**1. Connection oriented servers:**

**1.1 Iterative TCP File server**

**Readme.txt**

1. To Open the editor:   
gedit tcp\_iterative\_server.c  
  
2. To Compile to program  
make all  
  
3. To Execute the program  
./tcp\_iterative\_server

**Makefile**

.PHONY: server client

all: client

client:

gcc tcp\_iterative\_server.c -o tcp\_iterative\_server

clean:

rm client

**Source files**

/\*  
TCP iterative server program by group-02  
Make file, code, output are provided  
Reference: Internetworking with TCP/IP Vol. 3, Client-Server programming and applications, Comer and Stevens, Linux/POSIX Sockets version, ISBN: 0-13-032071-4, 2001.  
\*/  
#include <errno.h>  
#include <netinet/in.h>  
#include <time.h>  
#include <unistd.h>  
#include <stdarg.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <arpa/inet.h>  
#include <netdb.h>  
#include <string.h>  
#include <fcntl.h>  
  
#define \_\_USE\_BSD 1  
#define UNIXEPOCH 2208988800   
  
#ifndef INADDR\_NONE  
#define INADDR\_NONE 0xffffffff  
#endif /\* INADDR\_NONE \*/  
typedef unsigned long u\_long;  
typedef unsigned short u\_short;  
extern int errno;  
  
#define LINELEN 128  
  
int passiveTCP(const char \*service,int qlen);  
int errexit(const char \*format, ...);  
int passivesock(const char \*service, const char \*transport,int qlen);  
int TCPdaytimed(int fd);  
#define QLEN 5  
/\*------------------------------------------------------------------------  
\* main - Iterative TCP server  
\*------------------------------------------------------------------------  
\*/  
int main(int argc, char \*argv[])  
{  
int fil;  
size\_t rd,wr;  
struct sockaddr\_in fsin;  
char \*service = "echo";  
char buffer[2048];  
int msock,ssock;  
unsigned int alen;  
  
//printf("Before switch \n");  
switch (argc) {  
case 1:  
break;  
case 2:  
service = argv[1];  
break;  
default:  
errexit("usage: TCP [port]\n");  
}  
//printf("After Switch \n");  
//fflush(stdout);  
msock = passiveTCP(service, QLEN);  
while (1) {  
printf("Waiting for client\n");  
//fflush(stdout);  
ssock = accept(msock, (struct sockaddr \*)&fsin, &alen);  
//ssock = 2;  
if (ssock < 0)  
errexit("accept failed: %s\n", strerror(errno));  
  
printf("Connection Successful \n");  
//fflush(stdout);  
  
int file\_n=recv(ssock,buffer,1024,0);  
printf("Document requested by client is %s\n",buffer);  
  
fil=open(buffer,O\_RDONLY);  
printf("File is opened and the content of the file is read\n\n");  
//printf("%s",buffer);  
while((rd=read(fil,buffer,2048))>0)  
{  
wr=send(ssock,buffer,rd,0);  
  
}  
  
(void) close(ssock);  
}  
  
}  
  
  
  
  
/\* passiveTCP.c - passiveTCP \*/  
/\*------------------------------------------------------------------------  
\* passiveTCP - create a passive socket for use in a TCP server  
\*------------------------------------------------------------------------  
\*/  
int passiveTCP(const char \*service, int qlen)  
/\*  
\* Arguments:  
\* service - service associated with the desired port  
\* qlen  
- maximum server request queue length  
\*/  
{  
return passivesock(service, "tcp", qlen);  
}  
  
  
/\* passivesock.c - passivesock \*/  
  
u\_short portbase = 0;  
/\* port base, for non-root servers  
\*/  
/\*------------------------------------------------------------------------  
\* passivesock - allocate & bind a server socket using TCP or UDP  
\*------------------------------------------------------------------------  
\*/  
  
int passivesock(const char \*service, const char \*transport, int qlen)  
/\*  
\* Arguments:  
\* service  
- service associated with the desired port  
\* transport - transport protocol to use ("tcp" or "udp")  
\* qlen  
- maximum server request queue length  
\*/  
{  
struct servent \*pse; /\* pointer to service information entry \*/  
struct protoent \*ppe; /\* pointer to protocol information entry\*/  
struct sockaddr\_in sin; /\* an Internet endpoint address\*/  
int s, type;  
  
memset(&sin, 0, sizeof(sin));  
sin.sin\_family = AF\_INET;  
sin.sin\_addr.s\_addr = INADDR\_ANY;  
/\* Map service name to port number \*/  
if ( pse = getservbyname(service, transport) )  
sin.sin\_port = htons(10023);  
else if ( (sin.sin\_port = htons((u\_short)atoi(service))) == 0 )  
errexit("can't get \"%s\" service entry\n", service);  
/\* Map protocol name to protocol number \*/  
if ( (ppe = getprotobyname(transport)) == 0)  
errexit("can't get \"%s\" protocol entry\n", transport);  
/\* Use protocol to choose a socket type \*/  
if (strcmp(transport, "udp") == 0)  
type = SOCK\_DGRAM;  
else  
type = SOCK\_STREAM;  
/\* Allocate a socket \*/  
s = socket(PF\_INET, type, ppe->p\_proto);  
if (s < 0)  
errexit("can't create socket: %s\n", strerror(errno));  
/\* Bind the socket \*/  
if (bind(s, (struct sockaddr \*)&sin, sizeof(sin)) < 0)  
errexit("can't bind to %s port: %s\n", service,  
strerror(errno));  
if (type == SOCK\_STREAM && listen(s, qlen) < 0)  
errexit("can't listen on %s port: %s\n", service,  
strerror(errno));  
return s;  
}  
  
  
/\*------------------------------------------------------------------------  
\* errexit - print an error message and exit  
\*------------------------------------------------------------------------  
\*/  
/\*VARARGS1\*/  
int errexit(const char \*format, ...)  
{  
va\_list args;  
va\_start(args, format);  
vfprintf(stderr, format, args);  
va\_end(args);  
exit(1);  
}

**Output**

ubuntu@ubuntu:~/tcp/TCP\_server$ ./tcp\_iterative\_server

Waiting for client

Connection Successful

Document requested by client is test1

File is opened and the content of the file is read

Waiting for client

Connection Successful

Document requested by client is test2

File is opened and the content of the file is read

Waiting for client

**1.2 Concurrent multiprocessing TCP File server with one process per request**

**Readme.txt**

Steps to execute the program:

1. Open the terminal

2. Navigate to the program directory

3. Build server & client:

make all

Build server alone:

make server

Build client alone:

make client

5. Run server and client on two different terminals:

./server <port>

ex: ./server 10000

./client <server\_IP> <port> <file\_name>

ex: ./client 127.0.0.1 10000 one\_k\_file.txt

ex: ./client 127.0.0.1 10000 two\_k\_file.txt

6. Enter ^C to quit running the server.

TCP File Server: Server program creates child processes to handle the incoming client

request on demand. Child exits once client is served. server sends specified file to client.

Server can transfer 1. one\_k\_file.txt (1K size)

2. two\_k\_file.txt (2K size)

TCP test client: Client takes a file name input from the user and sends it to the TCP server.

Server transfer the specifies file back to the client. Client saves it on local disk with name:

test\_file.txt

If specified file does not exist on the server then server sends back an error message which

is saved into file "test\_file.txt". However file transfer is printed "successful" on the

client display. User must verify the copied file on the disk.

**Makefile**

# Make file targets to build TCP client & TCP server (forking on demand)

# Format

# target: dependencies

# action

.PHONY: server

all: server

server:

gcc tcp\_server.c -o server

clean:

rm server

**Source files**

**Tcp\_server.c**

// CMPE\_207 Assignment\_2

// Author - Team #2

// TCP server program

// This is a concurrent TCP server program. Server

// listens for active clients. It creates new child

// processes to handle each new client connection.

// child exits once client is served. Parent process

// waits for new connections in an infinite loop.

// To compile: gcc tcp\_server -o server

// To run: ./server port\_number

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/socket.h>

#include<sys/types.h>

#include<netinet/in.h>

#include<string.h>

#include<arpa/inet.h>

#include<netdb.h>

#include<sys/wait.h>

#include<sys/stat.h>

#include<fcntl.h>

#define ERROR 0

#define MAX\_CLIENTS 5

#define MAX\_DATA 1024

#define NUM\_CHILDREN 5

int passivesock(const char \*service, const char \*transport, int qlen, int \*sock)

{

struct servent \*pse;

struct protoent \*ppe;

struct sockaddr\_in server;

int res = 1;

int type;

//reset socket address structure.

memset(&server,0,sizeof(server));

server.sin\_family = AF\_INET;

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

//map service name to port number

if(pse = getservbyname(service, transport))

server.sin\_port = htons(ntohs((u\_short)pse->s\_port));

else if((server.sin\_port = htons((u\_short)atoi(service))) == 0)

//perror("Cant get \"%s\" service entry\n", service);

perror("Cant get service entry\n");

bzero(&server.sin\_zero, 8);

//map transport protocol name to protocol number

ppe = getprotobyname(transport);

if((ppe == NULL))

//perror("cant get \"%s\" protocol entry\n", transport);

perror("Cant get protocol entry\n");

//use protocol to chose a socket type

if(strcmp(transport, "udp") == 0)

type = SOCK\_DGRAM;

else

type = SOCK\_STREAM;

//allocate a socket

\*sock = socket(PF\_INET, type, ppe->p\_proto);

if (\*sock < 0){

res = -1;

perror("Cant create socket\n");

}

//Bind the socket

if(bind(\*sock, (struct sockaddr \*)&server, sizeof(server)) < 0)

{

res = -1;

perror("Cant bind to specified port\n");

}

//listen for incoming client connections

if(type == SOCK\_STREAM && (listen(\*sock, qlen) < 0))

{

res = -1;

perror("Listen error\n");

}

return res;

}

/\*\*

\* Simple utility function that reads a line from a file descriptor fd,

\* up to maxlen bytes -- ripped from Unix Network Programming, Stevens.

\*/

int

readline(int fd, char \*buf, int maxlen)

{

int n, rc;

char c;

for (n = 1; n < maxlen; n++) {

rc = read(fd, &c, 1);

if (rc == 1) {

\*buf++ = c;

if (c == '\n')

break;

} else if (rc == 0) {

if (n == 1)

return 0; // EOF, no data read

else

break; // EOF, read some data

} else

return -1; // error

}

\*buf = '\0'; // null-terminate

return n;

}

int main(int argc, char \*\*argv)

{

struct sockaddr\_in client; //client data structure

int n, sock, result, len;

int clientfd;

int num;

pid\_t cpid;

int data\_len;

char data[MAX\_DATA];

char line[MAX\_DATA];

char cpid\_s[32];

char err\_msg[32] = "";

int fd;

if(argc != 2)

{

printf("Usage: ./server <port>\n");

exit(0);

}

result = passivesock(argv[1], "tcp", MAX\_CLIENTS, &sock);

if (result == -1)

exit(-1);

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Concurrent TCP File Server Program: listening on socket %s..\n", argv[1]);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Child processes forked on demand.One per client connection: \n\n");

while (1)

{

socklen\_t sockaddr\_len = sizeof(struct sockaddr\_in);

socklen\_t client\_t=sizeof(struct sockaddr\_in);

memset(&client, 0, sizeof(client));

clientfd = accept(sock, (struct sockaddr\*)&client, &client\_t);

if ( clientfd < 0 ) {

perror("Accept Error");

exit(-1);

}

printf("New connection request from %s:%d ==> \n",inet\_ntoa(client.sin\_addr), ntohs(client.sin\_port));

cpid = fork();

if (cpid != 0){

sleep(0.05);

close(clientfd);

if(cpid == -1){

perror("fork error");

exit(-1);

}

}

else

{

// we are in child

close(sock);

printf("forking new child process %d to handle the client.. \n",getpid());

memset(line, 0, MAX\_DATA);

/\* Read a line from the client socket ... \*/

n = readline(clientfd, line, MAX\_DATA);

if (n <= 0) {

sprintf(err\_msg, "Child: %d - socket read error..\n",getpid());

//perror("Child: %d socket read error..",getpid());

exit(-1);

}

printf("File request from client: %s\n", line);

line[n-1] = '\0';

result = strcmp(line, "one\_k\_file.txt");

if(!result) { //line matches "test\_file\_1.txt"

fd = open("one\_k\_file.txt", O\_RDONLY);

} else {

result = strcmp(line, "two\_k\_file.txt");

if(!result){ //line matches "test\_file\_2.txt"

fd = open("two\_k\_file.txt", O\_RDONLY);

}else{

printf("File does not exist. Error msg sent back...\n");

strcpy(data, "Error: Requested File does not exist on the server.\n");

if(0 > send(clientfd, data, strlen(data), 0)){

perror("send error");

exit(-1);

}

printf("Closing client connection. child exiting...\n");

close(clientfd);

exit(0);

}

}

if (fd < 0)

{

printf("File open Error..\n");

close(clientfd);

exit(-1);

}

// set pointer to beginning of the file.

lseek(fd, 0, SEEK\_SET);

while(1)

{

//read MAX\_DATA data bytes from the file

result = readline(fd, data, MAX\_DATA);

//printf("File read result = %d bytes\n", result);

if(result == 0)

break;

/\*send MAX\_DATA bytes.\*/

result = send(clientfd, data, result, 0);

//printf("send result = %d bytes\n", result);

if (0 > result)

{

perror("Send Error");

break;

}

}//end inner while

//close file

close(fd);

if(result == 0)

printf("File transfer done.\n");

printf("\n%d ==> Closing client connection. Child exiting..\n\n", getpid());

close(clientfd);

exit(0);

}//end else

} //end while(1)

//close the server socket

close(sock);

printf("Main process exiting\n");

}

**Output**

kanti@kanti-linux:~/Assgn\_2/TCP\_fork\_server\_client$ ./server 10000

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Concurrent TCP File Server Program: listening on socket 10000..

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Child processes forked on demand.One per client connection:

New connection request from 127.0.0.1:39328 ==>

forking new child process 31338 to handle the client..

File request from client: two\_k\_file.txt

File transfer done.

31338 ==> Closing client connection. Child exiting..

New connection request from 127.0.0.1:39330 ==>

forking new child process 31355 to handle the client..

File request from client: one\_k\_file.txt

File transfer done.

31355 ==> Closing client connection. Child exiting..

New connection request from 127.0.0.1:39332 ==>

forking new child process 31362 to handle the client..

File request from client: one\_k\_frle.txt

File does not exist. Error msg sent back...

Closing client connection. child exiting...

**1.3 Concurrent multithreading TCP File server with one thread per request**

**Readme.txt**

Steps to execute the program:  
  
1. Open the terminal  
2. Navigate to the program directory  
3. Build server & client:  
 make MakeFile.txt - for server  
 make MakeFileClient.txt - for client  
4. Run server and client on two different terminals:  
 ./server <port>  
 ex: ./server 10000   
  
 ./client <server\_IP> <port>  
 ex: ./client 127.0.0.1 10000  
  
5. Enter ^C to quit running the server.

**Makefile**

all:  
 gcc -pthread -o server1 multithreaded\_tcp\_server\_new.c  
clean:  
 rm -f server1

**Source files**

TCP server program

/ CMPE\_207 Assignment\_2  
// Author - Team #2  
// TCP server program  
// This is a concurrent TCP server program. Server   
// listens for active clients.   
// To compile: gcc tcp\_server -o server  
// To run: ./server port\_number  
  
#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<sys/socket.h>  
#include<sys/types.h>  
#include<netinet/in.h>  
#include<string.h>  
#include<arpa/inet.h>  
#include<netdb.h>  
#include<sys/wait.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
  
#define ERROR 0  
#define MAX\_DATA 1024  
  
  
//Handler function by the threads created  
void\* file\_transfer(void\* new\_sock)  
{  
 int n,result;  
 int data\_len;  
 char data[MAX\_DATA];  
 char line[MAX\_DATA];  
 char err\_msg[32] = "";  
 int fd;  
 int clientfd=\*(int\*)new\_sock;  
  
 memset(line, 0, MAX\_DATA);  
 /\* Read a line from the client socket ... \*/  
 n = readline(clientfd, line, MAX\_DATA);  
 if (n <= 0)   
 {  
 printf("Error reading from client");  
 exit(-1);  
 }  
  
 printf("File request from client: %s\n", line);  
 line[n-1] = '\0';  
 result = strcmp(line, "one\_k\_file.txt");  
 if(!result)   
 {   
 //line matches "test\_file\_1.txt"   
 fd = open("one\_k\_file.txt", O\_RDONLY);  
 }   
 else   
 {  
  
 result = strcmp(line, "two\_k\_file.txt");  
 if(!result)  
 {   
 //line matches "test\_file\_2.txt"  
 fd = open("two\_k\_file.txt", O\_RDONLY);  
 }  
 else  
 {  
 printf("File does not exist. Error msg sent back...\n");  
 strcpy(data, "Error: Requested File does not exist on the server.\n");  
 if(0 > send(clientfd, data, strlen(data), 0))  
 {  
 perror("send error");  
 exit(-1);  
 }  
 printf("Closing client connection. Thread exits.\n");  
 close(clientfd);  
 exit(0);  
 }  
 }   
  
 if (fd < 0)  
 {  
 printf("File open Error..\n");  
 close(clientfd);  
 exit(-1);  
 }  
  
