

the domain **AF\_INET** is used. The next field type has the value **SOCK\_DGRAM**. It supports datagrams (connectionless, unreliable messages of a fixed maximum length). The protocol field specifies the protocol used. We always use 0. If the socket function call is successful, a socket descriptor is returned. Otherwise -1 is returned. The header files necessary for this function call are `sys/types.h` and `sys/socket.h`.

## 2. Filling the fields of the server address structure.

The socket address structure is of type `struct sockaddr_in`.

```
struct sockaddr_in {  
  
    u_short sin_family;  
    u_short sin_port;  
    struct in_addr sin_addr;  
    char sin_zero[8]; /*unused, always zero*/  
};  
struct in_addr {  
  
    u_long s_addr;  
  
};
```

The fields of the socket address structure are

**sin\_family** which in our case is **AF\_INET**

**sin\_port** which is the port number where socket binds

**sin\_addr** is used to store the IP address of the server machine and is of type `struct in_addr`

The header file that is to be used is **netinet/in.h**

The value for `servaddr.sin_addr` is assigned using the following function

```
inet_pton(AF_INET, "IP_Address", &servaddr.sin_addr);
```

The binary value of the dotted decimal IP address is stored in the field when the function returns.

## 3. Binding of a port to the socket in the case of server

This call is used to specify for a socket the protocol port number where it will wait for messages. A call to bind is optional in the case of client and compulsory on the server side.

**int bind(int sd, struct sockaddr\* addr, int addrlen);**

The first field is the socket descriptor. The second is a pointer to the address structure of this socket. The third field is the length in bytes of the size of the structure referenced by **addr**. The header files are **sys/types.h** and **sys/socket.h**. This function call returns an integer, which is 0 for success and -1 for failure.

#### 4. Receiving data

**ssize\_t recvfrom(int s, void \* buf, size\_t len, int flags, struct sockaddr \* from, socklen\_t \* fromlen);**

The **recvfrom** calls are used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection oriented. The first parameter **s** is the socket descriptor to read from. The second parameter **buf** is the buffer to read information into. The third parameter **len** is the maximum length of the buffer. The fourth parameter is flag. It is set to zero. The fifth parameter **from** is a pointer to **struct sockaddr** variable that will be filled with the IP address and port of the originating machine. The sixth parameter **fromlen** is a pointer to a **local int** variable that should be initialized to **sizeof(struct sockaddr)**. When the function returns, the integer variable that **fromlen** points to will contain the actual number of bytes that is contained in the socket address structure. The header files required are **sys/types.h** and **sys/socket.h**. When the function returns, the number of bytes received is returned or -1 if there is an error.

#### 5. Sending data

**sendto-** sends a message from a socket

**ssize\_t sendto(int s, const void \* buf, size\_t len, int flags, const struct sockaddr \* to, socklen\_t tolen);**

The first parameter **s** is the socket descriptor of the sending socket. The second parameter **buf** is the array which stores data that is to be sent. The third parameter **len** is the length of that data in bytes. The fourth parameter is the flag parameter. It is set to zero. The fifth parameter **to** points to a variable that contains the destination IP address and port. The sixth parameter **tolen** is set to **sizeof(struct sockaddr)**. This function returns the number of bytes actually sent or -1 on error. The header files used are **sys/types.h** and **sys/socket.h**.

## Algorithm

### Client

1. Create socket
2. Read the matrices from the standard input and send it to server using socket
3. Read product matrix from the socket and display it on the standard output
4. Close the socket

### Server

1. Create socket
2. bind IP address and port number to the socket
3. Read the matrices socket from the client using socket
4. Find product of matrices
5. Send the product matrix to the client using socket
6. close the socket

