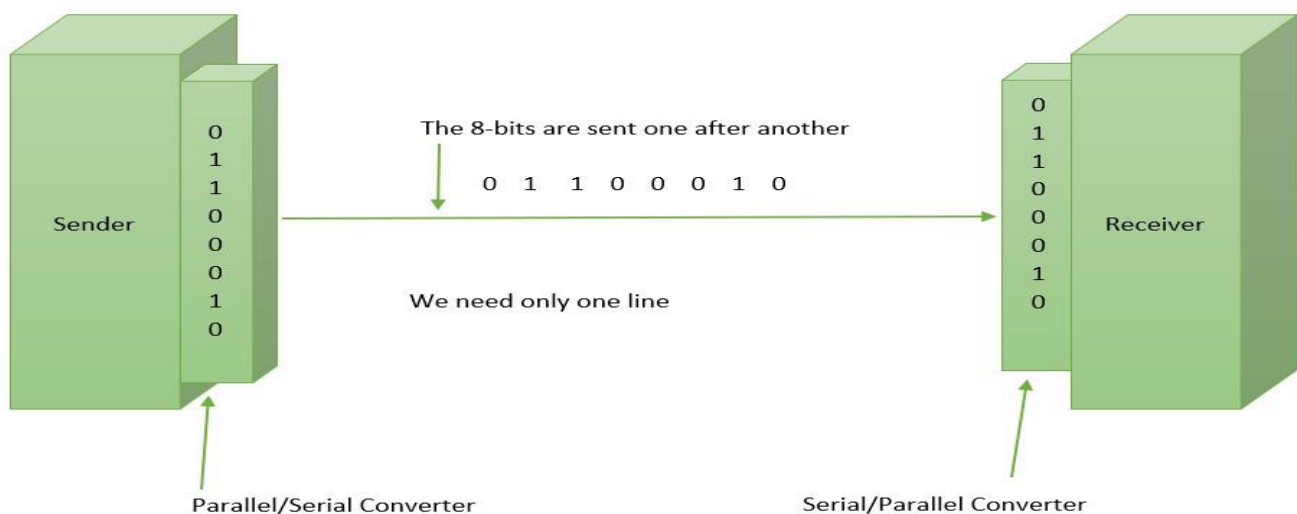
PARALLEL DATA TRANSMISSION

In parallel transmission, multiple bits of data are transmitted simultaneously over separate channels. Each channel carries one bit of the data, and all the bits together form a byte or a word of data. It allows for increased data transfer rates and improved overall network performance.



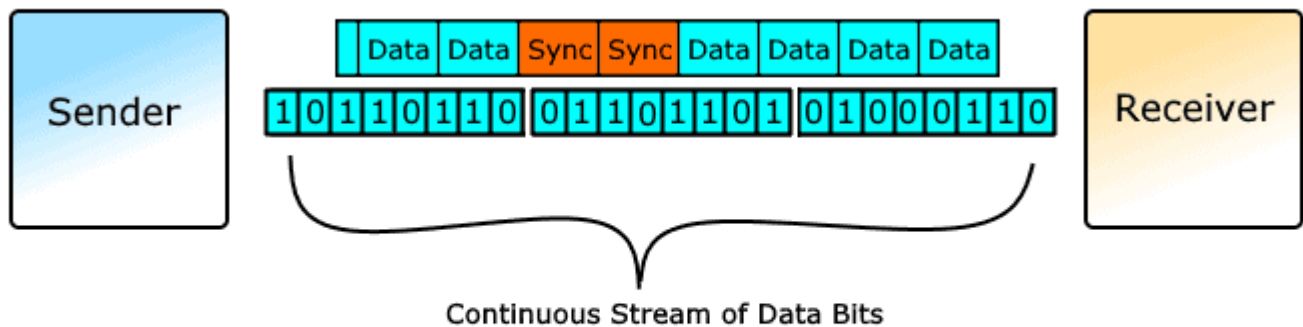
SERIAL DATA TRANSMISSION

Serial data transmission is a method of sending data sequentially, one bit at a time, over a communication channel or a physical connection. In this data is converted into a stream of individual bits, and these bits are then sent one after another over the communication medium. The receiving end interprets the stream of bits and reconstructs the original data.



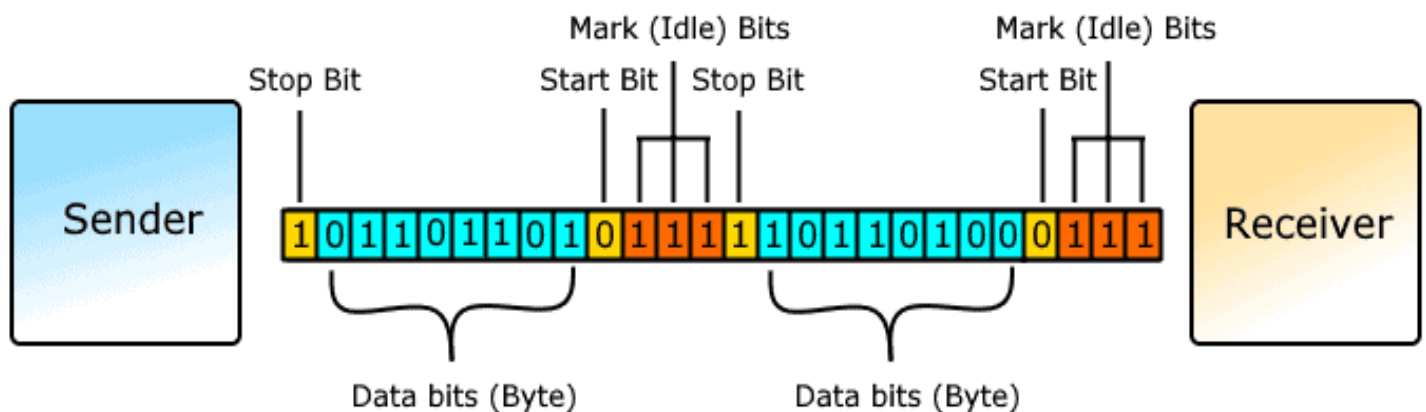
Serial data transmission is of two types:-

A) Synchronous transmission: In this data is transmitted in a continuous stream of bits, and the sender and receiver are synchronized using a common clock signal. The clock signal ensures that both ends know when to send or receive data.



Synchronous Transmission

B) Asynchronous transmission: In asynchronous transmission, data is transmitted in the form of individual characters or bytes, with each character self-contained and self-timed. There is no continuous clock signal shared between sender and receiver.



Asynchronous Transmission

Conclusion

In conclusion, handshaking of process data between two systems using network protocol is a critical concept in computer networks. It enables systems to communicate with each other efficiently and effectively, exchanging process data that can be used to optimize their performance.

As we continue to rely increasingly on interconnected systems, the importance of handshaking will only continue to grow. We hope this presentation has given you a better understanding of this fundamental concept and its role in enabling the exchange of information between systems.