 // set pointer to beginning of the file.  
 lseek(fd, 0, SEEK\_SET);  
 while(1)  
 {  
 //read MAX\_DATA data bytes from the file  
 result = readline(fd, data, MAX\_DATA);  
 //printf("File read result = %d bytes\n", result);  
 if(result == 0)  
 break;  
  
 /\*send MAX\_DATA bytes.\*/  
 result = send(clientfd, data, result, 0);  
 //printf("send result = %d bytes\n", result);  
 if (0 > result)  
 {  
 perror("Send Error");  
 break;  
 }  
 }//end inner while  
  
 //close file descriptor  
 close(fd);  
  
 if(result == 0)  
 {  
 printf("File transfer done.\n");  
 printf("\n%d ==> Closing client connection. Child exiting..\n\n", getpid());  
 close(clientfd);  
 exit(0);  
 }//end else  
  
   
  
  
}  
  
int passivesock(const char \*service, const char \*transport, int qlen, int \*sock)  
{  
 struct servent \*pse;  
 struct protoent \*ppe;  
 struct sockaddr\_in server;  
 int res = 1;  
 int type;  
  
 //reset socket address structure.  
 memset(&server,0,sizeof(server));  
  
 server.sin\_family = AF\_INET;  
 server.sin\_addr.s\_addr = INADDR\_ANY;  
  
 //map service name to port number  
 if(pse = getservbyname(service, transport))  
 server.sin\_port = htons(ntohs((u\_short)pse->s\_port));  
 else if((server.sin\_port = htons((u\_short)atoi(service))) == 0)  
 //perror("Cant get \"%s\" service entry\n", service);  
 perror("Cant get service entry\n");  
  
 bzero(&server.sin\_zero, 8);  
  
 //map transport protocol name to protocol number  
 ppe = getprotobyname(transport);  
 if((ppe == NULL))  
 //perror("cant get \"%s\" protocol entry\n", transport);  
 perror("Cant get protocol entry\n");  
  
 //use protocol to chose a socket type  
 if(strcmp(transport, "udp") == 0)  
 type = SOCK\_DGRAM;  
 else  
 type = SOCK\_STREAM;  
  
 //allocate a socket  
 \*sock = socket(PF\_INET, type, ppe->p\_proto);  
 if (\*sock < 0){  
 res = -1;  
 perror("Cant create socket\n");  
 }  
  
 //Bind the socket  
 if(bind(\*sock, (struct sockaddr \*)&server, sizeof(server)) < 0)  
 {  
 res = -1;  
 perror("Cant bind to specified port\n");  
 }  
  
 //listen for incoming client connections  
 if(type == SOCK\_STREAM && (listen(\*sock, qlen) < 0))  
 {   
 res = -1;  
 perror("Listen error\n");  
 }  
  
 return res;  
}  
  
/\*\*  
 \* Simple utility function that reads a line from a file descriptor fd,   
 \* up to maxlen bytes -- ripped from Unix Network Programming, Stevens.  
 \*/  
 int  
readline(int fd, char \*buf, int maxlen)  
{  
 int n, rc;  
 char c;  
  
 for (n = 1; n < maxlen; n++) {  
 rc = read(fd, &c, 1);  
  
 if (rc == 1) {  
 \*buf++ = c;  
 if (c == '\n')  
 break;  
   
 } else if (rc == 0) {  
 if (n == 1)  
 return 0; // EOF, no data read  
 else  
 break; // EOF, read some data  
 } else  
 return -1; // error  
 }  
  
 \*buf = '\0'; // null-terminate  
 return n;  
}  
  
  
int main(int argc, char \*\*argv)  
{  
 struct sockaddr\_in client; //client data structure  
  
 int n, sock, result, len,\*new\_sock;  
 int clientfd;  
 int num;  
 int data\_len;  
 char data[MAX\_DATA];  
 char line[MAX\_DATA];  
 char err\_msg[32] = "";  
 int fd;  
  
 if(argc != 2)  
 {  
 printf("Usage: ./server <port>\n");  
 exit(0);  
 }  
  
 result = passivesock(argv[1], "tcp", MAX\_DATA, &sock);  
  
 if (result == -1)  
 exit(-1);  
  
 printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("Concurrent TCP File Server Program: listening on socket %s..\n", argv[1]);  
 printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("Threads created based on the number of connections recieved: \n\n");   
 while (1)  
 {   
 socklen\_t sockaddr\_len = sizeof(struct sockaddr\_in);  
 socklen\_t client\_t=sizeof(struct sockaddr\_in);  
 memset(&client, 0, sizeof(client));  
 clientfd = accept(sock, (struct sockaddr\*)&client, &client\_t);  
 if ( clientfd < 0 ) {  
 perror("Accept Error");  
 exit(-1);  
 }  
 printf("New connection request from %s:%d ==> \n",inet\_ntoa(client.sin\_addr), ntohs(client.sin\_port));  
 //Create a Pthread with a thread\_id  
 pthread\_t thread\_id;  
 new\_sock = malloc(1);  
 \*new\_sock = (int)clientfd;  
  
 //Check whether the created thread contains error  
 if (pthread\_create(&thread\_id,NULL,file\_transfer,new\_sock) < 0){  
 sleep(0.05);  
 close(clientfd);  
 perror("thread error");  
 exit(-1);  
 }  
   
 else  
 {  
 // we are in thread  
 printf("Child thread created\n");  
 }  
 }  
  
 //close the server socket  
 close(sock);  
 printf("Main process exiting\n");  
}

**Output**

**ubuntu@ubuntu:~/Documents/Multithreaded-TCP$ ./server1 10000  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Concurrent TCP File Server Program: listening on socket 10000..  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Threads created based on the number of connections recieved:   
  
New connection request from 127.0.0.1:38008 ==>   
Child thread created  
File request from client: one\_k\_file.txt  
  
File transfer done.  
  
8806 ==> Closing client connection. Child exiting..**

**1.4 Concurrent pre-forked TCP multiprocessing server**

**Readme.txt**

Steps to execute the program:

1. Open the terminal

2. Navigate to the program directory

3. Build server & client:

make all

Build server alone:

make server

Build client alone:

make client

Clean all targets:

make clean

5. Run server and client on two different terminals:

./server <port> <number of child processes>

ex: ./server 10000 6

./client <server\_IP> <port>

ex: ./client 127.0.0.1 10000

6. Enter ^C to quit running the server.

TCP Server: Server program creates N(given by the user as command line argument) child processes in advance (process pool)to handle the incoming client request. Each child waits for the incoming connection by calling accept() system call. One of the children accepts the connection when a new connection is requested from the client and serves the client. child process returns back to the pool once client connection is served and disconnected.

TCP test client: client connects the server and receives number of preforked children at server side and prints it to the screen.

**Makefile**

# Make file targets to build TCP client & TCP server (Assignment 1, problem #3)

# Format

# target: dependencies

# action

.PHONY: server

all: server

server:

gcc tcp\_server.c -o server

clean:

rm server

**Source files**

**Tcp\_server.c**

// CMPE\_207 Assignment\_2

// Author - Team #2

// TCP server program

// This is a concurrent TCP server program. Server

// listens for active clients. server creates few child

// processes in advance to handle incoming connections.

// child process handle client connections and returns

// to the process pool once done serving the client.

// To compile: gcc tcp\_server -o server

// To run: ./server port\_number

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/socket.h>

#include<sys/types.h>

#include<netinet/in.h>

#include<string.h>

#include<arpa/inet.h>

#include<netdb.h>

#include<sys/wait.h>

#define ERROR 0

#define MAX\_CLIENTS 5

#define MAX\_DATA 1024

int passivesock(const char \*service, const char \*transport, int qlen, int \*sock)

{

struct servent \*pse;

struct protoent \*ppe;

struct sockaddr\_in server;

int res = 1;

int type;

//reset socket address structure.

memset(&server,0,sizeof(server));

server.sin\_family = AF\_INET;

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

//map service name to port number

if(pse = getservbyname(service, transport))

server.sin\_port = htons(ntohs((u\_short)pse->s\_port));

else if((server.sin\_port = htons((u\_short)atoi(service))) == 0)

//perror("Cant get \"%s\" service entry\n", service);

perror("Cant get service entry\n");

bzero(&server.sin\_zero, 8);

//map transport protocol name to protocol number

ppe = getprotobyname(transport);

if((ppe == NULL))

//perror("cant get \"%s\" protocol entry\n", transport);

perror("Cant get protocol entry\n");

//use protocol to chose a socket type

if(strcmp(transport, "udp") == 0)

type = SOCK\_DGRAM;

else

type = SOCK\_STREAM;

//allocate a socket

\*sock = socket(PF\_INET, type, ppe->p\_proto);

if (\*sock < 0){

res = -1;

perror("Cant create socket\n");

}

//Bind the socket

if(bind(\*sock, (struct sockaddr \*)&server, sizeof(server)) < 0)

{

res = -1;

perror("Cant bind to specified port\n");

}

//listen for incoming client connections

if(type == SOCK\_STREAM && (listen(\*sock, qlen) < 0))

{

res = -1;

perror("Listen error\n");

}

return res;

}

int main(int argc, char \*\*argv)

{

struct sockaddr\_in client; //client data structure

int n, sock, result;

pid\_t cpid;

int clientfd;

socklen\_t sockaddr\_len = sizeof(struct sockaddr\_in);

socklen\_t client\_t;

char line[MAX\_DATA];

Int NUM\_CHILDREN = 1;

if(argc != 3)

{

printf("Usage: ./server <port> <number of child processes>\n");

exit(0);

}

NUM\_CHILDREN = atoi(argv[2]);

//create server socket in listening mode.

result = passivesock(argv[1], "tcp", MAX\_CLIENTS, &sock);

if (result == -1)

exit(-1);

printf("\nConcurrent preforking TCP ECHO Server Program: listening on socket %s..\n", argv[1]);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Number of child processes preforked: %d\n\n", NUM\_CHILDREN);

/\* Fork NUM\_CHILDREN child processes to handle client requests. \*/

for (int i = 0; i < NUM\_CHILDREN; i++)

{

cpid = fork();

if (cpid == -1) {

perror("Couldn't fork");

exit(-1);

}

if (cpid == 0) { // We're in the child ...

for (;;) { // Run forever ...

/\* Necessary initialization for accept(2) \*/

client\_t = sizeof(client);

/\*child Blocks! \*/

clientfd = accept(sock, (struct sockaddr \*)&client, &client\_t);

if (clientfd == -1) {

perror("Couldn't accept a connection");

}

printf("\nChild process:%d accepted new client connection from %s\n\n", getpid(),inet\_ntoa(client.sin\_addr));

//send back response to the client

sprintf(line, "\nNumber of preforked processes at is server is %d\n", NUM\_CHILDREN);

if(0 > send(clientfd, line,MAX\_DATA, 0))

{

perror("send error");

exit(-1);

}

//close connection and return to the process pool.

printf("\nChild: %d closing the client connection. Returning to the process pool... \n", getpid());

close(clientfd);

}

}

}

/\*parent: Sit back and wait for all child processes to exit \*/

while (waitpid(-1, NULL, 0) > 0);

//close the server socket

close(sock);

printf("Main process exiting\n");

}

**Output**

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/TCP\_prefork\_server\_client$ ./server 10000

Concurrent preforking TCP ECHO Server Program: listening on socket 10000..

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Number of child processes preforked: 5

Child process:17735 accepted new client connection from 127.0.0.1

Child: 17735 closing the client connection. Returning to the process pool...

**1.5 Concurrent pre-threaded TCP multithreading server**

**Readme.txt**

Steps to execute the program:  
  
1. Open the terminal  
2. Navigate to the program directory  
3. Build server & client:  
 make MakeFile1.txt - for server  
 make MakeFile1Client.txt - for client  
4. Run server and client on two different terminals:  
 ./server <port> <Number of threads>  
 ex: ./server 10000 4  
  
 ./client <server\_IP> <port>  
 ex: ./client 127.0.0.1 10000  
  
5. Enter ^C to quit running the server.

**Makefile**

all:  
 gcc -pthread -o server2 prethreaded\_tcp\_server\_new.c  
clean:  
 rm -f server2

**Source files**

// CMPE\_207 Assignment\_2  
// Author - Team #2  
// TCP server program  
// This is a concurrent TCP server program. Server   
// listens for active clients.   
// To compile: gcc tcp\_server -o server  
// To run: ./server port\_number  
  
#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<sys/socket.h>  
#include<sys/types.h>  
#include<netinet/in.h>  
#include<string.h>  
#include<arpa/inet.h>  
#include<netdb.h>  
#include<sys/wait.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
  
#define ERROR 0  
#define MAX\_DATA 1024  
#define MAX\_THREAD 10  
  
  
//Handler function by the threads created  
void\* file\_transfer(void\* new\_sock)  
{  
 int n,result;  
 int data\_len;  
 char data[MAX\_DATA];  
 char line[MAX\_DATA];  
 char err\_msg[32] = "";  
 int fd;  
 int clientfd=\*(int\*)new\_sock;  
  
 memset(line, 0, MAX\_DATA);  
 /\* Read a line from the client socket ... \*/  
 n = readline(clientfd, line, MAX\_DATA);  
 if (n <= 0)   
 {  
 printf("Error reading from client");  
 exit(-1);  
 }  
  
 printf("File request from client: %s\n", line);  
 line[n-1] = '\0';  
 result = strcmp(line, "one\_k\_file.txt");  
 if(!result)   
 {   
 //line matches "test\_file\_1.txt"   
 fd = open("one\_k\_file.txt", O\_RDONLY);  
 }   
 else   
 {  
  
 result = strcmp(line, "two\_k\_file.txt");  
 if(!result)  
 {   
 //line matches "test\_file\_2.txt"  
 fd = open("two\_k\_file.txt", O\_RDONLY);  
 }  
 else  
 {  
 printf("File does not exist. Error msg sent back...\n");  
 strcpy(data, "Error: Requested File does not exist on the server.\n");  
 if(0 > send(clientfd, data, strlen(data), 0))  
 {  
 perror("send error");  
 exit(-1);  
 }  
 printf("Closing client connection. Thread exits.\n");  
 close(clientfd);  
 exit(0);  
 }  
 }   
  
 if (fd < 0)  
 {  
 printf("File open Error..\n");  
 close(clientfd);  
 exit(-1);  
 }  
  
 // set pointer to beginning of the file.  
 lseek(fd, 0, SEEK\_SET);  
 while(1)  
 {  
 //read MAX\_DATA data bytes from the file  
 result = readline(fd, data, MAX\_DATA);  
 //printf("File read result = %d bytes\n", result);  
 if(result == 0)  
 break;  
  
 /\*send MAX\_DATA bytes.\*/  
 result = send(clientfd, data, result, 0);  
 //printf("send result = %d bytes\n", result);  
 if (0 > result)  
 {  
 perror("Send Error");  
 break;  
 }  
 }//end inner while  
  
 //close file descriptor  
 close(fd);  
  
 if(result == 0)  
 {  
 printf("File transfer done.\n");  
 printf("\n%d ==> Closing client connection. Child exiting..\n\n", getpid());  
 close(clientfd);  
 exit(0);  
 }//end else  
  
   
  
  
}  
  
int passivesock(const char \*service, const char \*transport, int qlen, int \*sock)  
{  
 struct servent \*pse;  
 struct protoent \*ppe;  
 struct sockaddr\_in server;  
 int res = 1;  
 int type;  
  
 //reset socket address structure.  
 memset(&server,0,sizeof(server));  
  
 server.sin\_family = AF\_INET;  
 server.sin\_addr.s\_addr = INADDR\_ANY;  
  
 //map service name to port number  
 if(pse = getservbyname(service, transport))  
 server.sin\_port = htons(ntohs((u\_short)pse->s\_port));  
 else if((server.sin\_port = htons((u\_short)atoi(service))) == 0)  
 //perror("Cant get \"%s\" service entry\n", service);  
 perror("Cant get service entry\n");  
  
 bzero(&server.sin\_zero, 8);  
  
 //map transport protocol name to protocol number  
 ppe = getprotobyname(transport);  
 if((ppe == NULL))  
 //perror("cant get \"%s\" protocol entry\n", transport);  
 perror("Cant get protocol entry\n");  
  
 //use protocol to chose a socket type  
 if(strcmp(transport, "udp") == 0)  
 type = SOCK\_DGRAM;  
 else  
 type = SOCK\_STREAM;  
  
 //allocate a socket  
 \*sock = socket(PF\_INET, type, ppe->p\_proto);  
 if (\*sock < 0){  
 res = -1;  
 perror("Cant create socket\n");  
 }  
  
 //Bind the socket  
 if(bind(\*sock, (struct sockaddr \*)&server, sizeof(server)) < 0)  
 {  
 res = -1;  
 perror("Cant bind to specified port\n");  
 }  
  
 //listen for incoming client connections  
 if(type == SOCK\_STREAM && (listen(\*sock, qlen) < 0))  
 {   
 res = -1;  
 perror("Listen error\n");  
 }  
  
 return res;  
}  
  
/\*\*  
 \* Simple utility function that reads a line from a file descriptor fd,   
 \* up to maxlen bytes -- ripped from Unix Network Programming, Stevens.  
 \*/  
 int  
readline(int fd, char \*buf, int maxlen)  
{  
 int n, rc;  
 char c;  
  
 for (n = 1; n < maxlen; n++) {  
 rc = read(fd, &c, 1);  
  
 if (rc == 1) {  
 \*buf++ = c;  
 if (c == '\n')  
 break;  
   
