Name: Nadira Islam ID : IT-17051

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.

Java and **C#/C++**. cli/VB+ should support the creation of a socket server with relatively few lines of code, as (the same as **python**) they have already-made libraries supporting most of the functionality. They are more verbose than **Python** though so we'll write much more code.

Socket Types

Sockets are classified according to communication properties. Processes usually communicate between sockets of the same type. However, if the underlying communication protocols support the communication, sockets of different types can communicate.
Each socket has an associated type, which describes the semantics of communications using that socket. The socket type determines the socket communication properties such as reliability, ordering, and prevention of duplication of messages. The basic set of socket types is defined in the **sys/socket.h** file:

```
1. /*Standard socket types */
2. #define  SOCK_STREAM             1 /*virtual circuit*/
3. #define  SOCK_DGRAM              2 /*datagram*/
4. #define  SOCK_RAW                3 /*raw socket*/
5. #define  SOCK_RDM                4 /*reliably-delivered message*/
6. #define  SOCK_CONN_DGRAM         5 /*connection datagram*/
```

Three types of sockets are supported:

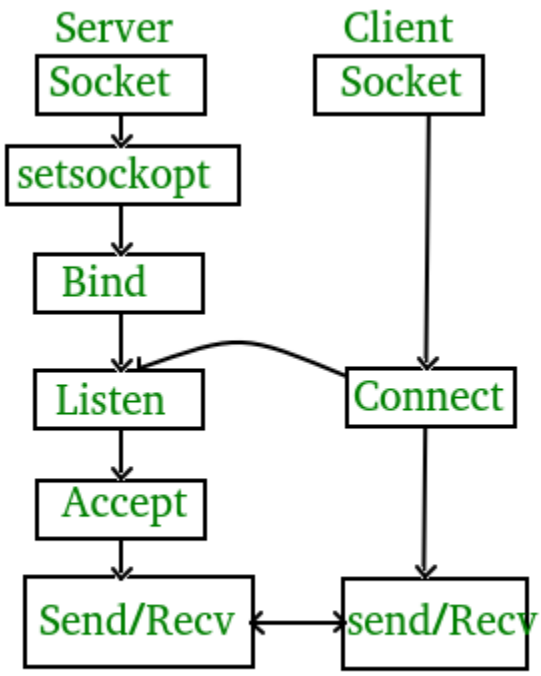
1. Stream sockets allow processes to communicate using TCP. A stream socket provides bidirectional, reliable, sequenced, and unduplicated flow of data with no record boundaries. After the connection has been established, data can be read from and written to these sockets as a byte stream. The socket 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

The TCP/IP protocol allows systems to communicate even if they use different types of network hardware. For example, TCP, through an Internet connection, transmits messages between a system using Ethernet and another system using Token Ring. TCP controls the accuracy of data transmission. IP, or Internet Protocol, performs the actual data transfer between different systems on the network or Internet.

Using TCP binding, we can create both client and server portions of client-server systems. In the client-server type of distributed database system, users on one or more client systems can process information stored in a database on another system, called the server.



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

// Server side python program to demonstrate Socket programming

```
#!/usr/bin/env python3

import socket

HOST = '127.0.0.1'  # Standard loopback interface address (localhost)
PORT = 65432        # Port to listen on (non-privileged ports are > 1023)
```

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Client.c

```
#!/usr/bin/env python3

import socket

HOST = '127.0.0.1'  # The server's hostname or IP address
PORT = 65432        # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)

print('Received', repr(data))
```

Output:

Open a terminal or command prompt, navigate to the directory that contains your scripts, and run the server:

```
$ ./echo-server.py
```

Your terminal will appear to hang. That's because the server is blocked (suspended) in a call:

```
conn, addr = s.accept()
```
It's waiting for a client connection. Now open another terminal window or command prompt and run the client:

```
$ ./echo-client.py
Received b'Hello, world'
```

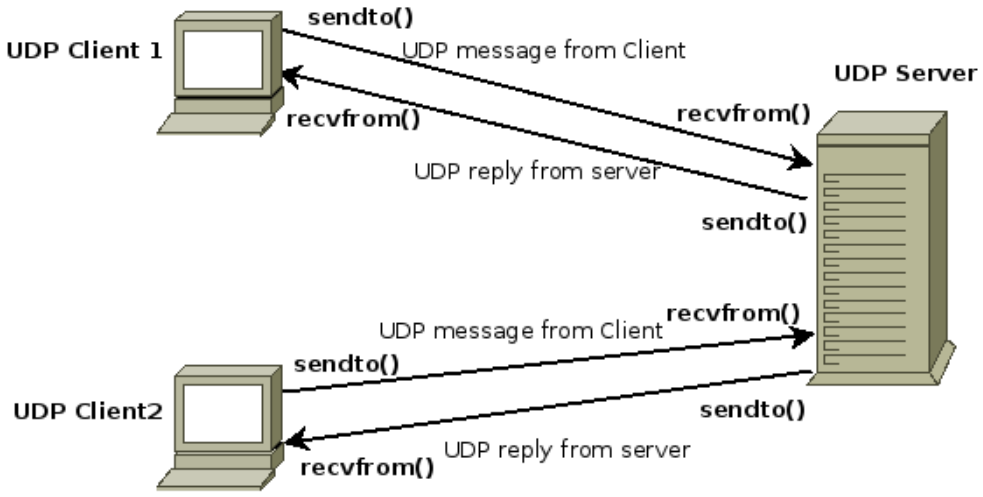
In the server window, we will see:

```
$ ./echo-server.py
Connected by ('127.0.0.1', 64623)
```
In the output above, the server printed the `addr` tuple returned from `s.accept()`. This is the client's IP address and TCP port number. The port number, 64623, will most likely be different when you run it on your machine.

