**Course: CSNC 2411**

**Computer Communications and Networks**

**(Lab)**



**Lab 3**

Socket Programming:

TCP Client Server Communication

(TCP iterative server)

Lab Manual 03

|  |
| --- |
| Objectives  * Introduction to TCP * TCP Iterative Server and Client communication * TCP related Socket API functions |

# Reference Material

**What is TCP?**

TCP (Transmission Control Protocol) is a standard that defines how application programs can exchange data reliably over the Internet, by establishing and maintaining network connections. It provides process to process communication. TCP operates on top of Internet Protocol (IP), which defines how hosts (computers) send/receive packets of data to each other, providing host to host connectivity. Together, TCP and IP are the basic rules defining the Internet operation.

TCP is a connection-oriented protocol, where a connection is established and maintained till the application programs at each end have finished exchanging messages. TCP uses 3-way handshake to establish the connection; and 4-way handshake to close the connection. Below is the sequence of system calls, used for both client and server, to communicate using TCP.

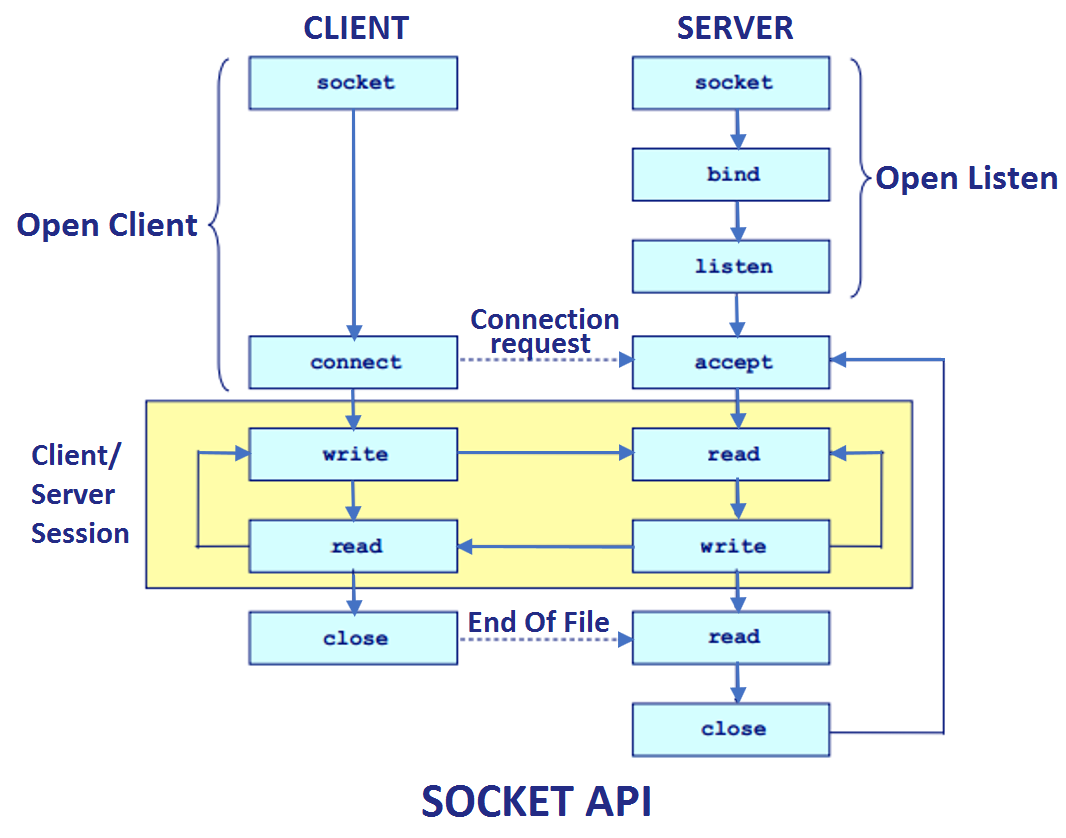


Figure 1: TCP Client Server (Iterative)

A TCP iterative Server can handle only one client at a time. Once the first client’s request is serviced, the Server can connect to the next waiting or incoming client.