 } else if (rc == 0) {  
 if (n == 1)  
 return 0; // EOF, no data read  
 else  
 break; // EOF, read some data  
 } else  
 return -1; // error  
 }  
  
 \*buf = '\0'; // null-terminate  
 return n;  
}  
  
  
int main(int argc, char \*\*argv)  
{  
 struct sockaddr\_in client; //client data structure  
  
 int n, sock, result, len,\*new\_sock;  
 int clientfd;  
 int num;  
 int data\_len;  
 char data[MAX\_DATA];  
 char line[MAX\_DATA];  
 char err\_msg[32] = "";  
 int fd;  
 int num\_of\_threads = atoi(argv[2]);  
 pthread\_t thread\_id[num\_of\_threads];  
  
 if(argc != 3)  
 {  
 printf("Usage: ./server <port>\n");  
 exit(0);  
 }  
  
 result = passivesock(argv[1], "tcp", MAX\_DATA, &sock);  
  
 if (result == -1)  
 exit(-1);  
  
 printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("Concurrent TCP File Server Program: listening on socket %s..\n", argv[1]);  
 printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("Threads created based on the number of connections recieved: \n\n");   
 while (1)  
 {   
 socklen\_t sockaddr\_len = sizeof(struct sockaddr\_in);  
 socklen\_t client\_t=sizeof(struct sockaddr\_in);  
 memset(&client, 0, sizeof(client));  
 clientfd = accept(sock, (struct sockaddr\*)&client, &client\_t);  
 if ( clientfd < 0 ) {  
 perror("Accept Error");  
 exit(-1);  
 }  
 printf("New connection request from %s:%d ==> \n",inet\_ntoa(client.sin\_addr), ntohs(client.sin\_port));  
 new\_sock = malloc(1);  
 \*new\_sock = (int)clientfd;  
 //Create a Pthread with a thread\_id  
 for (int i = 0; i < num\_of\_threads ; i++)  
 {  
 //Check whether the created thread contains error  
 if (pthread\_create(&thread\_id[i],NULL,file\_transfer,new\_sock) < 0){  
 sleep(0.05);  
 //close(clientfd);  
 perror("thread error");  
 exit(-1);  
 }  
   
 else  
 {  
 // we are in thread  
 printf("Child thread created %d\n", i);  
 }  
 }  
 for (int i = 0; i < num\_of\_threads; i++)  
 {  
 pthread\_join(thread\_id[i],NULL);  
 }  
  
 }  
  
 //close the server socket  
 close(sock);  
 printf("Main process exiting\n");  
}

**Output**

ubuntu@ubuntu:~/Documents/Prethreaded-TCP$ ./server2 10000 4  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Concurrent TCP File Server Program: listening on socket 10000..  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Threads created based on the number of connections recieved:   
  
New connection request from 127.0.0.1:38012 ==>   
Child thread created 0  
Child thread created 1  
Child thread created 2  
Child thread created 3  
File request from client: two\_k\_file.txt  
  
File transfer done.  
  
8889 ==> Closing client connection. Child exiting..

**2. Connectionless servers:**

**2.1 Iterative UDP File server**

**Readme.txt**

1. To Open the editor:   
gedit udp\_iterative\_server.c  
  
2. To Compile to program

make all  
3. To Execute the program  
./udp\_iterative\_server

**Makefile**

.PHONY: server

all: server

server:

gcc udp\_iterative\_server.c -o udp\_iterative\_server

clean:

rm server

**Source files**

/\*  
UDP iterative server program by group-02  
Make file, code, output are provided  
Reference: Internetworking with TCP/IP Vol. 3, Client-Server programming and applications, Comer and Stevens, Linux/POSIX Sockets version, ISBN: 0-13-032071-4, 2001.  
\*/  
#include <errno.h>  
#include <netinet/in.h>  
#include <time.h>  
#include <unistd.h>  
#include <stdarg.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <arpa/inet.h>  
#include <netdb.h>  
#include <string.h>  
#include <fcntl.h>  
  
#define \_\_USE\_BSD 1  
#define UNIXEPOCH 2208988800   
int passiveUDP(const char \*service);  
#ifndef INADDR\_NONE  
#define INADDR\_NONE 0xffffffff  
#endif /\* INADDR\_NONE \*/  
typedef unsigned long u\_long;  
typedef unsigned short u\_short;  
extern int errno;  
int passivesock(const char \*service, const char \*transport, int qlen);  
int errexit(const char \*format, ...);  
u\_short portbase = 0;  
/\* port base, for non-root servers  
\*/  
/\*------------------------------------------------------------------------  
\* passivesock - allocate & bind a server socket using TCP or UDP  
\*------------------------------------------------------------------------  
\*/  
#define LINELEN 128  
  
int passiveTCP(const char \*service,int qlen);  
int errexit(const char \*format, ...);  
int passivesock(const char \*service, const char \*transport,int qlen);  
int TCPdaytimed(int fd);  
#define QLEN 5  
/\*------------------------------------------------------------------------  
\* main - Iterative UDP server\*------------------------------------------------------------------------  
\*/  
int main(int argc, char \*argv[])  
{  
struct sockaddr\_in fsin;  
char \*service = "echo";  
char buffer[10000];  
int msock,ssock;  
unsigned int alen;  
int fil;  
size\_t rd,wr;  
  
//printf("Before switch \n");  
switch (argc) {  
case 1:  
break;  
case 2:  
service = argv[1];  
break;  
default:  
errexit("usage: UDP [port]\n");  
}  
  
//printf("After Switch \n");  
//fflush(stdout);  
msock = passiveUDP(service);  
while (1) {  
printf("Waiting for the client \n");  
//fflush(stdout);  
//ssock = accept(msock, (struct sockaddr \*)&fsin, &alen);  
//ssock = 2;  
//if (ssock < 0)  
//errexit("accept failed: %s\n", strerror(errno));  
  
//fflush(stdout);  
int file\_n=recvfrom(msock,buffer,10000,0,(struct sockaddr \*)&fsin, &alen);  
  
printf("File is opened in read only mode\n");  
fil=open(buffer,O\_RDONLY);  
//printf("%s",buffer);  
while((rd=read(fil,buffer,2048))>0)  
{  
wr=sendto(msock,buffer,rd,0,(struct sockaddr \*)&fsin,sizeof(fsin));  
printf("The data from the file is sent to the client\n\n");  
}  
  
(void) close(ssock);  
}  
  
}  
  
  
  
/\* passiveUDP.c - passiveUDP \*/  
  
/\*------------------------------------------------------------------------  
\* passiveUDP - create a passive socket for use in a UDP server  
\*------------------------------------------------------------------------  
\*/  
int passiveUDP(const char \*service)  
/\*  
\* Arguments:  
\*  
service - service associated with the desired port  
\*/  
{  
return passivesock(service, "udp", 0);  
}  
  
  
int passivesock(const char \*service, const char \*transport, int qlen)  
/\*  
\* Arguments:  
\* service  
- service associated with the desired port  
\* transport - transport protocol to use ("tcp" or "udp")  
\* qlen  
- maximum server request queue length  
\*/  
{  
struct servent \*pse; /\* pointer to service information entry \*/  
struct protoent \*ppe; /\* pointer to protocol information entry\*/  
struct sockaddr\_in sin; /\* an Internet endpoint address\*/  
int s, type;  
/\* socket descriptor and socket type  
\*/  
memset(&sin, 0, sizeof(sin));  
sin.sin\_family = AF\_INET;  
sin.sin\_addr.s\_addr = INADDR\_ANY;  
/\* Map service name to port number \*/  
if ( pse = getservbyname(service, transport) )  
sin.sin\_port = htons(10026);  
else if ( (sin.sin\_port = htons((u\_short)atoi(service))) == 0 )  
errexit("can't get \"%s\" service entry\n", service);  
/\* Map protocol name to protocol number \*/  
if ( (ppe = getprotobyname(transport)) == 0)  
errexit("can't get \"%s\" protocol entry\n", transport);  
/\* Use protocol to choose a socket type \*/  
if (strcmp(transport, "udp") == 0)  
type = SOCK\_DGRAM;  
else  
type = SOCK\_STREAM;  
/\* Allocate a socket \*/  
s = socket(PF\_INET, type, ppe->p\_proto);  
if (s < 0)  
errexit("can't create socket: %s\n", strerror(errno));  
/\* Bind the socket \*/  
if (bind(s, (struct sockaddr \*)&sin, sizeof(sin)) < 0)  
errexit("can't bind to %s port: %s\n", service,  
strerror(errno));  
if (type == SOCK\_STREAM && listen(s, qlen) < 0)  
errexit("can't listen on %s port: %s\n", service,  
strerror(errno));  
return s;  
}  
  
  
int errexit(const char \*format, ...)  
{  
va\_list args;  
va\_start(args, format);  
vfprintf(stderr, format, args);  
va\_end(args);  
exit(1);  
}

**Output**

ubuntu@ubuntu:~/UDP\_server$ ./udp\_iterative\_server

Waiting for the client

File is opened in read only mode

The data from the file is sent to the client

Waiting for the client

File is opened in read only mode

The data from the file is sent to the client

Waiting for the client

File is opened in read only mode

The data from the file is sent to the client

Waiting for the client

**2.2 Concurrent multiprocessing UDP File server with one process per request**

**Readme.txt**

Steps to execute the program:

1. Open the terminal

2. Navigate to the program directory

3. Build server & client:

make all

Build server alone:

make server

Build client alone:

make client

Clean all targets:

make clean

5. Run server and client on two different terminals:

./server <port>

ex: ./server 10000

./client <server\_IP> <port>

ex: ./client 127.0.0.1 10000

6. Enter ^C to quit running the server.

UDP File Server:Server maintains 2 files "one\_k\_file.txt" and "two\_k\_file.txt". Server program creates child processes on demand to handle the incoming client request.When client sends a file transfer request, server validates the file name and transfer the file. If File name sent

by the client does not match with server files then an error message is sent back to the client.

UDP test client:A File name is passed to the client as a command line argument. Client sends a datagram to the server requesting this file

from the server. Client then waits for the file transfer. It opens "test\_file.txt" file and writes messages transfered by the server into

this file. Then closes the connection.

**Makefile**

# Make file targets to build server and clients

# Format

# target: dependencies

# action

.PHONY: server

all: server

server:

gcc udp\_server.c -o server

clean:

rm server

**Source files**

**Udp\_server.c**

// CMPE\_207 Assignment\_2

// Author - Team #2

// UDP server program

// This is a concurrent UDP server program. Server

// listens for active clients. It creates new child

// processes to handle each new client connection.

// child exits once client is served. Parent process

// waits for new connections in an infinite loop.

// To compile: gcc udp\_server.c -o server

// To run: ./server port\_number

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/socket.h>

#include<sys/types.h>

#include<netinet/in.h>

#include<string.h>

#include<arpa/inet.h>

#include<netdb.h>

#include<sys/wait.h>

#include<sys/stat.h>

#include<fcntl.h>

#define ERROR 0

#define MAX\_DATA 1024

#define NUM\_CHILDREN 5

int passivesock(const char \*service, const char \*transport, int \*sock)

{

struct servent \*pse;

struct protoent \*ppe;

struct sockaddr\_in server;

int res = 1;

int type;

//reset socket address structure.

memset(&server,0,sizeof(server));

server.sin\_family = AF\_INET;

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

//map service name to port number

if(pse = getservbyname(service, transport))

server.sin\_port = htons(ntohs((u\_short)pse->s\_port));

else if((server.sin\_port = htons((u\_short)atoi(service))) == 0)

//perror("Cant get \"%s\" service entry\n", service);

perror("Cant get service entry\n");

bzero(&server.sin\_zero, 8);

//map transport protocol name to protocol number

ppe = getprotobyname(transport);

if((ppe == NULL))

//perror("cant get \"%s\" protocol entry\n", transport);

perror("Cant get protocol entry\n");

//use protocol to chose a socket type

if(strcmp(transport, "udp") == 0)

type = SOCK\_DGRAM;

else

type = SOCK\_STREAM;

//allocate a socket

\*sock = socket(PF\_INET, type, ppe->p\_proto);

if (\*sock < 0){

res = -1;

perror("Cant create socket\n");

}

//Bind the socket

if(bind(\*sock, (struct sockaddr \*)&server, sizeof(server)) < 0)

{

res = -1;

perror("Cant bind to specified port\n");

}

return res;

}

/\*\*

\* Simple utility function that reads a line from a file descriptor fd,

\* up to maxlen bytes -- ripped from Unix Network Programming, Stevens.

\*/

int

readline(int fd, char \*buf, int maxlen)

{

int n, rc;

char c;

for (n = 1; n < maxlen; n++) {

rc = read(fd, &c, 1);

if (rc == 1) {

\*buf++ = c;

if (c == '\n')

break;

} else if (rc == 0) {

if (n == 1)

return 0; // EOF, no data read

else

break; // EOF, read some data

} else

return -1; // error

}

\*buf = '\0'; // null-terminate

return n;

}

int main(int argc, char \*\*argv)

{

struct sockaddr\_in client; //client data structure

int bytes\_received = 0;

int n, sock, result, len;

int clientfd;

int num;

pid\_t cpid;

int data\_len;

char data[MAX\_DATA];

char parent\_line[MAX\_DATA];

char child\_line[MAX\_DATA];

char cpid\_s[32];

char err\_msg[32] = "";

int fd;

if(argc != 2)

{

printf("Usage: ./server <port>\n");

exit(0);

}

result = passivesock(argv[1], "udp", &sock);

if (result == -1)

exit(-1);

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Concurrent UDP File Server Program: listening on socket %s..\n", argv[1]);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Child processes forked on demand.One per client connection: \n\n");

while (1)

{

//socklen\_t sockaddr\_len = sizeof(struct sockaddr\_in);

socklen\_t client\_t=sizeof(struct sockaddr\_in);

memset(&client, 0, sizeof(client));

memset(parent\_line, 0, MAX\_DATA);

memset(child\_line,0, MAX\_DATA);

//parent: wait for incoming client datagram

bytes\_received = recvfrom(sock, parent\_line, MAX\_DATA, 0, (struct sockaddr \*)&client, &client\_t);

if(0 > bytes\_received)

{

perror("recvfrom error");

close(sock);

exit(-1);

}

strncpy(child\_line,parent\_line,strlen(parent\_line));

cpid = fork();

if (cpid != 0){

sleep(0.05);

if(cpid == -1){

perror("fork error");

exit(-1);

}

}

else

{

// we are in child

printf("forking new child process %d to handle the client %s:%d \n",getpid(),inet\_ntoa(client.sin\_addr),client.sin\_port);

printf("File request from client: %s \n", child\_line);

//n = strlen(child\_line);

//child\_line[n-1] = '\0';

result = strcmp(child\_line, "one\_k\_file.txt");

if(!result) { //line matches "test\_file\_1.txt"

printf("File name matched: one\_k\_file.txt\n");

fd = open("one\_k\_file.txt", O\_RDONLY);

} else {

result = strcmp(child\_line, "two\_k\_file.txt");

if(!result){ //line matches "test\_file\_2.txt"

printf("File name matched: two\_k\_file.txt");

fd = open("two\_k\_file.txt", O\_RDONLY);

}else{

printf("%s ==> File does not exist on the server. \n",child\_line);

strcpy(data, "Error: Requested File does not exist on the server.\n");

if(0 > sendto(sock, data, strlen(data), 0, (struct sockaddr \*)&client, client\_t))

{

perror("sendto error");

exit(-1);

}

printf("%d: Error msg sent. child exiting...\n", getpid());

exit(0);

}

}

if (fd < 0)

{

printf("File open Error..\n");

exit(-1);

}

// set pointer to beginning of the file.

lseek(fd, 0, SEEK\_SET);

int done = 0;

while(!done)

{

//read MAX\_DATA data bytes from the file

result = readline(fd, data, MAX\_DATA);

//printf("File read result = %d bytes\n", result);

if(result == 0)

{

// Ctrl-D to indicate EOF

data[0]='';

done=1;

result=1;

}

/\*send MAX\_DATA bytes.\*/

result = sendto(sock, data, result, 0, (struct sockaddr \*)&client, client\_t);

//printf("send result = %d bytes\n", result);

if (0 > result)

{

perror("Sendto Error");

break;

}

}//end inner while

//close file

close(fd);

if(done == 1)//if(result == 0)

printf("%d: File transfer done.Child Exiting..\n",getpid());

exit(0);

}//end else

} //end while(1)

//close the server socket

close(sock);

printf("Main process exiting\n");

}

**Output**

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/UDP\_fork\_server\_client$ ./server 10000

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Concurrent UDP File Server Program: listening on socket 10000..

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Child processes forked on demand.One per client connection:

forking new child process 10478 to handle the client 127.0.0.1:18376

File request from client: one\_k\_file.txt

File name matched: one\_k\_file.txt

10478: File transfer done.Child Exiting..

forking new child process 10489 to handle the client 127.0.0.1:4259

File request from client: t\_k\_file.txt

t\_k\_file.txt ==> File does not exist on the server.