### Client program

```
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<fcntl.h>
#include<stdlib.h>
main(int argc, char * argv[])
{
    int i,j,n;
    int sock_fd;
    struct sockaddr_in servaddr;
    int matrix_1[10][10], matrix_2[10][10], matrix_product[10][10];
    int size[2][2];
    int num_rows_1, num_cols_1, num_rows_2, num_cols_2;
    if(argc != 3)
    {
        fprintf(stderr, "Usage: ./client IPAddress_of_server port\n");
        exit(1);
    }
    printf("Enter the number of rows of first matrix\n");
```

```
scanf("%d", &num_rows_1);
printf("Enter the number of columns of first matrix\n");
scanf("%d", &num_cols_1);
printf("Enter the values row by row one on each line\n" );
for ( i = 0; i < num_rows_1; i++)
for( j=0; j<num_cols_1; j++)
{
scanf("%d", &matrix_1[i][j]);
}
size[0][0] = num_rows_1;
size[0][1] = num_cols_1;
printf("Enter the number of rows of second matrix\n");
scanf("%d", &num_rows_2);
printf("Enter the number of columns of second matrix\n");
scanf("%d", &num_cols_2);
if( num_cols_1 != num_rows_2)
{
printf("MATRICES CANNOT BE MULTIPLIED\n");
exit(1);
}
printf("Enter the values row by row one on each line\n");
for (i = 0; i < num_rows_2; i++)
for(j=0; j<num_cols_2; j++)
{
scanf("%d", &matrix_2[i][j]);
}
size[1][0] = num_rows_2;
size[1][1] = num_cols_2;
if((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
{
printf("Cannot create socket\n");
exit(1);
}
bzero((char*)&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(atoi(argv[2]));
inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

// SENDING MATRIX WITH SIZES OF MATRICES 1 AND 2
n = sendto(sock_fd, size, sizeof(size), 0, (struct sockaddr*)&servaddr, sizeof(servaddr));
```

```
if( n < 0)
{
perror("error in matrix 1 sending");
exit(1);
}
// SENDING MATRIX 1
n = sendto(sock_fd, matrix_1, sizeof(matrix_1),0, (struct sockaddr*)&servaddr,
sizeof(servaddr));
if( n < 0)
{
perror("error in matrix 1 sending");
exit(1);
}
// SENDING MATRIX 2
n = sendto(sock_fd, matrix_2, sizeof(matrix_2),0, (struct sockaddr*)&servaddr,
sizeof(servaddr));
if( n < 0)
{
perror("error in matrix 2 sending");
exit(1);
}
if((n=recvfrom(sock_fd, matrix_product, sizeof(matrix_product),0, NULL, NULL)) == -1)
{
perror("read error from server:");
exit(1);
}
printf("\n\nTHE PRODUCT OF MATRICES IS \n\n\n");
for( i=0; i < num_rows_1; i++)
{
for( j=0; j<num_cols_2; j++)
{
printf("%d ",matrix_product[i][j]);
}
printf("\n");
}
close(sock_fd);
}
```

### Server Program

```
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<fcntl.h>
#include<stdlib.h>

main(int argc, char * argv[])
{
    int n;
    int sock_fd;
    int i,j,k;
    int row_1, row_2, col_1, col_2;
    struct sockaddr_in servaddr, cliaddr;
    int len = sizeof(cliaddr);
    int matrix_1[10][10], matrix_2[10][10], matrix_product[10][10];
    int size[2][2];
    if(argc != 2)
    {
        fprintf(stderr, "Usage: ./server port\n");
        exit(1);
    }

    if((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        printf("Cannot create socket\n");
        exit(1);
    }
    bzero((char*)&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(atoi(argv[1]));
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    if(bind(sock_fd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0)
    {
        perror("bind failed:");
        exit(1);
    }
    // MATRICES RECEIVE
```

```
if((n = recvfrom(sock_fd, size, sizeof(size), 0, (struct sockaddr *)&cliaddr, &len)) == -1)
{
    perror("size not received:");
    exit(1);
}
// RECEIVE MATRIX 1
if((n = recvfrom(sock_fd, matrix_1, sizeof(matrix_1), 0, (struct sockaddr *)&cliaddr, &len)) ==
-1)
{
    perror("matrix 1 not received:");
    exit(1);
}
// RECEIVE MATRIX 2
if((n = recvfrom(sock_fd, matrix_2, sizeof(matrix_2), 0, (struct sockaddr *)&cliaddr, &len)) ==
-1)
{
    perror("matrix 2 not received:");
    exit(1);
}
row_1 = size[0][0];
col_1 = size[0][1];
row_2 = size[1][0];
col_2 = size[1][1];
for (i=0; i < row_1 ; i++)
for (j=0; j < col_2; j++)
{
    matrix_product[i][j] = 0;
}
for(i=0; i< row_1 ; i++)
for(j=0; j< col_2 ; j++)
for (k=0; k < col_1; k++)
{
    matrix_product[i][j] += matrix_1[i][k]*matrix_2[k][j];
}
n = sendto(sock_fd, matrix_product, sizeof(matrix_product),0, (struct sockaddr*)&cliaddr,
sizeof(cliaddr));
if( n < 0)
{
    perror("error in matrix product sending");
    exit(1);
}
```

```
}  
close(sock_fd);  
}
```

### Output

#### Server

```
anil@anil-300E4Z-300E5Z-300E7Z: ~/anil/Network_lab/expt2_udp  
File Edit View Search Terminal Help  
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt2_udp$ ./server 5300  
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt2_udp$
```