**TCP Related Socket API Calls**

**Create Socket**

Int socket (int family, int type, int protocol);

returns socket descriptor; -1 on error and sets errno

family : address family / protocol family

* AF\_INET for IPv4, AF\_INET6 for IPv6

type : type of communication

* SOCK\_STREAM for TCP
* SOCK\_DGRAM for UDP
* SOCK\_RAW for Raw socket

Protocol : protocol within family

* typically 0 (except for raw socket)

**Bind the Socket**

int bind (int sockfd, struct sockaddr\* serverAddr, int addrlen);

bind a socket to a local IP address & port number

returns 0 on success; -1 on failure and sets errno

sockfd : socket descriptor (returned from socket)

serverAddr : includes IP address and port number

* IP address set by kernel if value passed is INADDR\_ANY,   
  else set by caller
* port number set by kernel if value passed is 0,   
  else set by caller

addrlen : length of address structure

* sizeof (structsockaddr\_in)

**Listen**

int listen (int sockfd, int backlog);

server puts socket into listening state *(wait for connections rather than initiate a connection)*

returns 0 on success; -1 on failure and sets errno

sockfd : socket descriptor

backlog : bound on length of un-accept()ed connection queue

(connection backlog)

**Accept**

int accept (int sockfd, struct sockaddr\* cliaddr, int\* addrlen);

server accepts a new connection *(first one off the queue of pending connections)*

returns a new socket descriptor (connected socket) created by kernel;  
-1 on error and sets errno

sockfd : socket descriptor (listening socket)

cliaddr : IP address and port number of client (when returned)

addrlen : length of address structure

* addrlen is a value-result argument
* caller passes size of client’s socket address structure
* kernel returns number of bytes stored in the address structure

**Connect**

int connect (int sockfd, struct sockaddr\* servaddr, int addrlen);

client connects to another socket (server)

returns 0 on success; -1 on failure and sets errno

sockfd : socket descriptor

servaddr : IP address and port number of server

addrlen : length of address structure

# Lab Tasks

1. **Your task is to add required socket Api calls in the files provided to run TCP server and client. After that, compile and run both server and client, understand the code and paste the screenshot of the output here. [5 marks]**
2. **In this task,client needs to read the data from the file and encrypt it. Requirements are as follows [15 marks]**

* **Read data from the file named as fileData.txt**
* **Add 3 in all the lowercase letters of the data**
* **Add 2 in all the uppercase letters of the data**
* **Add 1 in the numeric letter of the data**
* **Send the encrypted data to server for decryption**

**Requirements for the server are as follows**

* **Receive data from client**
* **subtract 3 in all lowercase letters of the data**
* **subtract 2 in all the uppercase letters of the data**
* **subtract 1 in all the numeric letters of the data**
* **Send back the decrypted data to the client**

**In the end client will show the decrypted data on terminal sent by the server.**

**//Client**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define MAX\_BUFFER\_SIZE 1024

void encryptData(char \*data) {

int i = 0;

while (data[i] != '\0') {

if (data[i] >= 'a' && data[i] <= 'z')

data[i] = (data[i] - 'a' + 3) % 26 + 'a';

else if (data[i] >= 'A' && data[i] <= 'Z')

data[i] = (data[i] - 'A' + 2) % 26 + 'A';

else if (data[i] >= '0' && data[i] <= '9')

data[i] = (data[i] - '0' + 1) % 10 + '0';

i++;

}

}

int main() {

struct sockaddr\_in serverAddr;

int sock = 0, valread;

char buffer[MAX\_BUFFER\_SIZE] = {0};

char \*message = "Hello from client";

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {

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

return -1;

}

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_port = htons(PORT);

if(inet\_pton(AF\_INET, "127.0.0.1", &serverAddr.sin\_addr)<=0) {

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

return -1;

}

if (connect(sock, (struct sockaddr \*)&serverAddr, sizeof(serverAddr)) < 0) {

printf("\nConnection Failed \n");

return -1;

}

encryptData(message);

send(sock, message, strlen(message), 0);

valread = read(sock, buffer, MAX\_BUFFER\_SIZE);

printf("Message received from server: %s\n", buffer);

close(sock);

return 0;

}

//Server

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define MAX\_BUFFER\_SIZE 1024

void decryptData(char \*data) {

int i = 0;

while (data[i] != '\0') {

if (data[i] >= 'a' && data[i] <= 'z')

data[i] = (data[i] - 'a' - 3 + 26) % 26 + 'a';

else if (data[i] >= 'A' && data[i] <= 'Z')

data[i] = (data[i] - 'A' - 2 + 26) % 26 + 'A';

else if (data[i] >= '0' && data[i] <= '9')

data[i] = (data[i] - '0' - 1 + 10) % 10 + '0';

i++;

