Socket Programming

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.
Basically
A socket is an IPC mechanism. It is an operating system resource that serves to let two processes communicate with each other (a process is a running program). These two processes may or may not be in the same machine.

Socket Types

Socket types define the communication properties visible to a user. The Internet family sockets provide access to the TCP/IP transport protocols. The Internet family is identified by the value AF_INET6, for sockets that can communicate over both IPv6 and IPv4. The value AF_INET is also supported for source compatibility with old applications and for "raw" access to IPv4.

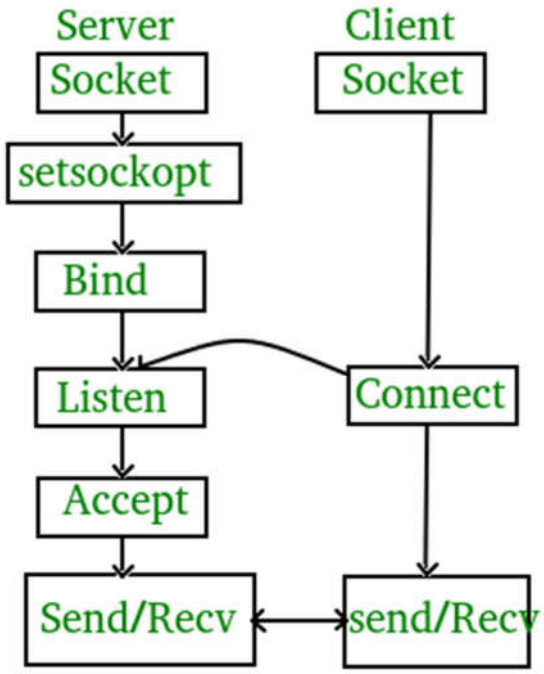
Three types of sockets are supported:

1. Stream sockets allow processes to communicate using TCP. A stream socket provides bidirectional, reliable, sequenced, and unduplicated flow of data with no record boundaries. After the connection has been established, data can be read from and written to these sockets as a byte stream. The socket type is SOCK_STREAM.
2. Datagram sockets allow processes to use UDP to communicate. A datagram socket supports bidirectional flow of messages. A process on a datagram socket can receive messages in a different order from the sending sequence and can receive duplicate messages. Record boundaries in the data are preserved. The socket type is SOCK_DGRAM.
3. Raw sockets provide access to ICMP. These sockets are normally datagram oriented, although their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not for most applications. They are provided to support developing new communication protocols or for access to more esoteric facilities of an existing protocol. Only superuser processes can use raw sockets. The socket type is SOCK_RAW.

Exercies 2.2.3:TCP Client Server Protocol

State diagram for server and client model of Tcp Protocol

If we are creating a connection between client and server using TCP then it has few functionality like, TCP is suited for applications that require high reliability, and transmission time is relatively less critical. It is used by other protocols like HTTP, HTTPs, FTP, SMTP, Telnet. TCP rearranges data packets in the order specified. There is absolute guarantee that the data transferred remains intact and arrives in the same order in which it was sent. TCP does Flow Control and requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control. It also does error checking and error recovery. Erroneous packets are retransmitted from the source to the destination.



A program to design a TCP Client –Server Which implements Echo protocol
Server.c

```
// Server side C/C++ program to demonstrate Socket programming
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#define PORT 8080
int main(int argc, char const *argv[])
{
   int server_fd, new_socket, valread;
   struct sockaddr_in address;
   int opt = 1;
   int addrlen = sizeof(address);
```

```
    char buffer[1024] = {0};
    char *hello = "Hello from server";

    // 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,
                    (socklen_t*)&addrlen))<0)
    {
        perror("accept");
        exit(EXIT_FAILURE);
    }
    valread = read( new_socket , buffer, 1024);
    printf("%s\n",buffer );
    send(new_socket , hello , strlen(hello) , 0 );
    printf("Hello message sent\n");
    return 0;
}
```