10489: Error msg sent. child exiting...

forking new child process 10496 to handle the client 127.0.0.1:35546

File request from client: one\_k\_file.txt

File name matched: one\_k\_file.txt

10496: File transfer done.Child Exiting..

**2.3 Concurrent multithreading UDP File server with one thread per request**

**Readme.txt**

Steps to execute the program:  
  
1. Open the terminal  
2. Navigate to the program directory  
3. Build server & client:  
 make MakeFile2.txt - for server  
 make MakeFile2Client.txt - for client  
  
4. Run server and client on two different terminals:  
 ./server <port>  
 ex: ./server 10000   
  
 ./client <server\_IP> <port>  
 ex: ./client 127.0.0.1 10000  
  
5. Enter ^C to quit running the server.

**Makefile**

all:  
 gcc -pthread -o server3 udp\_server\_thread.c  
clean:  
 rm -f server3

**Source files**

// CMPE\_207 Assignment\_2  
// Author - Team #2  
// UDP server program  
// This is a concurrent UDP server program. Server   
// listens for active clients. It creates new child  
// processes to handle each new client connection.  
// child exits once client is served. Parent process  
// waits for new connections in an infinite loop.  
// To compile: gcc udp\_server.c -o server  
// To run: ./server port\_number  
  
#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<sys/socket.h>  
#include<sys/types.h>  
#include<netinet/in.h>  
#include<string.h>  
#include<arpa/inet.h>  
#include<netdb.h>  
#include<sys/wait.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
#include<pthread.h>  
  
#define ERROR 0  
#define MAX\_DATA 1024  
  
//Create a structure for thread  
struct UDP\_thread  
{  
 socklen\_t len;  
 struct sockaddr\_in from\_client;  
 char buffer[MAX\_DATA];  
 int socket\_new;  
  
};  
  
//Handle request based on the thread creation  
void\* handle\_req(void \*UDP\_thread)  
{  
 struct UDP\_thread \*UDP\_th = UDP\_thread;  
 char child\_line[MAX\_DATA];  
 char data[MAX\_DATA];  
 int sock,result,fd;  
  
 //Copy file name from the client to the buffer child\_line  
 strncpy(child\_line,UDP\_th->buffer,strlen(UDP\_th->buffer));  
  
 printf("File request from client: %s \n", child\_line);  
   
 result = strcmp(child\_line, "one\_k\_file.txt");  
 if(!result)   
 {   
 //line matches "test\_file\_1.txt"   
 printf("File name matched: one\_k\_file.txt\n");  
 fd = open("one\_k\_file.txt", O\_RDONLY);  
 }   
 else   
 {  
  
 result = strcmp(child\_line, "two\_k\_file.txt");  
 if(!result)  
 {   
  
 //line matches "test\_file\_2.txt"  
 printf("File name matched: two\_k\_file.txt");  
 fd = open("two\_k\_file.txt", O\_RDONLY);  
 }  
 else  
 {  
 printf("%s ==> File does not exist on the server. \n",child\_line);  
 strcpy(data, " Error: Requested File does not exist on the server.\n");  
 if(0 > sendto(UDP\_th->socket\_new, data, strlen(data), 0, (struct sockaddr \*)&UDP\_th->from\_client, UDP\_th->len))  
 {  
 perror("sendto error");  
 exit(-1);  
 }  
 printf("Error msg sent. Thread exiting...\n");  
 exit(0);  
 }  
 }   
  
 if (fd < 0)  
 {  
 printf("File open Error..\n");  
 exit(-1);  
 }  
  
 // set pointer to beginning of the file.  
 lseek(fd, 0, SEEK\_SET);  
 int done = 0;  
 while(!done)  
 {  
 //read MAX\_DATA data bytes from the file  
 result = readline(fd, data, MAX\_DATA);  
   
 if(result == 0)  
 {  
 // Ctrl-D to indicate EOF  
 data[0]=' ';   
 done=1;  
 result=1;  
 }  
  
 /\*send MAX\_DATA bytes.\*/  
 result = sendto(UDP\_th->socket\_new, data, result, 0, (struct sockaddr \*)&UDP\_th->from\_client, UDP\_th->len);  
   
 if (0 > result)  
 {  
 perror("Sendto Error");  
 break;  
 }  
 }//end inner while  
  
 //close file  
 close(fd);  
  
 if(done == 1)//if(result == 0)  
 printf("File transfer done.Thread Exiting..\n");  
   
 exit(0);  
  
  
  
}  
  
  
int passivesock(const char \*service, const char \*transport, int \*sock)  
{  
 struct servent \*pse;  
 struct protoent \*ppe;  
 struct sockaddr\_in server;  
 int res = 1;  
 int type;  
  
 //reset socket address structure.  
 memset(&server,0,sizeof(server));  
  
 server.sin\_family = AF\_INET;  
 server.sin\_addr.s\_addr = INADDR\_ANY;  
  
 //map service name to port number  
 if(pse = getservbyname(service, transport))  
 server.sin\_port = htons(ntohs((u\_short)pse->s\_port));  
 else if((server.sin\_port = htons((u\_short)atoi(service))) == 0)  
 //perror("Cant get \"%s\" service entry\n", service);  
 perror("Cant get service entry\n");  
  
 bzero(&server.sin\_zero, 8);  
  
 //map transport protocol name to protocol number  
 ppe = getprotobyname(transport);  
 if((ppe == NULL))  
 //perror("cant get \"%s\" protocol entry\n", transport);  
 perror("Cant get protocol entry\n");  
  
 //use protocol to chose a socket type  
 if(strcmp(transport, "udp") == 0)  
 type = SOCK\_DGRAM;  
 else  
 type = SOCK\_STREAM;  
  
 //allocate a socket  
 \*sock = socket(PF\_INET, type, ppe->p\_proto);  
 if (\*sock < 0){  
 res = -1;  
 perror("Cant create socket\n");  
 }  
  
 //Bind the socket  
 if(bind(\*sock, (struct sockaddr \*)&server, sizeof(server)) < 0)  
 {  
 res = -1;  
 perror("Cant bind to specified port\n");  
 }  
  
 return res;  
}  
  
/\*\*  
 \* Simple utility function that reads a line from a file descriptor fd,   
 \* up to maxlen bytes -- ripped from Unix Network Programming, Stevens.  
 \*/  
int readline(int fd, char \*buf, int maxlen)  
{  
 int n, rc;  
 char c;  
  
 for (n = 1; n < maxlen; n++) {  
 rc = read(fd, &c, 1);  
  
 if (rc == 1) {  
 \*buf++ = c;  
 if (c == '\n')  
 break;  
   
 } else if (rc == 0) {  
 if (n == 1)  
 return 0; // EOF, no data read  
 else  
 break; // EOF, read some data  
 } else  
 return -1; // error  
 }  
  
 \*buf = '\0'; // null-terminate  
 return n;  
}  
  
  
int main(int argc, char \*\*argv)  
{  
 struct sockaddr\_in client; //client data structure  
 int bytes\_received = 0;  
 int n,result, len;  
 int clientfd;  
 int num;  
 pid\_t cpid;  
 int data\_len;  
 char data[MAX\_DATA];  
 char parent\_line[MAX\_DATA];  
 char child\_line[MAX\_DATA];  
 char cpid\_s[32];  
 char err\_msg[32] = "";  
 int fd;  
 struct UDP\_thread u;  
  
 if(argc != 2)  
 {  
 printf("Usage: ./server <port>\n");  
 exit(0);  
 }  
   
 u.len = sizeof(struct sockaddr\_in);  
   
 result = passivesock(argv[1], "udp", &u.socket\_new);  
  
 if (result == -1)  
 exit(-1);  
  
 printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("Concurrent UDP File Server Program: listening on socket %s..\n", argv[1]);  
 printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("Thread is created based on the input from client: \n\n");   
 while (1)  
 {   
   
  
 socklen\_t client\_t=sizeof(struct sockaddr\_in);  
 memset(&client, 0, sizeof(client));  
 memset(parent\_line, 0, MAX\_DATA);  
 memset(child\_line,0, MAX\_DATA);  
  
 //parent: wait for incoming client datagram  
 bytes\_received = recvfrom(u.socket\_new, u.buffer, MAX\_DATA, 0, (struct sockaddr \*)&u.from\_client,&u.len);  
 if(0 > bytes\_received)  
 {  
 perror("recvfrom error");  
 close(u.socket\_new);  
 exit(-1);  
 }  
   
 //Create Pthread ID  
 pthread\_t thread\_id;  
 int return\_c = 0;  
  
 //Create thread based on structure UDP\_Thread  
 return\_c = pthread\_create(&thread\_id,NULL,handle\_req,&u);  
 if (return\_c != 0){  
 sleep(0.05);  
 perror("Error creating thread\n");  
 exit(-1);  
   
 }  
 else  
 {  
 // we are in thread  
 printf("Thread created for the client %s:%d \n", client.sin\_port);  
 }  
 }  
  
   
  
 //close the server socket  
 close(u.socket\_new);  
 printf("Main process exiting\n");  
}

**Output**

ubuntu@ubuntu:~/Documents/UDP\_thread\_server\_client\_old$ ./server3 10000  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Concurrent UDP File Server Program: listening on socket 10000..  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Thread is created based on the input from client:   
  
Thread created for the client (null):0   
File request from client: one\_k\_file.txt

**2.4 Concurrent pre-forked UDP multiprocessing server**

**Readme.txt**

Steps to execute the program:

1. Open the terminal

2. Navigate to the program directory

3. Build server & client:

make all

Build server alone:

make server

Build client alone:

make client

Clean all targets:

make clean

5. Run server and client on two different terminals:

./server <port> <number of child processes>

ex: ./server 10000 5

./client <server\_IP> <port>

ex: ./client 127.0.0.1 10000

6. Enter ^C to quit running the server.

UDP prefork Server: Server program creates N(given by user) child processes in advance (process pool)to handle the incoming client request.When

a datagram is received from the client, active child process at the server sends back the response (total number of child processes

preforked at server side).

UDP test client: client connects the server and receives number of preforked children at server side and prints it to the screen.

**Makefile**

# Make file targets to build UDP server & client

# Format

# target: dependencies

# action

.PHONY: server

all: server

server:

gcc udp\_server.c -o server

clean:

rm server

**Source files**

**Udp\_server.c**

// CMPE\_207 Assignment\_2

// Author - Team #2

// UDP server program

// This is a concurrent UDP server program. Server

// listens for active clients. server creates few child

// processes in advance to handle incoming connections.

// child process handle client connections and returns

// to the process pool once done serving the client.

// To compile: gcc udp\_server -o server

// To run: ./server port\_number

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/socket.h>

#include<sys/types.h>

#include<netinet/in.h>

#include<string.h>

#include<arpa/inet.h>

#include<netdb.h>

#include<sys/wait.h>

#define ERROR 0

#define MAX\_CLIENTS 5

#define MAX\_DATA 1024

int passivesock(const char \*service, const char \*transport, int qlen, int \*sock)

{

struct servent \*pse;

struct protoent \*ppe;

struct sockaddr\_in server;

int res = 1;

int type;

//reset socket address structure.

memset(&server,0,sizeof(server));

server.sin\_family = AF\_INET;

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

//map service name to port number

if((pse = getservbyname(service, transport)))

server.sin\_port = htons(ntohs((u\_short)pse->s\_port));

else if((server.sin\_port = htons((u\_short)atoi(service))) == 0)

//perror("Cant get \"%s\" service entry\n", service);

perror("Cant get service entry\n");

bzero(&server.sin\_zero, 8);

//map transport protocol name to protocol number

ppe = getprotobyname(transport);

if(ppe == NULL)

//perror("cant get \"%s\" protocol entry\n", transport);

perror("Cant get protocol entry\n");

//use protocol to chose a socket type

if(strcmp(transport, "udp") == 0)

type = SOCK\_DGRAM;

else

type = SOCK\_STREAM;

//allocate a socket

\*sock = socket(PF\_INET, type, ppe->p\_proto);

if (\*sock < 0){

res = -1;

perror("Cant create socket\n");

}

//Bind the socket

if(bind(\*sock, (struct sockaddr \*)&server, sizeof(server)) < 0)

{

res = -1;

perror("Cant bind to specified port\n");

}

return res;

}

int main(int argc, char \*\*argv)

{

struct sockaddr\_in client; //client data structure

int n, sock, result;

pid\_t cpid;

int clientfd, bytes\_read;

socklen\_t sockaddr\_len = sizeof(struct sockaddr\_in);

socklen\_t client\_t;

char line[MAX\_DATA];

Int NUM\_CHILDREN = 1;

if(argc != 3)

{

printf("Usage: ./server <port> <number of child processes>\n");

exit(0);

}

NUM\_CHILDREN = atoi(argv[2]);

//create server socket and bind it to an address.

result = passivesock(argv[1], "udp", MAX\_CLIENTS, &sock);

if (result == -1)

exit(-1);

printf("\nConcurrent preforking UDP Server Program \n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Number of child processes preforked: %d\n\n", NUM\_CHILDREN);

/\* Fork NUM\_CHILDREN child processes to handle client requests. \*/

for (int i = 0; i < NUM\_CHILDREN; i++)

{

cpid = fork();

if (cpid == -1) {

perror("Couldn't fork");

exit(-1);

}

if (cpid == 0)

{ // We're in the child ...

printf("Child: %d forked....\n", getpid());

for (;;)

{ // Run forever ...

// wait for datagrams

client\_t = sizeof(struct sockaddr\*);

bytes\_read = recvfrom(sock,line, MAX\_DATA, 0, (struct sockaddr \*)&client, &client\_t);

if(bytes\_read > 0)

{

printf("Child %d: Request received from client: %s:%d\n", getpid(),inet\_ntoa(client.sin\_addr),client.sin\_port);

memset(line, 0, MAX\_DATA);

//send back response to the client

sprintf(line, "\nChild pid %d: Number of preforked processes at server is %d\n",getpid(), NUM\_CHILDREN);

if(0 > sendto(sock,line, strlen(line), 0, (struct sockaddr \*)&client, client\_t))

{

perror("sendto error");

exit(-1);

}

printf("Child %d: response sent to %s:%d\n",getpid(),inet\_ntoa(client.sin\_addr),client.sin\_port);

}

else

perror("recvfrom error");

}

}

}

/\*parent: Sit back and wait for all child processes to exit \*/

while (waitpid(-1, NULL, 0) > 0);

//close the server socket

close(sock);

printf("Main process exiting\n");

}

**Output**

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/UDP\_prefork\_server\_client$ ./server 10000

Concurrent preforking UDP Server Program

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Number of child processes preforked: 7

Child: 4753 forked....

Child: 4755 forked....

Child: 4756 forked....

Child: 4754 forked....

Child: 4757 forked....

Child: 4758 forked....

Child: 4759 forked....

