# SOCKET PROGRAMMING

**Experiment No : 13 DATE:06-05-2024**

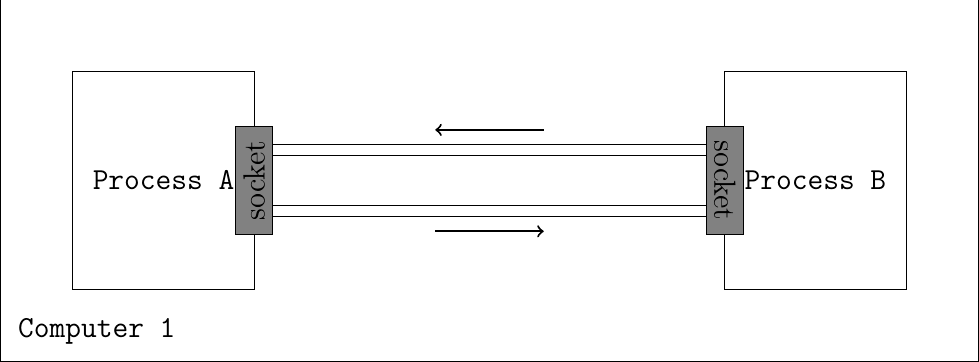
**Aim**: Implement inter process communication in C using Socket Programming.

Theory:

# Inter-process communication (IPC) using socket programming in C involves creating communication channels between processes running on the same machine or different machines over a network. Sockets provide a flexible and powerful mechanism for communication, allowing processes to exchange data bidirectionally.

# Basic overview:-

1. **Creating Sockets**: The first step is to create sockets for communication. This involves creating a socket descriptor using the socket() system call. You specify the communication domain (e.g., AF\_INET for IPv4), the type of socket (e.g., SOCK\_STREAM for TCP or SOCK\_DGRAM for UDP), and the protocol (usually 0 for the default protocol for the selected domain and type).
2. **Binding Sockets**: If the process is going to act as a server, it needs to bind the socket to a specific address and port using the bind() system call. This allows clients to connect to the server at the specified address and port.
3. **Listening for Connections (Server)**: If the process is a server using TCP sockets, it typically calls listen() after binding to start listening for incoming connections. This sets up a queue for pending connections.
4. **Accepting Connections (Server)**: When a client attempts to connect to the server, the server process accepts the connection using the accept() system call. This creates a new socket dedicated to communicating with the specific client.
5. **Connecting to a Server (Client)**: If the process is a client, it establishes a connection to the server using the connect() system call. It specifies the address and port of the server it wants to connect to.
6. **Sending and Receiving Data**: Once the connection is established (for TCP) or the socket is created (for UDP), processes can use functions like send(), recv(), sendto(), and recvfrom() to send and receive data over the socket. For TCP sockets, data is transmitted reliably and in order. For UDP sockets, data is sent as discrete packets, and there is no guarantee of delivery or order.
7. **Closing Sockets**: After communication is complete, processes should close the sockets using the close() system call to release system resources.



**CODE:**

/ /Server side

#include <netinet/in.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <unistd.h>

#include <asm-generic/socket.h>

#define PORT 8080

int main(int argc, char const \*argv[])

{

int server\_fd, new\_socket;

ssize\_t valread;

struct sockaddr\_in address;

int opt = 1;

socklen\_t addrlen = sizeof(address);

char buffer[1024] = {0};

char \*msg = "There's no place like 127.0.0.1";

// Creating socket file descriptor

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

perror("socket failed");

exit(EXIT\_FAILURE);

}

// Forcefully attaching socket to the port 8080

if (setsockopt(server\_fd, SOL\_SOCKET,

SO\_REUSEADDR | SO\_REUSEPORT, &opt,

sizeof(opt)))

{

perror("setsockopt");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

// Forcefully attaching socket to the port 8080

if (bind(server\_fd, (struct sockaddr \*)&address,

sizeof(address)) < 0)

{

perror("bind failed");

exit(EXIT\_FAILURE);

}

if (listen(server\_fd, 3) < 0)

{

perror("listen");

exit(EXIT\_FAILURE);

}

if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address,

&addrlen)) < 0)

{

perror("accept");

exit(EXIT\_FAILURE);

}

valread = read(new\_socket, buffer,

1024 - 1); // subtract 1 for the null

// terminator at the end

if (!strcmp(buffer, "Hello I am the client."))

printf("Connected to client.\n");

send(new\_socket, msg, strlen(msg), 0);

printf("Data : %s\n", msg);

printf("Data sent to client.\n");

// closing the connected socket

close(new\_socket);

// closing the listening socket

close(server\_fd);

return 0;

}

// Client side

#include <arpa/inet.h>

#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <unistd.h>

#define PORT 8080

int main(int argc, char const \*argv[])

{

int status, valread, client\_fd;

struct sockaddr\_in serv\_addr;

char \*hello = "Hello I am the client.";

char buffer[1024] = {0};

if ((client\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

printf("\n Socket creation error \n");

return -1;

}

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

// Convert IPv4 and IPv6 addresses from text to binary

// form

if (inet\_pton(AF\_INET, "172.26.10.10", &serv\_addr.sin\_addr) <= 0)

{

printf(

"\nInvalid address/ Address not supported \n");

return -1;

}

if ((status = connect(client\_fd, (struct sockaddr \*)&serv\_addr,

sizeof(serv\_addr))) < 0)

{

printf("\nConnection Failed \n");

return -1;

}

send(client\_fd, hello, strlen(hello), 0);

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

valread = read(client\_fd, buffer,

1024 - 1);

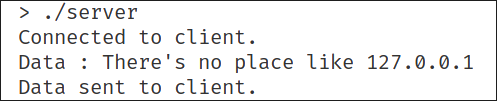
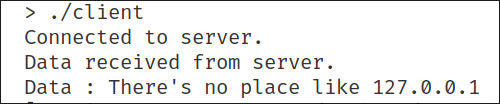
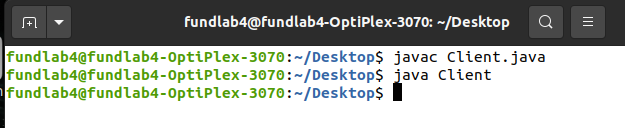
printf("Data received from server.\n");

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

// closing the connected socket

close(client\_fd);

return 0

****

**Conclusion:**

Inter process communication was successfully implemented in C using Socket Programming.