Client.c

```
// Client side C/C++ program to demonstrate Socket programming
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
   int sock = 0, valread;
   struct sockaddr_in serv_addr;
   char *hello = "Hello from client";
   char buffer[1024] = {0};
   if ((sock = 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, "127.0.0.1", &serv_addr.sin_addr)<=0)
   {
      printf("\nInvalid address/ Address not supported \n");
      return -1;
   }

   if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
   {
      printf("\nConnection Failed \n");
      return -1;
   }
   send(sock , hello , strlen(hello) , 0 );
   printf("Hello message sent\n");
   valread = read( sock , buffer, 1024);
   printf("%s\n",buffer );
   return 0;
}
```

Compiling:

```
gcc client.c -o client
gcc server.c -o server
```

Output :

```
Client:Hello message sent
Hello from server
Server:Hello from client
Hello message sent
```

Exercise 2.2.2

UDP Server-Client implementation in C

In UDP, the client does not form a connection with the server like in TCP and instead just sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive. Datagrams upon arrival contain the address of sender which the server uses to send data to the correct client.

Time Protocol

When used via UDP the time service works as follows:

S: Listen on port 37 (45 octal).

U: Send an empty datagram to port 37.

S: Receive the empty datagram.

S: Send a datagram containing the time as a 32 bit binary number.

U: Receive the time datagram.

The Time

The time is the number of seconds since 00:00 (midnight) 1 January 1900 GMT, such that the time 1 is 12:00:01 am on 1 January 1900 GMT; this base will serve until the year 2036.

For example:

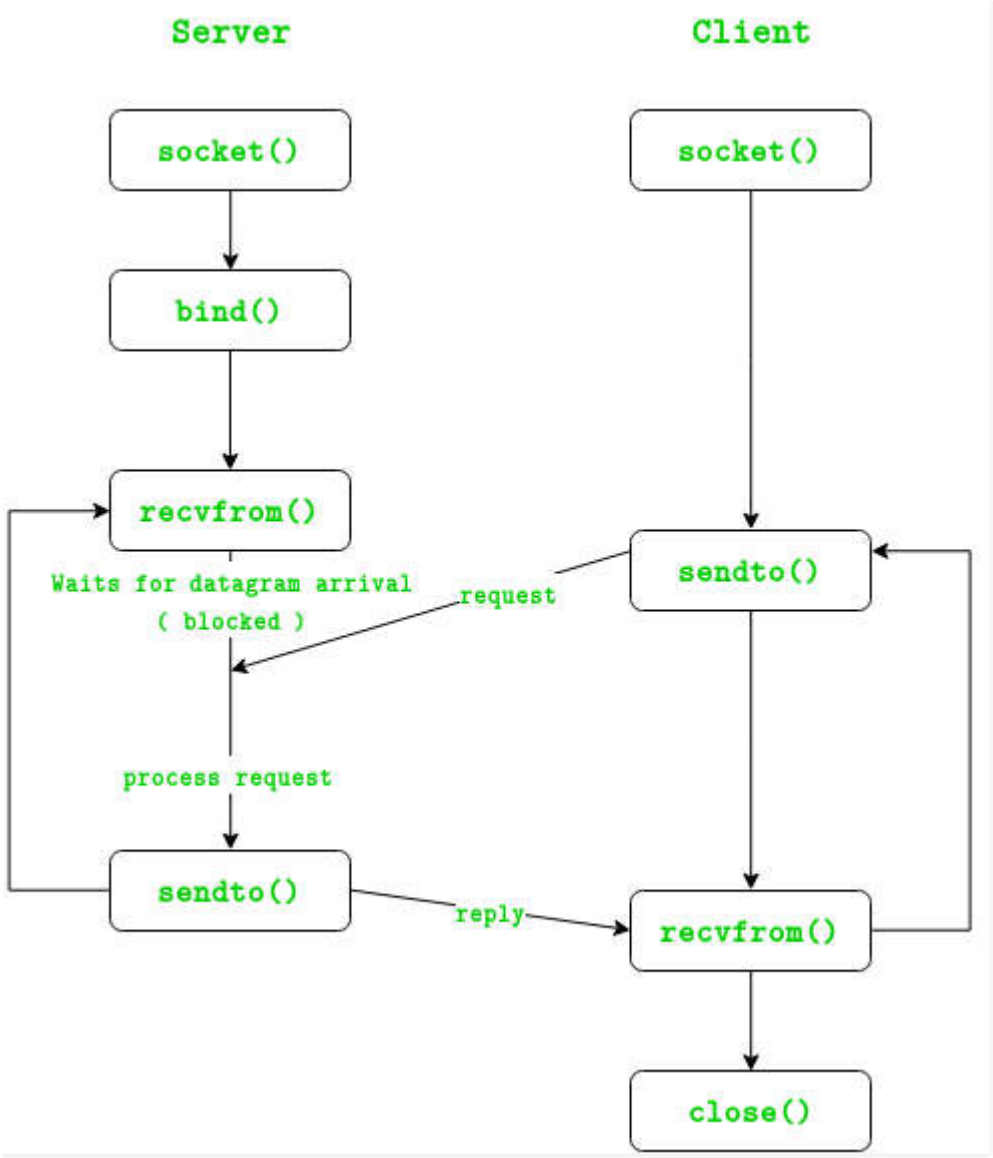
the time 2,208,988,800 corresponds to 00:00 1 Jan 1970 GMT,