}

}

int main() {

int server\_fd, new\_socket;

struct sockaddr\_in address;

int addrlen = sizeof(address);

char buffer[MAX\_BUFFER\_SIZE] = {0};

char \*response = "Message received by server";

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

perror("socket failed");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

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

address.sin\_port = htons(PORT);

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);

}

int valread = read(new\_socket, buffer, MAX\_BUFFER\_SIZE);

decryptData(buffer);

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

return 0;

}

1. **In previous Lab, you have done Client-Server communication for UDP. The flow chart for TCP Client-Server communication is provided in the reference material above.   
   You are now required to write the code for a TCP iterative Server and Client, following the steps from flow chart, and run the TCP Client-Server programs.** **[30 marks]**

* Take snap of the Server in listening stage in terminal window.
* When client is connected to Server, show its Port number and process ID in terminal window.
* Client sends a file name to Server.
* Server sends the file to Client.
* After receiving file, Client closes its connection with Server.
* But Server should keep running and now be ready to service a new Client request.

//Client

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#define PORT 8080

#define SERVER\_IP "127.0.0.1"

#define MAX\_BUFFER\_SIZE 1024

#define RECEIVED\_FILE "received\_file.txt"

int main() {

struct sockaddr\_in server\_addr;

int sock = 0;

char buffer[MAX\_BUFFER\_SIZE] = {0};

FILE \*fileptr;

char file\_data[MAX\_BUFFER\_SIZE];

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {

perror("socket creation failed");

exit(EXIT\_FAILURE);

}

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

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

if(inet\_pton(AF\_INET, SERVER\_IP, &server\_addr.sin\_addr)<=0) {

perror("invalid address/ Address not supported");

exit(EXIT\_FAILURE);

}

if (connect(sock, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) {

perror("connection failed");

exit(EXIT\_FAILURE);

}

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

printf("Enter the file name to request from server: ");

scanf("%s", buffer);

send(sock, buffer, strlen(buffer), 0);

fileptr = fopen(RECEIVED\_FILE, "w");

if (fileptr == NULL) {

perror("File opening failed");

exit(EXIT\_FAILURE);

}

printf("Receiving file from server...\n");

while (1) {

memset(file\_data, 0, MAX\_BUFFER\_SIZE);

int bytes\_received = recv(sock, file\_data, MAX\_BUFFER\_SIZE, 0);

if (bytes\_received <= 0) break;

fputs(file\_data, fileptr);

}

printf("File received successfully.\n");

fclose(fileptr);

close(sock);

return 0;

}

//Server

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#define PORT 8080

#define MAX\_BUFFER\_SIZE 1024

#define FILENAME "server\_file.txt"

int main() {

int server\_fd, new\_socket;

struct sockaddr\_in address;

int addrlen = sizeof(address);

char buffer[MAX\_BUFFER\_SIZE] = {0};

FILE \*fileptr;

char file\_data[MAX\_BUFFER\_SIZE];

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

perror("socket failed");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

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

address.sin\_port = htons(PORT);

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);

}

printf("Server is listening on port %d...\n", PORT);

while(1) {

if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address, (socklen\_t\*)&addrlen))<0) {

perror("accept");

exit(EXIT\_FAILURE);

}

printf("Client connected. Port: %d, Process ID: %d\n", ntohs(address.sin\_port), getpid());

memset(buffer, 0, MAX\_BUFFER\_SIZE);

read(new\_socket, buffer, MAX\_BUFFER\_SIZE);

printf("Requested file from client: %s\n", buffer);

fileptr = fopen(FILENAME, "r");

if (fileptr == NULL) {

perror("File opening failed");

exit(EXIT\_FAILURE);

}

memset(file\_data, 0, MAX\_BUFFER\_SIZE);

while (fgets(file\_data, MAX\_BUFFER\_SIZE, fileptr) != NULL) {

send(new\_socket, file\_data, strlen(file\_data), 0);

memset(file\_data, 0, MAX\_BUFFER\_SIZE);

}

fclose(fileptr);

close(new\_socket);

printf("File sent to client. Connection closed.\n");

}

return 0;

}