**CMPE 207 Lab Assignment 2**

**Team 1: Omkar Rege, Bing Shi, Lam Tran**

[**TCP client Code**](#_ms07njcibs63) **3**

[**UDP client Code**](#_eddyc849uxtq) **6**

[**Test File 1 (800 bytes)**](#_mgnkhf8kjwoq) **9**

[**Test File 2 (1600 bytes)**](#_65f9k4wntpft) **9**

[**Connection-oriented file servers**](#_ksx0qhmmi3qv) **10**

[Iterative server](#_5gj6ujqjqr28) 10

[Readme](#_rls6pckwgvw4) 10

[Makefile](#_pj1g9j6ufaxt) 10

[Server Code](#_wq0xkd5anyme) 11

[Concurrent Multiprocessing server with one process per request](#_cc9hkca7vyec) 14

[Readme](#_f2hec3u49o55) 14

[Makefile](#_ipmzjyripj9x) 14

[Server Code](#_r7t8762b2qc0) 15

[Concurrent Multithreading server with one thread per request](#_gyx9ko616cvt) 18

[Readme](#_n9c3qwbl9yny) 18

[Makefile](#_5mboizy0j2wt) 18

[Server Code](#_vex7torwip0q) 19

[Concurrent Pre-forked Multiprocessing server](#_cuxl3xomk196) 23

[Readme](#_ks35nd9t7h4d) 23

[Makefile](#_2ymmb11z65eh) 23

[Server Code](#_i82g5y5022ak) 24

[Concurrent Pre-threaded Multithreading server](#_xc5aj2iuf1xq) 27

[Readme](#_a7fvj6toe9k0) 27

[Makefile](#_3il7pglr1jsb) 27

[Server Code](#_845ebh5phb3c) 28

[**Connection-less file servers (UDP)**](#_axm0o8r9nxjt) **33**

[Iterative server](#_6qmdj8fb70r9) 33

[Readme](#_j5kvsrois698) 33

[Makefile](#_dl2whmd8g7mp) 33

[Server Code](#_pbeccyoumbe8) 34

[Concurrent Multiprocessing server with one process per request](#_7lrny9dmgi4z) 38

[Readme](#_1wn5g49x10e) 38

[Makefile](#_rartq7brt8cf) 38

[Server Code](#_68o4cllowy1y) 39

[Concurrent Multithreading server with one thread per request](#_j789kpvi8wpf) 42

[Readme](#_6q5pcvlq4951) 42

[Makefile](#_xuj2wmvqmap9) 42

[Server Code](#_qgtve3dcssfw) 43

[Concurrent Pre-forked Multiprocessing server](#_jmnfitdjqc8h) 48

[Readme](#_k5f0r7sjyd5u) 48

[Makefile](#_4ad9al6xhhtn) 48

[Server Code](#_gny1vlxky1e) 49

[Concurrent Pre-threaded Multithreading server](#_1vd20kynp5cp) 52

[Readme](#_6fntdx7hea02) 52

[Makefile](#_sb0l8xhbvvnd) 52

[Server Code](#_9ksgso498j4p) 53

# **TCP client Code**

(./TCPClient.c)

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

#include <stdarg.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <ctype.h>

#define LINELEN 4096

#ifndef INADDR\_NONE

#define INADDR\_NONE 0xffffffff

#endif /\* INADDR\_NONE \*/

int requestFile(const char \*host, const char \*service, const char \*filename);

int errexit(const char \*format, ...);

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

const char \*transport);

int main(int argc, char \*argv[]) {

char \*fileName = NULL;

char \*host = "localhost"; /\* host to use if none supplied \*/

char \*service = "9200"; /\* default service name \*/

switch (argc) {

case 2:

fileName = argv[1];

break;

default:

fprintf(stderr, "usage: TCPCLient [filename]\n");

exit(1);

}

requestFile(host, service, fileName);

exit(0);

}

int requestFile(const char \*host, const char \*service, const char \*filename) {

char buf[LINELEN + 1]; /\* buffer for one line of text \*/

int s, n; /\* socket descriptor, read count\*/

char fn[sizeof(filename) + 2];

FILE \*file;

char const \*path = "clientfiles/";

char \*fullpath[sizeof(path) + sizeof(filename)];

strcat(fullpath, path);

strcat(fullpath, filename);

strcpy(fn, filename);

fn[sizeof(fn) - 1] = '\0';

s = connectsock(host, service, "tcp");

(void) write(s, fn, sizeof(fn));

file = fopen(fullpath, "w+");

while ((n = read(s, buf, sizeof(buf))) > 0) {

fputs(buf, stdout);

fputs(buf, file);

memset(buf, 0, sizeof(buf));

}

fclose(file);

}

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

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 = pse->s\_port;

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;

}

int errexit(const char \*format, ...) {

va\_list args;

va\_start(args, format);

vfprintf(stderr, format, args);

va\_end(args);

exit(1);

}

# **UDP client Code**

(./UDPClient.c)