2,398,291,200 corresponds to 00:00 1 Jan 1976 GMT,

2,524,521,600 corresponds to 00:00 1 Jan 1980 GMT,

2,629,584,000 corresponds to 00:00 1 May 1983 GMT,

State diagram for server and client model of UDP Protocol



The entire process can be broken down into following steps :

UDP Server :

1. Create UDP socket.
2. Bind the socket to server address.
3. Wait until datagram packet arrives from client.
4. Process the datagram packet and send a reply to client.
5. Go back to Step 3.

UDP Client :

1. Create UDP socket.
2. Send message to server.
3. Wait until response from server is recieved.
4. Process reply and go back to step 2, if necessary.
5. Close socket descriptor and exit.

Code:

UDPServer.c

```
// Server side implementation of UDP client-server model
#include <stdio.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];
    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);
  }

  int len, n;

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

  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);
  printf("Hello message sent.\n");

  return 0;
}
```

UDPClient.C

```
// Client side implementation of UDP client-server model
#include <stdio.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];
  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, len;

    sendto(sockfd, (const char *)hello, strlen(hello),
        MSG_CONFIRM, (const struct sockaddr *) &servaddr,
            sizeof(servaddr));
    printf("Hello message sent.\n");

    n = recvfrom(sockfd, (char *)buffer, MAXLINE,
                MSG_WAITALL, (struct sockaddr *) &servaddr,
                &len);
    buffer[n] = '\0';
    printf("Server : %s\n", buffer);

    close(sockfd);
    return 0;
}
```

OutPut:

```
$ ./server
Client : Hello from client
Hello message sent.
```

```
$ ./client
Hello message sent.
Server : Hello from server
```

Exercise 2.3.1

TFTP :

Run TFTP Server
$./tftp_s
Run TFTP Client
$./tftp_c GET/PUT server_address file_name
Tftp Client.c

```
/**
 * tftp_c.c - tftp client
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include "utility.h"

void *get_in_addr(struct sockaddr *sa)
{
	if (sa->sa_family == AF_INET) {
		return &(((struct sockaddr_in*)sa)->sin_addr);
	}
	return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

//CHECKS FOR TIMEOUT
int check_timeout(int sockfd, char *buf, struct sockaddr_storage their_addr, socklen_t
addr_len){
	fd_set fds;
	int n;
	struct timeval tv;

	// set up the file descriptor set
	FD_ZERO(&fds);
	FD_SET(sockfd, &fds);

	// set up the struct timeval for the timeout
```

```
	tv.tv_sec = TIME_OUT;
	tv.tv_usec = 0;

	// wait until timeout or data received
	n = select(sockfd+1, &fds, NULL, NULL, &tv);
	if (n == 0){
		printf("timeout\n");
		return -2; // timeout!
	} else if (n == -1){
		printf("error\n");
		return -1; // error
	}

	return recvfrom(sockfd, buf, MAXBUFLEN-1 , 0, (struct sockaddr *)&their_addr,
&addr_len);
}

int main(int argc, char* argv[]){
	int sockfd;
	struct addrinfo hints, *servinfo, *p;
	int rv;
	int numbytes;
	char buf[MAXBUFLEN];
	char s[INET6_ADDRSTRLEN];
	struct sockaddr_storage their_addr;
	socklen_t addr_len;

	if(argc != 4){// CHECKS IF args ARE VALID
		fprintf(stderr,"USAGE: tftp_c GET/PUT server filename\n");
		exit(1);
	}

	char *server = argv[2];// server address
	char *file = argv[3]; // file name on which operation has to be done

	//===========CONFIGURATION OF CLIENT - STARTS===========
	memset(&hints, 0, sizeof hints);
	hints.ai_family = AF_UNSPEC;
	hints.ai_socktype = SOCK_DGRAM;
	if((rv = getaddrinfo(server, SERVERPORT, &hints, &servinfo)) != 0){
		fprintf(stderr, "CLIENT: getaddrinfo: %s\n", gai_strerror(rv));
		return 1;
	}
```

```
	// loop through all the results and make a socket
	for(p = servinfo; p != NULL; p = p->ai_next) {
		if ((sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1){
			perror("CLIENT: socket");
			continue;
		}
		break;
	}
	if(p == NULL){
		fprintf(stderr, "CLIENT: failed to bind socket\n");
		return 2;
	}
```