#### Client

```
anil@anil-300E4Z-300E5Z-300E7Z: ~/anil/Network_lab/expt2_udp  
File Edit View Search Terminal Help  
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt2_udp$ ./client 127.0.0.1  
5300  
Enter the number of rows of first matrix  
3  
Enter the number of columns of first matrix  
4  
Enter the values row by row one on each line  
1 2 3 4  
5 6 7 8  
1 2 3 4  
Enter the number of rows of second matrix  
4  
Enter the number of columns of second matrix  
3  
Enter the values row by row one on each line  
1 2 3  
4 5 6  
7 8 9  
1 2 3  
  
THE PRODUCT OF MATRICES IS  
34 44 54  
86 112 138  
34 44 54  
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt2_udp$
```



## Experiment 5

**Simulate sliding window flow control protocols. (Stop and Wait, Go back N, Selective Repeat ARQ protocols)**

### sliding window flow control protocols

Flow control deals with problem that sender transmits frames faster than receiver can accept, and solution is to limit sender into sending no faster than receiver can handle. Consider the simplex case: data is transmitted in one direction (Note although data frames are transmitted in one direction, frames are going in both directions, i.e. link is duplex). Stop and wait: sender sends one data frame, waits for acknowledgement (ACK) from receiver before proceeding to transmit next frame. This simple flow control will break down if ACK gets lost or errors occur → sender may wait for ACK that never arrives.

#### Go-back-n ARQ

The basic idea of go-back-n error control is: If frame  $i$  is damaged, receiver requests retransmission

of all frames starting from frame  $i$

Notice that all possible cases of damaged frame and ACK / NAK must be taken into account

In selective-reject ARQ error control, the only frames retransmitted are those receive a NAK or which time out

### 1. Stop and Wait

Server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <unistd.h>
#include <arpa/inet.h>
typedef struct packet{
char data[1024];
}Packet;
typedef struct frame{
int frame_kind; //ACK:0, SEQ:1 FIN:2
int sq_no;
int ack;
Packet packet;
}Frame;
int main(int argc, char** argv){
if (argc != 2){
printf("Usage: %s <port>", argv[0]);
exit(0);
}
int port = atoi(argv[1]);
int sockfd;
struct sockaddr_in serverAddr, newAddr;
char buffer[1024];
socklen_t addr_size;
int frame_id=0;
Frame frame_recv;
Frame frame_send;
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
memset(&serverAddr, '\0', sizeof(serverAddr));
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(port);
serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
bind(sockfd, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
addr_size = sizeof(newAddr);
while(1){
int f_rcv_size = recvfrom(sockfd, &frame_recv, sizeof(Frame), 0, (struct
sockaddr*)&newAddr, &addr_size);
if (f_rcv_size > 0 && frame_recv.frame_kind == 1 && frame_recv.sq_no ==
frame_id){
```

```
printf("[+]Frame Received: %s\n", frame_recv.packet.data);
frame_send.sq_no = 0;
frame_send.frame_kind = 0;
frame_send.ack = frame_recv.sq_no + 1;
sendto(sockfd, &frame_send, sizeof(frame_send), 0, (struct
sockaddr*)&newAddr, addr_size);
printf("[+]Ack Send\n");
}else{
printf("[+]Frame Not Received\n");
}
frame_id++;
}
close(sockfd);
return 0;
}
```

### **client.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
typedef struct packet{
char data[1024];
}Packet;
typedef struct frame{
int frame_kind; //ACK:0, SEQ:1 FIN:2
int sq_no;
int ack;
Packet packet;

}Frame;
int main(int argc, char **argv){
if (argc != 2){
printf("Usage: %s <port>", argv[0]);
exit(0);
}
int port = atoi(argv[1]);
```

```
int sockfd;
struct sockaddr_in serverAddr;
char buffer[1024];
socklen_t addr_size;
int frame_id = 0;
Frame frame_send;
Frame frame_recv;
int ack_recv = 1;
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
memset(&serverAddr, '\0', sizeof(serverAddr));
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(port);
serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
while(1){
    if(ack_recv == 1){
        frame_send.sq_no = frame_id;
        frame_send.frame_kind = 1;
        frame_send.ack = 0;
        printf("Enter Data: ");
        scanf("%s", buffer);
        strcpy(frame_send.packet.data, buffer);
        sendto(sockfd, &frame_send, sizeof(Frame), 0, (struct
        sockaddr*)&serverAddr, sizeof(serverAddr));
        printf("[+]Frame Send\n");
    }
    int addr_size = sizeof(serverAddr);
    int f_recv_size = recvfrom(sockfd, &frame_recv, sizeof(frame_recv), 0, (struct
    sockaddr*)&serverAddr, &addr_size);
    if( f_recv_size > 0 && frame_recv.sq_no == 0 && frame_recv.ack ==
    frame_id+1){
        printf("[+]Ack Received\n");
        ack_recv = 1;
    }else{
        printf("[-]Ack Not Received\n");
        ack_recv = 0;
    }
    frame_id++;
}
close(sockfd);
return 0;
```