Child 4753: Request received from client: 127.0.0.1:54453

Child 4753: response sent to 127.0.0.1:54453

^C

**2.5 Concurrent pre-threaded UDP multithreading server**

**Readme.txt**

Steps to execute the program:  
  
1. Open the terminal  
2. Navigate to the program directory  
3. Build server & client:  
 make MakeFile3.txt - for server  
 make MakeFile3Client.txt - for client  
4. Run server and client on two different terminals:  
 ./server <port> <Number of threads>  
 ex: ./server 10000 4  
  
 ./client <server\_IP> <port>  
 ex: ./client 127.0.0.1 10000  
  
5. Enter ^C to quit running the server.

**Makefile**

all:  
 gcc -pthread -o server4 udp\_server\_prethreading.c  
clean:  
 rm -f server4

**Source files**

// CMPE\_207 Assignment\_2  
// Author - Team #2  
// UDP server program for threading  
  
#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<sys/socket.h>  
#include<sys/types.h>  
#include<netinet/in.h>  
#include<string.h>  
#include<arpa/inet.h>  
#include<netdb.h>  
#include<sys/wait.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
#include<pthread.h>  
  
#define ERROR 0  
#define MAX\_DATA 1024  
  
  
//Structure to create an instance of thread  
struct UDP\_thread  
{  
 socklen\_t len;  
 struct sockaddr\_in from\_client;  
 char buffer[MAX\_DATA];  
 int socket\_new;  
  
};  
  
//Handle request function to handle the requests from the thread creation  
//Performs the file transfer of the requested file from server to client  
void\* handle\_req(void \*UDP\_thread)  
{  
 struct UDP\_thread \*UDP\_th = UDP\_thread;  
 char child\_line[MAX\_DATA];  
 char data[MAX\_DATA];  
 int sock,result,fd;  
  
 //Copy the file name received from the client by the server to child\_line buffer  
 strncpy(child\_line,UDP\_th->buffer,strlen(UDP\_th->buffer));  
  
 printf("File request from client: %s \n", child\_line);  
   
 //Compare the file names   
 result = strcmp(child\_line, "one\_k\_file.txt");  
 if(!result)   
 {   
 //line matches "test\_file\_1.txt"   
 printf("File name matched: one\_k\_file.txt\n");  
 fd = open("one\_k\_file.txt", O\_RDONLY);  
 }   
 else   
 {  
  
 result = strcmp(child\_line, "two\_k\_file.txt");  
 if(!result)  
 {   
 //line matches "test\_file\_2.txt"  
 printf("File name matched: two\_k\_file.txt");  
 fd = open("two\_k\_file.txt", O\_RDONLY);  
 }  
 else  
 {  
 printf("%s ==> File does not exist on the server. \n",child\_line);  
 strcpy(data, " Error: Requested File does not exist on the server.\n");  
 if(0 > sendto(UDP\_th->socket\_new, data, strlen(data), 0, (struct sockaddr \*)&UDP\_th->from\_client, UDP\_th->len))  
 {  
 perror("sendto error");  
 exit(-1);  
 }  
 printf("Error msg sent. Thread exiting...\n");  
 exit(0);  
 }  
 }   
  
 //Check error on file descriptor fd  
 if (fd < 0)  
 {  
 printf("File open Error..\n");  
 exit(-1);  
 }  
  
 // set pointer to beginning of the file.  
 lseek(fd, 0, SEEK\_SET);  
 int done = 0;  
 while(!done)  
 {  
 //read MAX\_DATA data bytes from the file  
 result = readline(fd, data, MAX\_DATA);  
 if(result == 0)  
 {  
 // Ctrl-D to indicate EOF  
 data[0]=' ';   
 done=1;  
 result=1;  
 }  
  
 /\*send MAX\_DATA bytes.\*/  
 result = sendto(UDP\_th->socket\_new, data, result, 0, (struct sockaddr \*)&UDP\_th->from\_client, UDP\_th->len);  
   
 if (0 > result)  
 {  
 perror("Sendto Error");  
 break;  
 }  
 }//end of while  
  
 //close file descriptor  
 close(fd);  
  
 if(done == 1)//if(result == 0)  
 printf("File transfer done.Thread Exiting..\n");  
   
 exit(0);  
}  
  
  
int passivesock(const char \*service, const char \*transport, int \*sock)  
{  
 struct servent \*pse;  
 struct protoent \*ppe;  
 struct sockaddr\_in server;  
 int res = 1;  
 int type;  
  
 //reset socket address structure.  
 memset(&server,0,sizeof(server));  
  
 server.sin\_family = AF\_INET;  
 server.sin\_addr.s\_addr = INADDR\_ANY;  
  
 //map service name to port number  
 if(pse = getservbyname(service, transport))  
 server.sin\_port = htons(ntohs((u\_short)pse->s\_port));  
 else if((server.sin\_port = htons((u\_short)atoi(service))) == 0)  
 //perror("Cant get \"%s\" service entry\n", service);  
 perror("Cant get service entry\n");  
  
 bzero(&server.sin\_zero, 8);  
  
 //map transport protocol name to protocol number  
 ppe = getprotobyname(transport);  
 if((ppe == NULL))  
 //perror("cant get \"%s\" protocol entry\n", transport);  
 perror("Cant get protocol entry\n");  
  
 //use protocol to chose a socket type  
 if(strcmp(transport, "udp") == 0)  
 type = SOCK\_DGRAM;  
 else  
 type = SOCK\_STREAM;  
  
 //allocate a socket  
 \*sock = socket(PF\_INET, type, ppe->p\_proto);  
 if (\*sock < 0){  
 res = -1;  
 perror("Cant create socket\n");  
 }  
  
 //Bind the socket  
 if(bind(\*sock, (struct sockaddr \*)&server, sizeof(server)) < 0)  
 {  
 res = -1;  
 perror("Cant bind to specified port\n");  
 }  
  
 return res;  
}  
  
/\*\*  
 \* Simple utility function that reads a line from a file descriptor fd,   
 \* up to maxlen bytes -- ripped from Unix Network Programming, Stevens.  
 \*/  
int readline(int fd, char \*buf, int maxlen)  
{  
 int n, rc;  
 char c;  
  
 for (n = 1; n < maxlen; n++) {  
 rc = read(fd, &c, 1);  
  
 if (rc == 1) {  
 \*buf++ = c;  
 if (c == '\n')  
 break;  
   
 } else if (rc == 0) {  
 if (n == 1)  
 return 0; // EOF, no data read  
 else  
 break; // EOF, read some data  
 } else  
 return -1; // error  
 }  
  
 \*buf = '\0'; // null-terminate  
 return n;  
}  
  
  
int main(int argc, char \*\*argv)  
{  
 struct sockaddr\_in client; //client data structure  
 int bytes\_received = 0;  
 int n,result, len;  
 int clientfd;  
 int num;  
 pid\_t cpid;  
 int data\_len;  
 char data[MAX\_DATA];  
 char parent\_line[MAX\_DATA];  
 char child\_line[MAX\_DATA];  
 char cpid\_s[32];  
 char err\_msg[32] = "";  
 int fd;  
 struct UDP\_thread u;//Create an instance to hold data for thread  
  
 //Check for the arguments passed to program  
 if(argc != 3)  
 {  
 printf("Usage: ./server <port>\n");  
 exit(0);  
 }  
  
 u.len = sizeof(struct sockaddr\_in);  
  
 //Create the threads for the argument passed in comman line  
 int num\_of\_threads = atoi(argv[2]);  
  
 result = passivesock(argv[1], "udp", &u.socket\_new);  
  
 if (result == -1)  
 exit(-1);  
  
 printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("Concurrent UDP File Server Program: listening on socket %s..\n", argv[1]);  
 printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("Threads created based on the command line argument: \n\n");   
 while (1)  
 {   
  
 socklen\_t client\_t=sizeof(struct sockaddr\_in);  
 memset(&client, 0, sizeof(client));  
 memset(parent\_line, 0, MAX\_DATA);  
 memset(child\_line,0, MAX\_DATA);  
  
 //parent: wait for incoming client datagram  
 bytes\_received = recvfrom(u.socket\_new, u.buffer, MAX\_DATA, 0, (struct sockaddr \*)&u.from\_client, &u.len);  
 if(0 > bytes\_received)  
 {  
 perror("recvfrom error");  
 close(u.socket\_new);  
 exit(-1);  
 }  
 //Create thread ID for the number of threads specified (num\_of\_threads)  
 pthread\_t thread\_id[num\_of\_threads];  
 int return\_c = 0;  
 //Create the threads to handle the function handle\_request  
 for (int i = 0 ; i < num\_of\_threads; i++)  
 {  
  
 return\_c = pthread\_create(&thread\_id[i],NULL,handle\_req,&u);  
 if (return\_c != 0){  
 sleep(0.05);  
 perror("Error creating thread\n");  
 exit(-1);  
   
 }  
 else  
 {  
 // we are in thread  
 printf("Thread %d created for the client \n", i+1);  
 }  
 }  
 //Wait for all the threads to complete  
 for (int i = 0 ; i < num\_of\_threads; i++)  
 {  
  
 pthread\_join(thread\_id[i],NULL);  
 }  
 }  
  
   
  
 //close the server socket  
 close(u.socket\_new);  
 printf("Main process exiting\n");  
}

**Output**

ubuntu@ubuntu:~/Documents/UDP\_prethread\_client\_server$ ./server4 10000 5  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Concurrent UDP File Server Program: listening on socket 10000..  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Threads created based on the command line argument:   
  
Thread 1 created for the client   
Thread 2 created for the client   
Thread 3 created for the client   
Thread 4 created for the client   
Thread 5 created for the client   
File request from client: one\_k\_file.txt   
File name matched: one\_k\_file.txt  
File transfer done.Thread Exiting..

**3. Test Clients:**

**3.1 Iterative TCP File server - Test client**

**Readme.txt**

1. To Open the editor:   
gedit tcp\_client.c  
  
2. To Compile to program  
make all  
  
3. To Execute the program  
To get data from the 1 KB file: ./tcp\_client 127.0.0.1 test1 10023  
To get data from the 2 KB file: ./tcp\_client 127.0.0.1 test2 10023

**Makefile**

.PHONY: server client

all: client

client:

gcc tcp\_client.c -o tcp\_client

clean:

rm client

**Source files**

/\*  
TCP\_Client program by group-02  
Make file, code, output are provided  
Reference: Internetworking with TCP/IP Vol. 3, Client-Server programming and applications, Comer and Stevens, Linux/POSIX Sockets version, ISBN: 0-13-032071-4, 2001.  
\*/  
#include <errno.h>  
#include <netinet/in.h>  
#include <time.h>  
#include <unistd.h>  
#include <stdarg.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <arpa/inet.h>  
#include <netdb.h>  
#include <string.h>  
#include <fcntl.h>  
  
#define \_\_USE\_BSD 1  
#define UNIXEPOCH 2208988800   
  
#ifndef INADDR\_NONE  
#define INADDR\_NONE 0xffffffff  
#endif /\* INADDR\_NONE \*/  
typedef unsigned long u\_long;  
typedef unsigned short u\_short;  
extern int errno;  
  
#define LINELEN 128  
  
//int passiveTCP(const char \*service,int qlen);  
int errexit(const char \*format, ...);  
//int passivesock(const char \*service, const char \*transport,int qlen);  
  
int connectTCP(const char \*host, const char \*service);  
  
int connectsock(const char \*host, const char \*service,const char \*transport);  
  
//int TCPdaytimed(int fd);  
//#define QLEN 5  
/\*------------------------------------------------------------------------  
\* main - TCP client  
\*------------------------------------------------------------------------  
\*/  
int main(int argc, char \*argv[])  
{  
struct sockaddr\_in fsin;  
char \*service = "echo";  
char buffer[2048];  
int msock,ssock;  
unsigned int alen;  
char \*host1;  
char \*file\_n;  
int port\_n;  
//printf("Before switch");  
char recvv[10000];  
int count=1;  
  
ssize\_t s\_file\_n,recv\_data;  
int new\_file;  
  
switch (argc)  
{  
case 1:  
host1 = "localhost";  
break;  
case 4:  
host1 = argv[1];  
file\_n=argv[2];  
port\_n=atoi(argv[3]);  
break;  
  
default:  
fprintf(stderr, "usage: TCP [host [port]]\n");  
exit(1);  
}  
  
//printf("After Switch");  
msock = connectTCP(host1,service);  
  
  
send(msock,file\_n,2048,0);  
printf("%s file is requested from the server \n",file\_n);  
  
while ((recv\_data=recv(msock,recvv,10000,0))>0) {  
  
if(count==1)  
{  
new\_file=open(file\_n,O\_WRONLY|O\_CREAT,0644);  
printf("File is created\n");  
count++;  
}  
  
if(count!=1)  
{  
printf("%s \n is the data received from the server\n\n Written into the file\n",recvv);  
write(new\_file,recvv,recv\_data);  
}  
  
printf("connection closed\n");  
(void) close(msock);  
}  
  
}  
  
  
  
/\*------------------------------------------------------------------------  
\* connectTCP - connect to a specified TCP service on a specified host  
\*------------------------------------------------------------------------  
\*/  
int connectTCP(const char \*host, const char \*service )  
/\*  
\* Arguments:  
\* host - name of host to which connection is desired  
\* service - service associated with the desired port  
\*/  
{  
return connectsock( host, service, "tcp");  
}  
  
  
  
  
/\*------------------------------------------------------------------------  
\* connectsock - allocate & connect a socket using TCP or UDP  
\*------------------------------------------------------------------------  
\*/  
int connectsock(const char \*host, const char \*service, const char \*transport )  
/\*  
\* Arguments:  
\* host - name of host to which connection is desired  
\* service - service associated with the desired port  
\* transport - name of transport protocol to use ("tcp" or "udp")  
\*/  
{  
struct hostent \*phe; /\* pointer to host information entry \*/  
struct servent \*pse; /\* pointer to service information entry \*/  
struct protoent \*ppe; /\* pointer to protocol information entry\*/  
struct sockaddr\_in sin; /\* an Internet endpoint address \*/  
int s, type; /\* socket descriptor and socket type \*/  
memset(&sin, 0, sizeof(sin));  
sin.sin\_family = AF\_INET;  
/\* Map service name to port number \*/  
if ( pse = getservbyname(service, transport) )  
sin.sin\_port = htons(10023);  
else if ( (sin.sin\_port = htons((u\_short)atoi(service))) == 0 )  
errexit("can't get \"%s\" service entry\n", service);  
/\* Map host name to IP address, allowing for dotted decimal \*/  
if ( phe = gethostbyname(host) )  
memcpy(&sin.sin\_addr, phe->h\_addr, phe->h\_length);  
else if ( (sin.sin\_addr.s\_addr = inet\_addr(host)) == INADDR\_NONE )  
errexit("can't get \"%s\" host entry\n", host);  
/\* Map transport protocol name to protocol number \*/  
if ( (ppe = getprotobyname(transport)) == 0)  
errexit("can't get \"%s\" protocol entry\n", transport);  
/\* Use protocol to choose a socket type \*/  
if (strcmp(transport, "udp") == 0)  
type = SOCK\_DGRAM;  
else  
type = SOCK\_STREAM;  
/\* Allocate a socket \*/  
s = socket(PF\_INET, type, ppe->p\_proto);  
if (s < 0)  
errexit("can't create socket: %s\n", strerror(errno));  
/\* Connect the socket \*/  
if (connect(s, (struct sockaddr \*)&sin, sizeof(sin)) < 0)  
errexit("can't connect to %s.%s: %s\n", host, service,strerror(errno));  
return s;  
  
}  
  
  
  
/\*------------------------------------------------------------------------  
\* errexit - print an error message and exit  
\*------------------------------------------------------------------------  
\*/  
/\*VARARGS1\*/  
int errexit(const char \*format, ...)  
{  
va\_list args;  
va\_start(args, format);  
vfprintf(stderr, format, args);  
va\_end(args);  
exit(1);  
}

**Output**

ubuntu@ubuntu:~/tcp/TCP\_client$ gcc tcp\_client.c -o tcp\_client

ubuntu@ubuntu:~/tcp/TCP\_client$ ./tcp\_client 127.0.0.1 test1 10023

test1 file is requested from the server

File is created

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

is the data received from the server

Written into the file

connection closed

ubuntu@ubuntu:~/tcp/TCP\_client$ ./tcp\_client 127.0.0.1 test2 10023

test2 file is requested from the server

File is created

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

is the data received from the server

Written into the file

connection closed

**3.2 Concurrent multiprocessing TCP File server with one process per request - Test client**

**Readme.txt**

Steps to execute the program:

1. Open the terminal

2. Navigate to the program directory

3. Build server & client:

make all

Build server alone:

make server

Build client alone:

make client

5. Run server and client on two different terminals:

./server <port>

ex: ./server 10000

./client <server\_IP> <port> <file\_name>

ex: ./client 127.0.0.1 10000 one\_k\_file.txt

ex: ./client 127.0.0.1 10000 two\_k\_file.txt

6. Enter ^C to quit running the server.

TCP File Server: Server program creates child processes to handle the incoming client

request on demand. Child exits once client is served. server sends specified file to client.

Server can transfer 1. one\_k\_file.txt (1K size)

2. two\_k\_file.txt (2K size)

TCP test client: Client takes a file name input from the user and sends it to the TCP server.

Server transfer the specifies file back to the client. Client saves it on local disk with name:

test\_file.txt

If specified file does not exist on the server then server sends back an error message which

is saved into file "test\_file.txt". However file transfer is printed "successful" on the

client display. User must verify the copied file on the disk.

**Makefile**

# Make file targets to build TCP client & TCP server (Assignment 1, problem #3)

# Format

# target: dependencies

# action

.PHONY: client

all: client

client:

gcc test\_client.c -o client

clean:

rm client

**Source files**

**TCP Client:**

// CMPE\_297 Assignment\_1 Program #3

// TCP Client

// Author - Team #2

// This is a TCP test client program. Client connects

// the server. Requests a specific file (1K or 2K file)

// from the server. Save the file on the local disk.