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

#include <stdarg.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <ctype.h>

#define LINELEN 4096

#ifndef INADDR\_NONE

#define INADDR\_NONE 0xffffffff

#endif /\* INADDR\_NONE \*/

int requestFile(const char \*host, const char \*service, const char \*filename);

int errexit(const char \*format, ...);

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

const char \*transport, struct sockaddr\_in \*fsin);

int main(int argc, char \*argv[]) {

char \*fileName = NULL;

char \*host = "localhost"; /\* host to use if none supplied \*/

char \*service = "9200"; /\* default service name \*/

switch (argc) {

case 2:

fileName = argv[1];

break;

default:

fprintf(stderr, "usage: UDPCLient [filename]\n");

exit(1);

}

requestFile(host, service, fileName);

exit(0);

}

int requestFile(const char \*host, const char \*service, const char \*filename) {

int s, n; /\* socket descriptor, read count\*/

char fn[sizeof(filename) + 2];

FILE \*file;

char buf[LINELEN + 1]; /\* buffer for one line of text \*/

char const \*path = "clientfiles/";

char \*fullpath[sizeof(path) + sizeof(filename)];

struct sockaddr\_in sin;

struct sockaddr\_in ser; /\* the address of a client \*/

socklen\_t addrlen = sizeof(ser); /\* length of addresses \*/

strcat(fullpath, path);

strcat(fullpath, filename);

strcpy(fn, filename);

fn[sizeof(fn) - 1] = '\0';

s = connectsock(host, service, "udp", &sin);

if (n = (sendto(s, fn, sizeof(fn), 0, (struct sockaddr \*) &sin, sizeof(sin))) < 0) {

errexit("Fail to send the file name: %s", fn);

}

memset(buf, 0, sizeof buf);

int filesize = 0;

char sizeBuf[LINELEN];

recvfrom(s, sizeBuf, sizeof(sizeBuf), 0, (struct sockaddr \*) &ser, &addrlen);

filesize = atoi(sizeBuf);

// filesize = ntohl(filesize);

// printf("------------>file size is %s", sizeBuf);

fflush(stdout);

memset(buf, 0, sizeof buf);

file = fopen(fullpath, "w+");

// printf("1----- %d", filesize);

fflush(stdout);

while (filesize > 0) {

int n = recvfrom(s, buf, sizeof(buf), 0, (struct sockaddr \*) &ser, &addrlen);

filesize -= n;

printf("----- %d", n);

fputs(buf, stdout);

fputs(buf, file);

fflush(stdout);

memset(buf, 0, sizeof buf);

} /\*else {

errexit("Fail to receive in the server ");

}\*/

fclose(file);

}

int connectsock(const char \*host, const char \*service, const char \*transport, struct sockaddr\_in \*fsin) {

struct sockaddr\_in sin; /\* an Internet endpoint address \*/

struct hostent \*phe; /\* pointer to host information entry \*/

struct servent \*pse; /\* pointer to service information entry \*/

struct protoent \*ppe; /\* pointer to protocol information entry\*/

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 = pse->s\_port;

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);

/\* Allocate a socket \*/

s = socket(PF\_INET, SOCK\_DGRAM, ppe->p\_proto);

if (s < 0)

errexit("can't create socket: %s\n", strerror(errno));

\*fsin = sin;

return s;

}

int errexit(const char \*format, ...) {

va\_list args;

va\_start(args, format);

vfprintf(stderr, format, args);

va\_end(args);

exit(1);

}

# **Test File 1 (800 bytes)**

(./files/file1.txt)

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

# **Test File 2 (1600 bytes)**

(./files/file2.txt)

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

# **Connection-oriented file servers**

## ***Iterative server***

### Readme

To compile:

1. Open terminal at code folder
2. make

OR

1. Open terminal at code folder
2. gcc -o server TCPIterativeServer.c

gcc -o client TCPClient.c

To run:

1. make runs
2. Open another terminal at code folder
3. make runc1
4. make runc2

OR

1. Run server: ./server
2. Open another terminal at code folder
3. Run client: ./client file1.txt
4. Run client: ./client file2.txt

To stop server:

1. Ctrl + C

To clean the files:

1. make clean

OR

2. rm -f server client

### Makefile

all: server client

server:

gcc -o server TCPIterativeServer.c

client:

gcc -o client TCPClient.c

runs:

./server

runc1:

./client file1.txt

runc2:

./client file2.txt

clean:

rm -f server client

### Server Code

(./TCPIterativeServer.c)

#include <sys/types.h>

#include <sys/signal.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <sys/resource.h>

#include <sys/wait.h>

#include <sys/errno.h>

#include <netinet/in.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <netdb.h>

#include <stdarg.h>

#include <arpa/inet.h>

#define QLEN 5 /\* maximum connection queue length \*/

#define BUFSIZE 4096

extern int errno;

u\_short portbase = 0; /\* port base, for non-root servers \*/

int errexit(const char \*format, ...);