Exercise 2.2.2

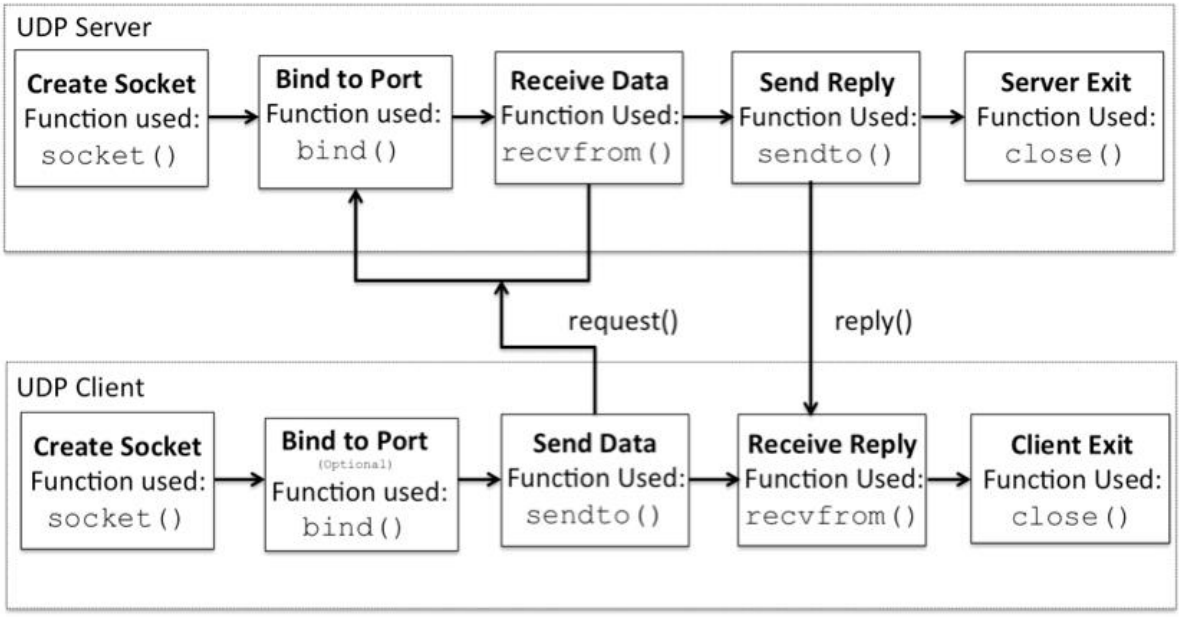
UDP Server-Client implementation in python

UDP is the abbreviation of User Datagram Protocol. UDP makes use of Internet Protocol of the TCP/IP suit. In communications using UDP, a client program sends a message packet to a destination server wherein the destination server also runs on UDP.



Properties of UDP:

- The UDP does not provide guaranteed delivery of message packets. If for some issue in a network if a packet is lost it could be lost forever.
- Since there is no guarantee of assured delivery of messages, UDP is considered an unreliable protocol.
- The underlying mechanisms that implement UDP involve no connection-based communication. There is no streaming of data between a UDP server or and an UDP Client.
- An UDP client can send "n" number of distinct packets to an UDP server and it could also receive "n" number of distinct packets as replies from the UDP server.
- Since UDP is connectionless protocol the overhead involved in UDP is less compared to a connection based protocol like TCP.



Example: UDP Server using Python

```
import socket


localIP     = "127.0.0.1"

localPort   = 20001

bufferSize  = 1024


msgFromServer       = "Hello UDP Client"

bytesToSend         = str.encode(msgFromServer)
```

```
# Create a datagram socket

UDPServerSocket = socket.socket(family=socket.AF_INET,
type=socket.SOCK_DGRAM)


# Bind to address and ip

UDPServerSocket.bind((localIP, localPort))


print("UDP server up and listening")


# Listen for incoming datagrams

while(True):

    bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)

    message = bytesAddressPair[0]

    address = bytesAddressPair[1]

    clientMsg = "Message from Client:{}".format(message)
    clientIP  = "Client IP Address:{}".format(address)

    print(clientMsg)
    print(clientIP)


    # Sending a reply to client

    UDPServerSocket.sendto(bytesToSend, address)
```

```
UDP server up and listening

Message from Client:b"Hello UDP Server"

Client IP Address:("127.0.0.1", 51696)
```

Example: UDP Client using Python

```
import socket


msgFromClient       = "Hello UDP Server"

bytesToSend         = str.encode(msgFromClient)

serverAddressPort   = ("127.0.0.1", 20001)
```

```
bufferSize          = 1024

# Create a UDP socket at client side
UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

# Send to server using created UDP socket
UDPClientSocket.sendto(bytesToSend, serverAddressPort)

msgFromServer = UDPClientSocket.recvfrom(bufferSize)

msg = "Message from Server {}".format(msgFromServer[0])
print(msg)
```

Output:

```
Message from Server b"Hello UDP Client"
```

Exercise 2.3.1

TFTP :
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

TCP is comparatively slower than **UDP**. **UDP** is faster, simpler and more efficient than **TCP**. Retransmission of lost packets is possible in **TCP**, but not in **UDP**. There is no retransmission of lost packets in User Datagram Protocol (**UDP**). Published advice states that **TCP** should not be swallowed, and recommends drinking plenty of water if 30ml or more of **TCP** is swallowed, and seeking medical advice if discomfort persists. Phenolic compounds such as those in **TCP** are harmful to cats.

TFTP is a simple **protocol** for transferring files, implemented on top of the UDP/IP protocols using well-known **port** number 69.