// To compile: gcc tcp\_client\_3.c -o client

// To run: ./client server\_address server\_port

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/socket.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<netinet/in.h>

#include<string.h>

#include<ctype.h>

#include<arpa/inet.h>

#include<netdb.h>

#include<sys/wait.h>

#define ERROR -1

#define BUFFER 1024

int connectsock(const char \*host, const char \*service, const char \*transport)

{

struct hostent \*phe;

struct servent \*pse;

struct protoent \*ppe;

struct sockaddr\_in remote\_server;

int sock, type;

//reset socket address structure.

memset(&remote\_server,0,sizeof(remote\_server));

remote\_server.sin\_family = AF\_INET;

//map service name to port number

if(pse = getservbyname(service, transport))

remote\_server.sin\_port = pse->s\_port;

else if((remote\_server.sin\_port = htons((u\_short)atoi(service))) == 0)

//perror("Cant get \"%s\" service entry\n", service);

perror("Cant get service entry\n");

//map host name to IP address allowing dotted decimal

if(phe = gethostbyname(host))

memcpy(&remote\_server.sin\_addr, phe->h\_addr, phe->h\_length);

else if ((remote\_server.sin\_addr.s\_addr = inet\_addr(host) == INADDR\_NONE))

//perror("Cant get \"%s\" host entry\n", host);

perror("Cant get host entry\n");

bzero(&remote\_server.sin\_zero, 8);

//map transport protocol name to protocol number

ppe = getprotobyname(transport);

if((ppe == NULL))

//perror("cant get \"%s\" protocol entry\n", transport);

perror("Cant get protocol entry\n");

//use protocol to chose a socket type

if(strcmp(transport, "udp") == 0)

type = SOCK\_DGRAM;

else

type = SOCK\_STREAM;

//allocate a socket

sock = socket(PF\_INET, type, ppe->p\_proto);

if(sock < 0)

//perror("Cant create socket: %s\n", strerror(errno));

perror("Cant create socket\n");

//connect the socket

if(connect(sock, (struct sockaddr \*)&remote\_server, sizeof(remote\_server)) < 0)

//perror("Cant connect to %s %s: %s\n", host, service, sererror(errno));

perror("Cant connect to the remote host\n");

return sock;

}

int main(int argc, char \*\*argv)

{

int sock;

int c, fd;

char input[BUFFER];

char output[BUFFER];

int len;

if(argc < 4)

{

printf("Usage: ./client <server IP> <server port> <file\_name> \n");

printf("Server hosts \"one\_k\_file.txt\" & \"two\_k\_file.txt\". \n");

printf("User must request one of these two files for <file\_name> argument\n");

exit(0);

} else

{

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("TCP test client program. Requesting %s file from server.\n",argv[3]);

printf("File will be stored in the client local current directory as \"test\_file.txt\"\n");

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

sock = connectsock(argv[1], argv[2], "tcp");

if (sock < 0){

printf("connection Failed.\n");

exit(-1);

}

// connection successfuly established

printf("Connection successful.Requesting file transfer..\n");

memset(input,0,BUFFER);

strcpy(input, argv[3]);

input[strlen(input)] = '\n';

// send file number

if (0 > send(sock, input, strlen(input), 0))

{

perror("send error");

close(sock);

exit(1);

}

// open file for writing

fd = open("test\_file.txt", O\_CREAT|O\_TRUNC|O\_WRONLY, S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);

if(fd < 0){

printf("Failed to create file\n");

exit(1);

}

//receive file and write it to the file

while(1)

{

memset(output, 0, BUFFER);

len = recv(sock, output, BUFFER, 0);

if (len < 0)

{

perror("recv error");

close(fd);

close(sock);

exit(1);

}

if (len == 0)

break;

//write to file

len = write(fd, output, strlen(output));

if(len < 0) {

printf("File write error..exiting..\n");

close(fd);

close(sock);

exit(1);

}

}

close(fd);

//close client socket

printf("File transfer successful. Closing the socket.Exiting the child.\n");

close(sock);

}

}

**Output**

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/TCP\_fork\_server\_client$ ./client 127.0.0.1 10000 one\_k\_file.txt

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TCP test client program. Requesting one\_k\_file.txt file from server.

File will be stored in the client local current directory as "test\_file.txt"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Connection successful.Requesting file transfer..

File transfer successful. Closing the socket.Exiting the child.

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/TCP\_fork\_server\_client$ cat test\_file.txt

start test\_file\_1

Given a pathname for a file, open() returns a file descriptor, a small, non-negative integer for use in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.). The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

The new file descriptor is set to remain open across an execve(2) (i.e., the FD\_CLOEXEC file descriptor flag described in fcntl(2) is initially disabled). The file offset is set to the beginning of the file (see lseek(2)).

A call to open() creates a new open file description, an entry in the system-wide table of open files. This entry records the file offset and the file status flags (modifiable via the fcntl() F\_SETFL operation). A file descriptor is a reference to one of these entries; this reference is unaffected if pathname is subsequently removed or modified to refer to a different file. The new open file description is initially not shared with any other process, but sharing may arise via fork(2).

end test\_file\_1

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/TCP\_fork\_server\_client$ ./client 127.0.0.1 10000 one\_k\_frle.txt

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TCP test client program. Requesting one\_k\_frle.txt file from server.

File will be stored in the client local current directory as "test\_file.txt"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Connection successful.Requesting file transfer..

File transfer successful. Closing the socket.Exiting the child.

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/TCP\_fork\_server\_client$ cat test\_file.txt

Error: Requested File does not exist on the server.

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/TCP\_fork\_server\_client$

**3.3 Concurrent multithreading TCP File server with one thread per request - Test client**

**Readme.txt**

Steps to execute the program:  
  
1. Open the terminal  
2. Navigate to the program directory  
3. Build server & client:  
 make MakeFile.txt - for server  
 make MakeFileClient.txt - for client  
4. Run server and client on two different terminals:  
 ./server <port>  
 ex: ./server 10000   
  
 ./client <server\_IP> <port>  
 ex: ./client 127.0.0.1 10000  
  
5. Enter ^C to quit running the server.

**Makefile**

all:  
 gcc -o client1 multithreaded\_test\_client\_new.c  
clean:  
 rm -f client1

**Source files**

TCP client program

/\* Client - Multithreaded server  
Assignment-02 Group-02  
Step 1: Creating a socket  
Step 2: Get portnumber and IP address  
step 3: Connect with the socket  
step 4: Read the values  
\*/  
  
  
#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<sys/socket.h>  
#include<sys/types.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
#include<netinet/in.h>  
#include<string.h>  
#include<ctype.h>  
#include<arpa/inet.h>  
#include<netdb.h>  
#include<sys/wait.h>  
  
#define ERROR -1  
#define BUFFER 1024  
  
int connectsock(const char \*host, const char \*service, const char \*transport)  
{  
 struct hostent \*phe;  
 struct servent \*pse;  
 struct protoent \*ppe;  
 struct sockaddr\_in remote\_server;  
 int sock, type;  
  
 //reset socket address structure.  
 memset(&remote\_server,0,sizeof(remote\_server));  
  
 remote\_server.sin\_family = AF\_INET;  
  
 //map service name to port number  
 if(pse = getservbyname(service, transport))  
 remote\_server.sin\_port = pse->s\_port;  
 else if((remote\_server.sin\_port = htons((u\_short)atoi(service))) == 0)  
 //perror("Cant get \"%s\" service entry\n", service);  
 perror("Cant get service entry\n");  
  
 //map host name to IP address allowing dotted decimal  
 if(phe = gethostbyname(host))  
 memcpy(&remote\_server.sin\_addr, phe->h\_addr, phe->h\_length);  
 else if ((remote\_server.sin\_addr.s\_addr = inet\_addr(host) == INADDR\_NONE))  
 //perror("Cant get \"%s\" host entry\n", host);  
 perror("Cant get host entry\n");  
  
 bzero(&remote\_server.sin\_zero, 8);  
  
 //map transport protocol name to protocol number  
 ppe = getprotobyname(transport);  
 if((ppe == NULL))  
 //perror("cant get \"%s\" protocol entry\n", transport);  
 perror("Cant get protocol entry\n");  
  
 //use protocol to chose a socket type  
 if(strcmp(transport, "udp") == 0)  
 type = SOCK\_DGRAM;  
 else  
 type = SOCK\_STREAM;  
  
 //allocate a socket  
 sock = socket(PF\_INET, type, ppe->p\_proto);  
 if(sock < 0)  
 //perror("Cant create socket: %s\n", strerror(errno));  
 perror("Cant create socket\n");  
  
 //connect the socket  
 if(connect(sock, (struct sockaddr \*)&remote\_server, sizeof(remote\_server)) < 0)  
 //perror("Cant connect to %s %s: %s\n", host, service, sererror(errno));  
 perror("Cant connect to the remote host\n");  
  
 return sock;  
}  
  
  
int main(int argc, char \*\*argv)  
{  
 int sock;  
 int c, fd;  
 char input[BUFFER];  
 char output[BUFFER];  
 int len;   
  
 if(argc < 4)  
 {  
 printf("Usage: ./client <server IP> <server port> <file\_name> \n");  
 printf("Server hosts \"one\_k\_file.txt\" & \"two\_k\_file.txt\". \n");  
 printf("User must request one of these two files for <file\_name> argument\n");  
 exit(0);  
 } else  
 {  
 printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("TCP test client program. Requesting %s file from server.\n",argv[3]);  
 printf("File will be stored in the client local current directory as \"test\_file.txt\"\n");  
 printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
   
 sock = connectsock(argv[1], argv[2], "tcp");  
 if (sock < 0){  
 printf("connection Failed.\n");  
 exit(-1);  
 }  
  
 // connection successfuly established  
 printf("Connection successful.Requesting file transfer..\n");  
  
 memset(input,0,BUFFER);  
 strcpy(input, argv[3]);  
 input[strlen(input)] = '\n';  
  
 // send file number  
 if (0 > send(sock, input, strlen(input), 0))  
 {  
 perror("send error");  
 close(sock);  
 exit(1);  
 }  
  
 // open file for writing  
 fd = open("test\_file.txt", O\_CREAT|O\_TRUNC|O\_WRONLY, S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);  
 if(fd < 0){  
 printf("Failed to create file\n");  
 exit(1);  
 }  
  
 //receive file and write it to the file   
 while(1)  
 {  
 memset(output, 0, BUFFER);  
 len = recv(sock, output, BUFFER, 0);  
 if (len < 0)  
 {  
 perror("recv error");  
 close(fd);  
 close(sock);  
 exit(1);  
 }  
 if (len == 0)  
 break;  
  
 //write to file  
 len = write(fd, output, strlen(output));  
 if(len < 0) {  
 printf("File write error..exiting..\n");  
 close(fd);  
 close(sock);  
 exit(1);  
 }  
 }  
  
 close(fd);  
 //close client socket  
 printf("File transfer successful. Closing the socket.\n");  
 close(sock);  
 }  
}

**Output**

ubuntu@ubuntu:~/Documents/Multithreaded-TCP$ ./client1 127.0.0.1 10000 one\_k\_file.txt  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
TCP test client program. Requesting one\_k\_file.txt file from server.  
File will be stored in the client local current directory as "test\_file.txt"  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Connection successful.Requesting file transfer..  
File transfer successful. Closing the socket.

**3.4 Concurrent pre-forked TCP multiprocessing server - Test client**

**Readme.txt**

Steps to execute the program:

1. Open the terminal

2. Navigate to the program directory

3. Build server & client:

make all

Build server alone:

make server

Build client alone:

make client

Clean all targets:

make clean

5. Run server and client on two different terminals:

./server <port>

ex: ./server 10000

./client <server\_IP> <port>

ex: ./client 127.0.0.1 10000

6. Enter ^C to quit running the server.

TCP Server: Server program creates 5 child processes in advance (process pool)to handle the incoming client request. Each child waits for the incoming connection by calling accept() system call. One of the children accepts the connection when a new connection is requested from the client and serves the client. child process returns back to the pool once client connection is served and disconnected.

TCP test client: client connects the server and receives number of preforked children at server side and prints it to the screen.

**Makefile**

# Make file targets to build TCP client & TCP server (Assignment 1, problem #3)

# Format

# target: dependencies

# action

.PHONY: client

all: client

client:

gcc test\_client.c -o client

clean:

rm client

**Source files**

// CMPE\_297 Assignment\_2

// TCP Client

// Author - Team #2

// This is a TCP test client program. It connects to the server

// and receives total number of processes preforked at the

// server side. It then prints the number on the output screen.

// To compile: gcc test\_client.c -o client

// To run: ./client server\_address server\_port

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/socket.h>

#include<sys/types.h>

#include<netinet/in.h>

#include<string.h>

#include<ctype.h>

#include<arpa/inet.h>

#include<netdb.h>

#include<sys/wait.h>

#define ERROR -1

#define BUFFER 1024

int connectsock(const char \*host, const char \*service, const char \*transport)

{

struct hostent \*phe;

struct servent \*pse;

struct protoent \*ppe;

struct sockaddr\_in remote\_server;

int sock, type;

//reset socket address structure.

memset(&remote\_server,0,sizeof(remote\_server));

remote\_server.sin\_family = AF\_INET;

//map service name to port number

if(pse = getservbyname(service, transport))

remote\_server.sin\_port = pse->s\_port;

else if((remote\_server.sin\_port = htons((u\_short)atoi(service))) == 0)

//perror("Cant get \"%s\" service entry\n", service);

perror("Cant get service entry\n");

//map host name to IP address allowing dotted decimal

if(phe = gethostbyname(host))

memcpy(&remote\_server.sin\_addr, phe->h\_addr, phe->h\_length);

else if ((remote\_server.sin\_addr.s\_addr = inet\_addr(host) == INADDR\_NONE))

//perror("Cant get \"%s\" host entry\n", host);

perror("Cant get host entry\n");

bzero(&remote\_server.sin\_zero, 8);

//map transport protocol name to protocol number

ppe = getprotobyname(transport);

if((ppe == NULL))

//perror("cant get \"%s\" protocol entry\n", transport);

perror("Cant get protocol entry\n");

//use protocol to chose a socket type

if(strcmp(transport, "udp") == 0)

type = SOCK\_DGRAM;

else

type = SOCK\_STREAM;

//allocate a socket

sock = socket(PF\_INET, type, ppe->p\_proto);

if(sock < 0)

//perror("Cant create socket: %s\n", strerror(errno));

perror("Cant create socket\n");

//connect the socket

if(connect(sock, (struct sockaddr \*)&remote\_server, sizeof(remote\_server)) < 0)

//perror("Cant connect to %s %s: %s\n", host, service, sererror(errno));

perror("Cant connect to the remote host\n");

return sock;

}

int main(int argc, char \*\*argv)

{

int sock;

int len;

char output[BUFFER];

if(argc < 3)

{

printf("Usage: ./client <server IP> <server port> \n");

exit(0);

}

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("TCP test client: This client connects to the concurrent preforked TCP server\n");

printf("receives number of preforked child processes at the server and prints it.\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

printf("Initiating a connection to TCP server ===> ");

sock = connectsock(argv[1], argv[2], "tcp");

if (sock < 0){

printf("Connection Failed\n");

exit(-1);

}

// connection successfuly established

printf("Connection established.");

memset(output, 0, BUFFER);

// receive server's response and print.

len = recv(sock, output, BUFFER, 0);

if (len < 0)

{

perror("recv error..");

exit(-1);

}

output[len] = '\0';

printf("%s", output);

close(sock);

}

**Output**

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/TCP\_prefork\_server\_client$ ./client 127.0.0.1 10000

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TCP test client: This client connects to the concurrent preforked TCP server

receives number of preforked child processes at the server and prints it.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Initiating a connection to TCP server ===> Connection established.

Number of preforked processes at is server is 5

kanti@kanti-linux:

**3.5 Concurrent Prethreaded TCP multithreading server - Test client**

**Readme.txt**

Steps to execute the program:  
  
1. Open the terminal  
2. Navigate to the program directory  
3. Build server & client:  
 make MakeFile1.txt - for server  
 make MakeFile1Client.txt - for client  
4. Run server and client on two different terminals:  
 ./server <port> <Number of threads>  
 ex: ./server 10000 4  
  
 ./client <server\_IP> <port>  
 ex: ./client 127.0.0.1 10000  
  
5. Enter ^C to quit running the server.

**Makefile**

all:  
 gcc -o client2 prethreaded\_test\_client\_new.c  
clean:  
 rm -f client2

**Source files**

/\* Client - Multithreaded server  
Assignment-02 Group-02  
Step 1: Creating a socket  
Step 2: Get portnumber and IP address  
step 3: Connect with the socket  
step 4: Read the values  
\*/  
  
  
#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<sys/socket.h>  
#include<sys/types.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
#include<netinet/in.h>  
#include<string.h>  
#include<ctype.h>  
#include<arpa/inet.h>  
#include<netdb.h>  
#include<sys/wait.h>  
  
#define ERROR -1  
#define BUFFER 1024  
  
int connectsock(const char \*host, const char \*service, const char \*transport)  
{  
 struct hostent \*phe;  
 struct servent \*pse;  
 struct protoent \*ppe;  
 struct sockaddr\_in remote\_server;  
 int sock, type;  
  
 //reset socket address structure.  
 memset(&remote\_server,0,sizeof(remote\_server));  
  
 remote\_server.sin\_family = AF\_INET;  
  
 //map service name to port number  
 if(pse = getservbyname(service, transport))  
 remote\_server.sin\_port = pse->s\_port;  
 else if((remote\_server.sin\_port = htons((u\_short)atoi(service))) == 0)  
 //perror("Cant get \"%s\" service entry\n", service);  
 perror("Cant get service entry\n");  
  
 //map host name to IP address allowing dotted decimal  
 if(phe = gethostbyname(host))  
 memcpy(&remote\_server.sin\_addr, phe->h\_addr, phe->h\_length);  
 else if ((remote\_server.sin\_addr.s\_addr = inet\_addr(host) == INADDR\_NONE))  
 //perror("Cant get \"%s\" host entry\n", host);  
 perror("Cant get host entry\n");  
  
 bzero(&remote\_server.sin\_zero, 8);  
  
 //map transport protocol name to protocol number  
 ppe = getprotobyname(transport);  
 if((ppe == NULL))  
 //perror("cant get \"%s\" protocol entry\n", transport);  
 perror("Cant get protocol entry\n");  
  
 //use protocol to chose a socket type  
 if(strcmp(transport, "udp") == 0)  
 type = SOCK\_DGRAM;  
 else  
 type = SOCK\_STREAM;  
  
 //allocate a socket  
 sock = socket(PF\_INET, type, ppe->p\_proto);  
 if(sock < 0)  
 //perror("Cant create socket: %s\n", strerror(errno));  
 perror("Cant create socket\n");  
  