Tftp Server.c

```
/**
	 * tftp_s.c - tftp server
	 */
	#include <stdio.h>
	#include <stdlib.h>
	#include <unistd.h>
	#include <errno.h>
	#include <string.h>
	#include <sys/types.h>
	#include <sys/socket.h>
	#include <netinet/in.h>
	#include <arpa/inet.h>
	#include <netdb.h>
	#include "utility.h"

	void *get_in_addr(struct sockaddr *sa)
	{
		if (sa->sa_family == AF_INET) {
			return &(((struct sockaddr_in*)sa)->sin_addr);
		}
		return &(((struct sockaddr_in6*)sa)->sin6_addr);
	}

	//CHECKS FOR TIMEOUT
	int check_timeout(int sockfd, char *buf, struct sockaddr_storage their_addr, socklen_t
	addr_len){
```

```
	fd_set fds;
	int n;
	struct timeval tv;

	// set up the file descriptor set
	FD_ZERO(&fds);
	FD_SET(sockfd, &fds);

	// set up the struct timeval for the timeout
	tv.tv_sec = TIME_OUT;
	tv.tv_usec = 0;

	// wait until timeout or data received
	n = select(sockfd+1, &fds, NULL, NULL, &tv);
	if (n == 0){
		printf("timeout\n");
		return -2; // timeout!
	} else if (n == -1){
		printf("error\n");
		return -1; // error
	}

	return recvfrom(sockfd, buf, MAXBUFLEN-1 , 0, (struct sockaddr *)&their_addr,
&addr_len);
}

int main(void){
	int sockfd;
	struct addrinfo hints, *servinfo, *p;
	int rv;
	int numbytes;
	struct sockaddr_storage their_addr;
	char buf[MAXBUFLEN];
	socklen_t addr_len;
	char s[INET6_ADDRSTRLEN];

	//===========CONFIGURATION OF SERVER - START===========
	memset(&hints, 0, sizeof hints);
	hints.ai_family = AF_UNSPEC; // set to AF_INET to force IPv4
	hints.ai_socktype = SOCK_DGRAM;
	hints.ai_flags = AI_PASSIVE; // use my IP

	if ((rv = getaddrinfo(NULL, MYPORT, &hints, &servinfo)) != 0) {
```

```
        fprintf(stderr, "SERVER: getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
}

// loop through all the results and bind to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
                perror("SERVER: socket");
                continue;
        }
        if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
                close(sockfd);
                perror("SERVER: bind");
                continue;
        }
        break;
}
if (p == NULL) {
        fprintf(stderr, "SERVER: failed to bind socket\n");
        return 2;
}
freeaddrinfo(servinfo);

printf("SERVER: waiting to recvfrom...\n")
```

Summary

TFTP is a very simple protocol used to transfer files. It is from this that its name comes, Trivial File Transfer Protocol or TFTP. Each nonterminal packet is acknowledged separately. This document describes the protocol and its types of packets. The document also explains the reasons behind some of the design decisions.