 //connect the socket  
 if(connect(sock, (struct sockaddr \*)&remote\_server, sizeof(remote\_server)) < 0)  
 //perror("Cant connect to %s %s: %s\n", host, service, sererror(errno));  
 perror("Cant connect to the remote host\n");  
  
 return sock;  
}  
  
  
int main(int argc, char \*\*argv)  
{  
 int sock;  
 int c, fd;  
 char input[BUFFER];  
 char output[BUFFER];  
 int len;   
  
 if(argc < 4)  
 {  
 printf("Usage: ./client <server IP> <server port> <file\_name> \n");  
 printf("Server hosts \"one\_k\_file.txt\" & \"two\_k\_file.txt\". \n");  
 printf("User must request one of these two files for <file\_name> argument\n");  
 exit(0);  
 } else  
 {  
 printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("TCP test client program. Requesting %s file from server.\n",argv[3]);  
 printf("File will be stored in the client local current directory as \"test\_file.txt\"\n");  
 printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
   
 sock = connectsock(argv[1], argv[2], "tcp");  
 if (sock < 0){  
 printf("connection Failed.\n");  
 exit(-1);  
 }  
  
 // connection successfuly established  
 printf("Connection successful.Requesting file transfer..\n");  
  
 memset(input,0,BUFFER);  
 strcpy(input, argv[3]);  
 input[strlen(input)] = '\n';  
  
 // send file number  
 if (0 > send(sock, input, strlen(input), 0))  
 {  
 perror("send error");  
 close(sock);  
 exit(1);  
 }  
  
 // open file for writing  
 fd = open("test\_file.txt", O\_CREAT|O\_TRUNC|O\_WRONLY, S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);  
 if(fd < 0){  
 printf("Failed to create file\n");  
 exit(1);  
 }  
  
 //receive file and write it to the file   
 while(1)  
 {  
 memset(output, 0, BUFFER);  
 len = recv(sock, output, BUFFER, 0);  
 if (len < 0)  
 {  
 perror("recv error");  
 close(fd);  
 close(sock);  
 exit(1);  
 }  
 if (len == 0)  
 break;  
  
 //write to file  
 len = write(fd, output, strlen(output));  
 if(len < 0) {  
 printf("File write error..exiting..\n");  
 close(fd);  
 close(sock);  
 exit(1);  
 }  
 }  
  
 close(fd);  
 //close client socket  
 printf("File transfer successful. Closing the socket.\n");  
 close(sock);  
 }  
}

**Output**

ubuntu@ubuntu:~/Documents/Prethreaded-TCP$ ./client2 127.0.0.1 10000 two\_k\_file.txt  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
TCP test client program. Requesting two\_k\_file.txt file from server.  
File will be stored in the client local current directory as "test\_file.txt"  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Connection successful.Requesting file transfer..  
File transfer successful. Closing the socket.

**3.6 Iterative UDP File server - Test client**

**Readme.txt**

1. To Open the editor:   
gedit udp\_client.c  
  
2. To Compile to program  
make all  
  
3. To Execute the program  
To get data from the 1 KB file: ./udp\_client 127.0.0.1 test1 10026  
To get data from the 2 KB file: ./udp\_client 127.0.0.1 test2 10026

**Makefile**

.PHONY: server client

all: client

client:

gcc udp\_client.c -o udp\_client

clean:

rm client

**Source files**

/\*  
UDP Client program by Group-02  
Make file, code, output are provided  
Reference: Internetworking with TCP/IP Vol. 3, Client-Server programming and applications, Comer and Stevens, Linux/POSIX Sockets version, ISBN: 0-13-032071-4, 2001.  
\*/  
#include <errno.h>  
#include <netinet/in.h>  
#include <time.h>  
#include <unistd.h>  
#include <stdarg.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <arpa/inet.h>  
#include <netdb.h>  
#include <string.h>  
#include <fcntl.h>  
  
#define \_\_USE\_BSD 1  
#define UNIXEPOCH 2208988800   
  
#ifndef INADDR\_NONE  
#define INADDR\_NONE 0xffffffff  
#endif /\* INADDR\_NONE \*/  
typedef unsigned long u\_long;  
typedef unsigned short u\_short;  
extern int errno;  
  
#define LINELEN 128  
  
//int passiveTCP(const char \*service,int qlen);  
int errexit(const char \*format, ...);  
//int passivesock(const char \*service, const char \*transport,int qlen);  
  
int connectUDP(const char \*host, const char \*service);  
  
int connectsock(const char \*host, const char \*service,const char \*transport);  
  
  
#define QLEN 5  
/\*------------------------------------------------------------------------  
\* main  
\*------------------------------------------------------------------------  
\*/  
int main(int argc, char \*argv[])  
{  
int n=0,count=1;  
struct sockaddr\_in fsin;  
char \*service = "echo";  
char buffer[10000];  
int msock,ssock;  
unsigned int alen;  
char \*host1;  
char \*file\_n;  
int port\_n;  
ssize\_t s\_file\_n,recv\_data;  
//printf("Before switch");  
int new\_file;  
  
switch (argc)  
{  
case 1:  
host1 = "localhost";  
break;  
case 4:  
host1 = argv[1];  
file\_n=argv[2];  
port\_n=atoi(argv[3]);  
break;  
  
default:  
fprintf(stderr, "usage: UDP [host [port]]\n");  
exit(1);  
}  
  
//printf("After Switch");  
msock = connectUDP(host1,service);  
  
send(msock,file\_n,2048,0);  
  
printf("%s is the file requested from the server\n",file\_n);  
if((recv\_data=recv(msock,buffer,10000,0))>0)  
{  
  
if(1)  
{  
printf("File is created\n");  
new\_file=open(file\_n,O\_WRONLY|O\_CREAT,0644);  
count++;  
}  
  
if(1)  
{  
  
write(new\_file,buffer,recv\_data);  
  
printf("%s \nis the data received from the server\n",buffer);  
}  
close(msock);  
}  
printf("Done\n");  
  
(void) close(msock);  
  
}  
  
  
  
/\*------------------------------------------------------------------------  
\* connectUDP - connect to a specified UDP service on a specified host  
\*------------------------------------------------------------------------  
\*/  
int connectUDP(const char \*host, const char \*service )  
/\*  
\* Arguments:  
\* host - name of host to which connection is desired  
\* service - service associated with the desired port  
\*/  
{  
return connectsock(host,service,"udp");  
}  
  
  
  
/\*------------------------------------------------------------------------  
\* connectsock - allocate & connect a socket using TCP or UDP  
\*------------------------------------------------------------------------  
\*/  
int connectsock(const char \*host, const char \*service, const char \*transport )  
/\*  
\* Arguments:  
\* host - name of host to which connection is desired  
\* service - service associated with the desired port  
\* transport - name of transport protocol to use ("tcp" or "udp")  
\*/  
{  
struct hostent \*phe; /\* pointer to host information entry \*/  
struct servent \*pse; /\* pointer to service information entry \*/  
struct protoent \*ppe; /\* pointer to protocol information entry\*/  
struct sockaddr\_in sin; /\* an Internet endpoint address \*/  
int s, type; /\* socket descriptor and socket type \*/  
memset(&sin, 0, sizeof(sin));  
sin.sin\_family = AF\_INET;  
/\* Map service name to port number \*/  
if ( pse = getservbyname(service, transport) )  
sin.sin\_port = htons(10026);  
else if ( (sin.sin\_port = htons((u\_short)atoi(service))) == 0 )  
errexit("can't get \"%s\" service entry\n", service);  
/\* Map host name to IP address, allowing for dotted decimal \*/  
if ( phe = gethostbyname(host) )  
memcpy(&sin.sin\_addr, phe->h\_addr, phe->h\_length);  
else if ( (sin.sin\_addr.s\_addr = inet\_addr(host)) == INADDR\_NONE )  
errexit("can't get \"%s\" host entry\n", host);  
/\* Map transport protocol name to protocol number \*/  
if ( (ppe = getprotobyname(transport)) == 0)  
errexit("can't get \"%s\" protocol entry\n", transport);  
/\* Use protocol to choose a socket type \*/  
if (strcmp(transport, "udp") == 0)  
type = SOCK\_DGRAM;  
else  
type = SOCK\_STREAM;  
/\* Allocate a socket \*/  
s = socket(PF\_INET, type, ppe->p\_proto);  
if (s < 0)  
errexit("can't create socket: %s\n", strerror(errno));  
/\* Connect the socket \*/  
if (connect(s, (struct sockaddr \*)&sin, sizeof(sin)) < 0)  
errexit("can't connect to %s.%s: %s\n", host, service,strerror(errno));  
return s;  
}  
  
  
  
/\*------------------------------------------------------------------------  
\* errexit - print an error message and exit  
\*------------------------------------------------------------------------  
\*/  
/\*VARARGS1\*/  
int errexit(const char \*format, ...)  
{  
va\_list args;  
va\_start(args, format);  
vfprintf(stderr, format, args);  
va\_end(args);  
exit(1);  
}

**Output**

ubuntu@ubuntu:~/UDP\_client$ gcc udp\_client.c -o udp\_client

ubuntu@ubuntu:~/UDP\_client$ ./udp\_client 127.0.0.1 test1 10026

test1 is the file requested from the server

File is created

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

This is assignment 1. This is test 1 - 1 KB.

is the data received from the server

Done

ubuntu@ubuntu:~/UDP\_client$ ./udp\_client 127.0.0.1 test2 10026

test2 is the file requested from the server

File is created

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

This is assignment 1. This is test 2 - 2 KB.

is the data received from the server

Done

**3.7 Concurrent multiprocessing UDP File server with one process per request - Test client**

**Readme**

Steps to execute the program:

1. Open the terminal

2. Navigate to the program directory

3. Build server & client:

make all

Build server alone:

make server

Build client alone:

make client

Clean all targets:

make clean

5. Run server and client on two different terminals:

./server <port>

ex: ./server 10000

./client <server\_IP> <port>

ex: ./client 127.0.0.1 10000

6. Enter ^C to quit running the server.

UDP File Server:Server maintains 2 files "one\_k\_file.txt" and "two\_k\_file.txt". Server program creates child processes on demand to handle the incoming client request.When client sends a file transfer request, server validates the file name and transfer the file. If File name sent

by the client does not match with server files then an error message is sent back to the client.

UDP test client:A File name is passed to the client as a command line argument. Client sends a datagram to the server requesting this file

from the server. Client then waits for the file transfer. It opens "test\_file.txt" file and writes messages transfered by the server into

this file. Then closes the connection.

**Makefile**

# Make file targets to build server and clients

# Format

# target: dependencies

# action

.PHONY: client

all: client

client:

gcc udp\_test\_client.c -o client

clean:

rm client

**Source files**

**Udp\_test\_client.c**

// CMPE\_207 Assignment 2

// UDP Client

// Author - Team #2

// This is a UDP test client program. Client connects

// the server. Requests a specific file (1K or 2K file)

// from the server. Save the file on the local disk.

// To compile: gcc udp\_test\_client.c -o client

// To run: ./client <server\_address> <server\_port> <file\_name>

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/socket.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<netinet/in.h>

#include<string.h>

#include<ctype.h>

#include<arpa/inet.h>

#include<netdb.h>

#include<sys/wait.h>

#define ERROR -1

#define BUFFER 1024

int createsock(const char \*host, const char \*service, const char \*transport, struct sockaddr\_in \*remote\_server)

{

struct hostent \*phe;

struct servent \*pse;

struct protoent \*ppe;

//struct sockaddr\_in remote\_server;

int sock, type;

//reset socket address structure.

memset(remote\_server,0,sizeof(struct sockaddr\_in));

remote\_server->sin\_family = AF\_INET;

//map service name to port number

if(pse = getservbyname(service, transport))

remote\_server->sin\_port = pse->s\_port;

else if((remote\_server->sin\_port = htons((u\_short)atoi(service))) == 0)

//perror("Cant get \"%s\" service entry\n", service);

perror("Cant get service entry\n");

//map host name to IP address allowing dotted decimal

if(phe = gethostbyname(host))

memcpy(&((\*remote\_server).sin\_addr), phe->h\_addr, phe->h\_length);

else if ((remote\_server->sin\_addr.s\_addr = inet\_addr(host) == INADDR\_NONE))

//perror("Cant get \"%s\" host entry\n", host);

perror("Cant get host entry\n");

bzero(remote\_server->sin\_zero, 8);

//map transport protocol name to protocol number

ppe = getprotobyname(transport);

if((ppe == NULL))

//perror("cant get \"%s\" protocol entry\n", transport);

perror("Cant get protocol entry\n");

//use protocol to chose a socket type

if(strcmp(transport, "udp") == 0)

type = SOCK\_DGRAM;

else

type = SOCK\_STREAM;

//allocate a socket

sock = socket(PF\_INET, type, ppe->p\_proto);

if(sock < 0)

//perror("Cant create socket: %s\n", strerror(errno));

perror("Cant create socket\n");

return sock;

}

int main(int argc, char \*\*argv)

{

int sock;

int c, fd;

char input[BUFFER];

char output[BUFFER];

int len;

struct sockaddr\_in remote\_server;

socklen\_t addr\_len;

if(argc < 4)

{

printf("Usage: ./client <server IP> <server port> <file\_name> \n");

printf("Server hosts \"one\_k\_file.txt\" & \"two\_k\_file.txt\". \n");

printf("User must request one of these two files for <file\_name> argument\n");

exit(0);

} else

{

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("UDP test client program. Requesting %s file from server.\n",argv[3]);

printf("File will be stored in the client local current directory as \"test\_file.txt\"\n");

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

sock = createsock(argv[1], argv[2], "udp", &remote\_server);

if (sock < 0){

printf("Failed to create socket.\n");

exit(-1);

}

// connection successfuly established

printf("Sending datagram requesting file transfer..\n");

memset(input,0,BUFFER);

strcpy(input, argv[3]);

//input[strlen(input)] = '\n';

// send file number

if (0 > sendto(sock, input, strlen(input), 0, (struct sockaddr \*)&remote\_server,sizeof(remote\_server)))

{

perror("send error");

close(sock);

exit(1);

}

// open file for writing

fd = open("test\_file.txt", O\_CREAT|O\_TRUNC|O\_WRONLY, S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);

if(fd < 0){

printf("Failed to create file\n");

exit(1);

}

addr\_len = sizeof(remote\_server);

//receive file and write it to the file

while(1)

{

memset(output, 0, BUFFER);

len = recvfrom(sock, output, BUFFER, 0, (struct sockaddr \*)&remote\_server, &addr\_len);

//printf("recvfrom len = %d\n", len);

if (len < 0)

{

perror("recv error");

close(fd);

close(sock);

exit(1);

}

if (len == 0)

break;

if (output[0] == '') {

if (len > 1) {

output[0]=' ';

output[len]='\0';

printf("%s\n", output);

close(sock);

exit(-1);

}

//close(sock);

//exit(-1);

break;

}

//write to file

len = write(fd, output, strlen(output));

//printf("write len = %d\n", len);

if(len < 0) {

printf("File write error..exiting..\n");

close(fd);

close(sock);

exit(1);

}

}

close(fd);

//close client socket

printf("File transfer successful. Closing the socket.Exiting the child.\n");

close(sock);

}

}

**Output**

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/UDP\_fork\_server\_client$ ./client 127.0.0.1 10000 one\_k\_file.txt

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

UDP test client program. Requesting one\_k\_file.txt file from server.

File will be stored in the client local current directory as "test\_file.txt"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Sending datagram requesting file transfer..

File transfer successful. Closing the socket.Exiting the child.

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/UDP\_fork\_server\_client$ ./client 127.0.0.1 10000 t\_k\_file.txt

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

UDP test client program. Requesting t\_k\_file.txt file from server.

File will be stored in the client local current directory as "test\_file.txt"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Sending datagram requesting file transfer..

Error: Requested File does not exist on the server.

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/UDP\_fork\_server\_client$ cat test\_file.txt

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/UDP\_fork\_server\_client$ ./client 127.0.0.1 10000 one\_k\_file.txt

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

UDP test client program. Requesting one\_k\_file.txt file from server.

File will be stored in the client local current directory as "test\_file.txt"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Sending datagram requesting file transfer..

File transfer successful. Closing the socket.Exiting the child.

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/UDP\_fork\_server\_client$ cat test\_file.txt

start test\_file\_1

Given a pathname for a file, open() returns a file descriptor, a small, non-negative integer for use in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.). The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

The new file descriptor is set to remain open across an execve(2) (i.e., the FD\_CLOEXEC file descriptor flag described in fcntl(2) is initially disabled). The file offset is set to the beginning of the file (see lseek(2)).

A call to open() creates a new open file description, an entry in the system-wide table of open files. This entry records the file offset and the file status flags (modifiable via the fcntl() F\_SETFL operation). A file descriptor is a reference to one of these entries; this reference is unaffected if pathname is subsequently removed or modified to refer to a different file. The new open file description is initially not shared with any other process, but sharing may arise via fork(2).

end test\_file\_1

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/UDP\_fork\_server\_client$

**3.8 Concurrent multithreading UDP File server with one thread per request - Test client**

**Readme.txt**

Steps to execute the program:  
  
1. Open the terminal  
2. Navigate to the program directory  
3. Build server & client:  
 make MakeFile2.txt - for server  
 make MakeFile2Client.txt - for client  
  
4. Run server and client on two different terminals:  
 ./server <port>  
 ex: ./server 10000   
  
 ./client <server\_IP> <port>  
 ex: ./client 127.0.0.1 10000  
  
5. Enter ^C to quit running the server.

**Makefile**

all:  
 gcc -o client3 udp\_test\_client.c  
clean:  
 rm -f client3

**Source files**

// CMPE\_207 Assignment 2  
// UDP Client  
// Author - Team #2  
// This is a UDP test client program. Client connects  
// the server. Requests a specific file (1K or 2K file)  
// from the server. Save the file on the local disk.  
// To compile: gcc udp\_test\_client.c -o client  
// To run: ./client <server\_address> <server\_port> <file\_name>  
  
  
#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<sys/socket.h>  
#include<sys/types.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
#include<netinet/in.h>  
#include<string.h>  
#include<ctype.h>  
#include<arpa/inet.h>  
#include<netdb.h>  
#include<sys/wait.h>  
  
#define ERROR -1  
#define BUFFER 1024  
  
int createsock(const char \*host, const char \*service, const char \*transport, struct sockaddr\_in \*remote\_server)  
{  
 struct hostent \*phe;  
 struct servent \*pse;  
 struct protoent \*ppe;  
 //struct sockaddr\_in remote\_server;  
 int sock, type;  
  
 //reset socket address structure.  
 memset(remote\_server,0,sizeof(struct sockaddr\_in));  
  
 remote\_server->sin\_family = AF\_INET;  
  
 //map service name to port number  
 if(pse = getservbyname(service, transport))  
 remote\_server->sin\_port = pse->s\_port;  
 else if((remote\_server->sin\_port = htons((u\_short)atoi(service))) == 0)  
 //perror("Cant get \"%s\" service entry\n", service);  
 perror("Cant get service entry\n");  
  
 //map host name to IP address allowing dotted decimal  
 if(phe = gethostbyname(host))  
 memcpy(&((\*remote\_server).sin\_addr), phe->h\_addr, phe->h\_length);  
 else if ((remote\_server->sin\_addr.s\_addr = inet\_addr(host) == INADDR\_NONE))  
 //perror("Cant get \"%s\" host entry\n", host);  
 perror("Cant get host entry\n");  
  
 bzero(remote\_server->sin\_zero, 8);  
  
 //map transport protocol name to protocol number  
 ppe = getprotobyname(transport);  
 if((ppe == NULL))  
 //perror("cant get \"%s\" protocol entry\n", transport);  
 perror("Cant get protocol entry\n");  
  
 //use protocol to chose a socket type  
 if(strcmp(transport, "udp") == 0)  
 type = SOCK\_DGRAM;  
 else  
 type = SOCK\_STREAM;  
  
 //allocate a socket  
 sock = socket(PF\_INET, type, ppe->p\_proto);  
 if(sock < 0)  
 //perror("Cant create socket: %s\n", strerror(errno));  
 perror("Cant create socket\n");  
  
 return sock;  
}  
  
  
int main(int argc, char \*\*argv)  
{  
 int sock;  
 int c, fd;  
 char input[BUFFER];  
 char output[BUFFER];  
 int len;   
 struct sockaddr\_in remote\_server;  
 socklen\_t addr\_len;  
 if(argc < 4)  
 {  
 printf("Usage: ./client <server IP> <server port> <file\_name> \n");  
 printf("Server hosts \"one\_k\_file.txt\" & \"two\_k\_file.txt\". \n");  
 printf("User must request one of these two files for <file\_name> argument\n");  
 exit(0);  
 } else  
 {  
 printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("UDP test client program. Requesting %s file from server.\n",argv[3]);  
 printf("File will be stored in the client local current directory as \"test\_file.txt\"\n");  
 printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
   
 sock = createsock(argv[1], argv[2], "udp", &remote\_server);  
 if (sock < 0){  
 printf("Failed to create socket.\n");  
 exit(-1);  
 }  
  
 // connection successfuly established  
 printf("Sending datagram requesting file transfer..\n");  
  
 memset(input,0,BUFFER);  
 strcpy(input, argv[3]);  
 //input[strlen(input)] = '\n';  
  
 // send file number  
 if (0 > sendto(sock, input, strlen(input), 0, (struct sockaddr \*)&remote\_server,sizeof(remote\_server)))  
 {  
 perror("send error");  
 close(sock);  
 exit(1);  
 }  
  
 // open file for writing  
 fd = open("test\_file.txt", O\_CREAT|O\_TRUNC|O\_WRONLY, S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);  
 if(fd < 0){  
 printf("Failed to create file\n");  
 exit(1);  
 }  
  
 addr\_len = sizeof(remote\_server);  
 //receive file and write it to the file   
 while(1)  
 {  
   
 memset(output, 0, BUFFER);  
 len = recvfrom(sock, output, BUFFER, 0, (struct sockaddr \*)&remote\_server, &addr\_len);  
 //printf("recvfrom len = %d\n", len);  
 if (len < 0)  
 {  
 perror("recv error");  
 close(fd);  
 close(sock);  
 exit(1);  
 }  
 if (len == 0)  
 break;  
  
 if (output[0] == ' ') {  
 if (len > 1) {  
 output[0]=' ';  
 output[len]='\0';  
 printf("%s\n", output);  
 close(sock);  
 exit(-1);  
 }  
 //close(sock);  
 //exit(-1);  
 break;  
 }  
  
 //write to file  
 len = write(fd, output, strlen(output));  
 //printf("write len = %d\n", len);  
 if(len < 0) {  
 printf("File write error..exiting..\n");  
 close(fd);  
 close(sock);  
 exit(1);  
 }  
 }  
  
 close(fd);  
 //close client socket  
 printf("File transfer successful. Closing the socket.Exiting the child.\n");  
 close(sock);  
 }  
}

**Output**

ubuntu@ubuntu:~/Documents/UDP\_thread\_server\_client\_old$ ./client3 127.0.0.1 10000 one\_k\_file.txt  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
UDP test client program. Requesting one\_k\_file.txt file from server.  
File will be stored in the client local current directory as "test\_file.txt"  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Sending datagram requesting file transfer..  
File transfer successful. Closing the socket.Exiting the child.  
  
File name matched: one\_k\_file.txt  
File transfer done.Thread Exiting..

**3.9 Concurrent pre-forked UDP multiprocessing server - Test client**

**Readme.txt**

Steps to execute the program:

1. Open the terminal

2. Navigate to the program directory

3. Build server & client:

make all

Build server alone:

make server

Build client alone:

make client

Clean all targets:

make clean

5. Run server and client on two different terminals:

./server <port>

ex: ./server 10000

./client <server\_IP> <port>

ex: ./client 127.0.0.1 10000

6. Enter ^C to quit running the server.

UDP prefork Server: Server program creates 7 child processes in advance (process pool)to handle the incoming client request.When

a datagram is received from the client, active child process at the server sends back the response (total number of child processes

preforked at server side).

UDP test client: client connects the server and receives number of preforked children at server side and prints it to the screen.

**Makefile**

# Make file targets to build UDP server and clients.

# Format

# target: dependencies

# action

.PHONY: client

all: client

client:

gcc udp\_test\_client.c -o client

clean:

rm client

**Source files**

**Udp\_test\_client.c**

// CMPE\_207 Assignment\_2

// UDP test Client

// Author - Team #2

// This is a TCP test client program. It connects to the server

// and receives total number of processes preforked at the

// server side. It then prints the number on the output screen.

// To compile: gcc udp\_test\_client.c -o client

// To run: ./client <server\_address> <server\_port>

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/socket.h>

#include<sys/types.h>

#include<netinet/in.h>

#include<string.h>

#include<ctype.h>

#include<arpa/inet.h>

#include<netdb.h>

#include<sys/wait.h>

#define ERROR -1

#define BUFFER 1024

int createsock(const char \*host, const char \*service, const char \*transport,struct sockaddr\_in \*remote\_server)

{

struct hostent \*phe;

struct servent \*pse;

struct protoent \*ppe;

int sock, type;

//reset socket address structure.

memset(remote\_server,0,sizeof(struct sockaddr\_in));

remote\_server->sin\_family = AF\_INET;

//map service name to port number

if((pse = getservbyname(service, transport)))

remote\_server->sin\_port = pse->s\_port;

else if((remote\_server->sin\_port = htons((u\_short)atoi(service))) == 0)

//perror("Cant get \"%s\" service entry\n", service);

perror("Cant get service entry\n");

//map host name to IP address allowing dotted decimal

if((phe = gethostbyname(host)))

memcpy(&((\*remote\_server).sin\_addr), phe->h\_addr, phe->h\_length);

else if ((remote\_server->sin\_addr.s\_addr = inet\_addr(host) == INADDR\_NONE))

//perror("Cant get \"%s\" host entry\n", host);

perror("Cant get host entry\n");

bzero(remote\_server->sin\_zero, 8);

//map transport protocol name to protocol number

ppe = getprotobyname(transport);

if(ppe == NULL)

//perror("cant get \"%s\" protocol entry\n", transport);

perror("Cant get protocol entry\n");

//use protocol to chose a socket type

if(strcmp(transport, "udp") == 0)

type = SOCK\_DGRAM;

else

type = SOCK\_STREAM;

//allocate a socket

sock = socket(PF\_INET, type, ppe->p\_proto);

if(sock < 0)

//perror("Cant create socket: %s\n", strerror(errno));

perror("Cant create socket\n");

return sock;

}

int main(int argc, char \*\*argv)

{

int sock;

socklen\_t addr\_size;

int len, bytes\_read;

char output[BUFFER];

char input[BUFFER];

struct sockaddr\_in remote\_server;

if(argc < 3)

{

printf("Usage: ./client <server IP> <server port> \n");

exit(0);

}

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("UDP test client: Connects to the concurrent preforked UDP server\n");

printf("receives number of preforked child processes at the server and prints it.\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

//Create UDP socket endpoint

sock = createsock(argv[1], argv[2], "udp",&remote\_server);

if (sock < 0){

printf("Socket creation failed\n");

exit(-1);

}

// send a datagram

memset(input, '1', BUFFER);

if (0 > sendto(sock, input, strlen(input), 0,(struct sockaddr \*)&remote\_server, sizeof(remote\_server)))

{

perror("sendto error");

close(sock);

exit(-1);

}

printf("Request sent......waiting for server's response\n");

memset(output, 0, BUFFER);

addr\_size = sizeof(remote\_server);

// receive server's response and print.

len = recvfrom(sock, output, BUFFER, 0,(struct sockaddr \*)&remote\_server, &addr\_size);

if (len < 0)

{

perror("recv error..");

exit(-1);

}

output[len] = '\0';

printf("%s", output);

close(sock);

}

**Output**

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/UDP\_prefork\_server\_client$ ./client 127.0.0.1 10000

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

UDP test client: Connects to the concurrent preforked UDP server

receives number of preforked child processes at the server and prints it.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Request sent......waiting for server's response

Child pid 4753: Number of preforked processes at is server is 7

kanti@kanti-linux:~/kanti\_sem\_3/207/Assgn\_2/UDP\_prefork\_server\_client$

**3.10 Concurrent pre-threaded UDP multithreading server - Test client**

**Readme.txt**

Steps to execute the program:  
  
1. Open the terminal  
2. Navigate to the program directory  
3. Build server & client:  
 make MakeFile3.txt - for server  
 make MakeFile3Client.txt - for client  
4. Run server and client on two different terminals:  
 ./server <port> <Number of threads>  
 ex: ./server 10000 4  
  
 ./client <server\_IP> <port>  
 ex: ./client 127.0.0.1 10000  
  
5. Enter ^C to quit running the server.

**Makefile**

all:  
 gcc -o client4 udp\_test\_client\_prethreading.c  
clean:  
 rm -f client4

**Source files**

// CMPE\_207 Assignment 2  
// UDP Client  
// Author - Team #2  
// This is a UDP test client program. Client connects  
// the server. Requests a specific file (1K or 2K file)  
// from the server. Save the file on the local disk.  
// To compile: gcc udp\_test\_client.c -o client  
// To run: ./client <server\_address> <server\_port> <file\_name>  
  
  
#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<sys/socket.h>  
#include<sys/types.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
#include<netinet/in.h>  
#include<string.h>  
#include<ctype.h>  
#include<arpa/inet.h>  
#include<netdb.h>  
#include<sys/wait.h>  
  
#define ERROR -1  
#define BUFFER 1024  
  
int createsock(const char \*host, const char \*service, const char \*transport, struct sockaddr\_in \*remote\_server)  
{  
 struct hostent \*phe;  
 struct servent \*pse;  
 struct protoent \*ppe;  
 int sock, type;  
  
 //reset socket address structure.  
 memset(remote\_server,0,sizeof(struct sockaddr\_in));  
  
 remote\_server->sin\_family = AF\_INET;  
  
 //map service name to port number  
 if(pse = getservbyname(service, transport))  
 remote\_server->sin\_port = pse->s\_port;  
 else if((remote\_server->sin\_port = htons((u\_short)atoi(service))) == 0)  
 perror("Cant get service entry\n");  
  
 //map host name to IP address allowing dotted decimal  
 if(phe = gethostbyname(host))  
 memcpy(&((\*remote\_server).sin\_addr), phe->h\_addr, phe->h\_length);  
 else if ((remote\_server->sin\_addr.s\_addr = inet\_addr(host) == INADDR\_NONE))  
 perror("Cant get host entry\n");  
  
 bzero(remote\_server->sin\_zero, 8);  
  
 //map transport protocol name to protocol number  
 ppe = getprotobyname(transport);  
 if((ppe == NULL))  
 perror("Cant get protocol entry\n");  
  
 //use protocol to chose a socket type  
 if(strcmp(transport, "udp") == 0)  
 type = SOCK\_DGRAM;  
 else  
 type = SOCK\_STREAM;  
  
 //allocate a socket  
 sock = socket(PF\_INET, type, ppe->p\_proto);  
 if(sock < 0)  
 perror("Cant create socket\n");  
  
 return sock;  
}  
  
  
int main(int argc, char \*\*argv)  
{  
 int sock;  
 int c, fd;  
 char input[BUFFER];  
 char output[BUFFER];  
 int len;   
 struct sockaddr\_in remote\_server;  
 socklen\_t addr\_len;  
 if(argc < 4)  
 {  
 printf("Usage: ./client <server IP> <server port> <file\_name> \n");  
 printf("Server hosts \"one\_k\_file.txt\" & \"two\_k\_file.txt\". \n");  
 printf("User must request one of these two files for <file\_name> argument\n");  
 exit(0);  
 } else  
 {  
 printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("UDP test client program. Requesting %s file from server.\n",argv[3]);  
 printf("File will be stored in the client local current directory as \"test\_file.txt\"\n");  
 printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
   
 sock = createsock(argv[1], argv[2], "udp", &remote\_server);  
 if (sock < 0){  
 printf("Failed to create socket.\n");  
 exit(-1);  
 }  
  
 // connection successfuly established  
 printf("Sending datagram requesting file transfer..\n");  
  
 memset(input,0,BUFFER);  
 strcpy(input, argv[3]);  
  
 // send file number  
 if (0 > sendto(sock, input, strlen(input), 0, (struct sockaddr \*)&remote\_server,sizeof(remote\_server)))  
 {  
 perror("send error");  
 close(sock);  
 exit(1);  
 }  
  
 // open file for writing  
 fd = open("test\_file.txt", O\_CREAT|O\_TRUNC|O\_WRONLY, S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);  
 if(fd < 0){  
 printf("Failed to create file\n");  
 exit(1);  
 }  
  
 addr\_len = sizeof(remote\_server);  
 //receive file and write it to the file   
 while(1)  
 {  
   
 memset(output, 0, BUFFER);  
 len = recvfrom(sock, output, BUFFER, 0, (struct sockaddr \*)&remote\_server, &addr\_len);  
 if (len < 0)  
 {  
 perror("recv error");  
 close(fd);  
 close(sock);  
 exit(1);  
 }  
 if (len == 0)  
 break;  
  
 if (output[0] == ' ') {  
 if (len > 1) {  
 output[0]=' ';  
 output[len]='\0';  
 printf("%s\n", output);  
 close(sock);  
 exit(-1);  
 }  
 //close(sock);  
 //exit(-1);  
 break;  
 }  
  
 //write to file  
 len = write(fd, output, strlen(output));  
  
 if(len < 0) {  
 printf("File write error..exiting..\n");  
 close(fd);  
 close(sock);  
 exit(1);  
 }  
 }  
  
 close(fd);  
 //close client socket  
 printf("File transfer successful. Closing the socket.Exiting the child.\n");  
 close(sock);  
 }  
}

**Output**

ubuntu@ubuntu:~/Documents/UDP\_prethread\_client\_server$ ./client4 127.0.0.1 10000 one\_k\_file.txt  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
UDP test client program. Requesting one\_k\_file.txt file from server.  
File will be stored in the client local current directory as "test\_file.txt"  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Sending datagram requesting file transfer..  
File transfer successful. Closing the socket.Exiting the child.

**Reference:**

1. Internetworking with TCP/IP Vol. 3, Client-Server programming and applications, Comer and Stevens, Linux/POSIX Sockets version, ISBN: 0-13-032071-4, 2001.
2. UNIX Network Programming Vol. 1, 3/e: The Sockets Networking API, Stevens,Fenner & Rudoff, ISBN: 0-13-141155-1,2004.