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

int passiveTCP(const char \*service, int qlen);

int main(int argc, char \*argv[]) {

char \*service = "9200"; /\* service name or port number \*/

struct sockaddr\_in fsin; /\* the address of a client \*/

int alen; /\* length of client's address \*/

int msock; /\* master server socket \*/

int ssock; /\* slave server socket \*/

char buf[BUFSIZ];

char fileBuf[BUFSIZ];

int n;

char const \*filepath = "files/";

FILE \*file;

msock = passiveTCP(service, QLEN);

while (1) {

ssock = accept(msock, (struct sockaddr \*)&fsin, &alen);

if (ssock < 0)

errexit("accept failed: %s\n", strerror(errno));

if (read(ssock, buf, BUFSIZ) < 0) {

errexit("Fail to read from Socket");

}

char fullpath[sizeof(filepath) + sizeof(buf) +1];

strcpy(fullpath, filepath);

strcat(fullpath, buf);

strcat(fullpath, "\0");

file = fopen(fullpath, "r");

n = 0;

if (file) {

while ((n = fread(fileBuf, 1, sizeof(fileBuf) - 1, file)) > 0) {

fileBuf[n] = '\0';

printf("------====== %s", fileBuf);

fflush(stdout);

write(ssock, fileBuf, n + 1);

}

fclose(file);

close(ssock);

} else {

errexit("File %s does not exist", fullpath);

}

}

}

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

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(ntohs((u\_short) pse->s\_port) + portbase);

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 passiveTCP(const char \*service, int qlen){

return passivesock(service, "tcp", qlen);

}

/\*--------------------------------------------------HELPER--------------------------------------------------\*/

int errexit(const char \*format, ...) {

va\_list args;

va\_start(args, format);

vfprintf(stderr, format, args);

va\_end(args);

exit(1);

}

## Concurrent Multiprocessing server with one process per request

### Readme

To compile:

1. Open terminal at code folder
2. make

OR

1. Open terminal at code folder
2. gcc -o server TCPConProcessServer.c

gcc -o client TCPClient.c

To run:

1. make runs
2. Open another terminal at code folder
3. make runc1
4. make runc2

OR

1. Run server: ./server
2. Open another terminal at code folder
3. Run client: ./client file1.txt
4. Run client: ./client file2.txt

To stop server:

1. Ctrl + C

To clean the files:

1. make clean

OR

1. rm -f server client

### Makefile

all: server client

server:

gcc -o server TCPConProcessServer.c

client:

gcc -o client TCPClient.c

runs:

./server

runc1:

./client file1.txt

runc2:

./client file2.txt

clean:

rm -f server client

### Server Code

(./TCPConProcessServer.c)

#include <sys/types.h>

#include <sys/signal.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <sys/resource.h>

#include <sys/wait.h>

#include <sys/errno.h>

#include <netinet/in.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <netdb.h>

#include <stdarg.h>

#include <arpa/inet.h>

#define QLEN 5 /\* maximum connection queue length \*/

#define BUFSIZE 4096

u\_short portbase = 0; /\* port base, for non-root servers \*/

void reaper(int);

int processRequest(int fd);

int errexit(const char \*format, ...);

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

int main(int argc, char \*argv[]) {

char \*service = "9200"; /\* service name or port number \*/

struct sockaddr\_in fsin; /\* the address of a client \*/

int alen; /\* length of client's address \*/

int msock; /\* master server socket \*/

int ssock; /\* slave server socket \*/

msock = passivesock(service, "tcp", QLEN);

(void) signal(SIGCHLD, reaper);

while (1) {

alen = sizeof(fsin);

ssock = accept(msock, (struct sockaddr \*) &fsin, &alen);

if (ssock < 0) {

if (errno == EINTR)

continue;

errexit("ACCEPT: %s\n", strerror(errno));

}

switch (fork()) {

case 0: /\* child \*/

(void) close(msock);

exit(processRequest(ssock));

default: /\* parent \*/

(void) close(ssock);

break;

case -1:

errexit("fork: %s\n", strerror(errno));

}

}

}

int processRequest(int fd) {

char buf[BUFSIZ];

char fileBuf[BUFSIZ];

int n;

int count = 0;

char const \*filepath = "files/";

FILE \*file;

if (read(fd, buf, BUFSIZ) < 0) {

errexit("Fail to read from Socket");

}

char fullpath[sizeof(filepath) + sizeof(buf) +1];

strcpy(fullpath, filepath);

strcat(fullpath, buf);

strcat(fullpath, "\0");

file = fopen(fullpath, "r");

n = 0;

if (file) {

while ((n = fread(fileBuf, 1, sizeof(fileBuf) - 1, file)) > 0) {

fileBuf[n] = '\0';

printf("------====== %s", fileBuf);

fflush(stdout);

write(fd, fileBuf, n + 1);

}

fclose(file);

close(fd);

} else {

errexit("File %s does not exist", fullpath);

}

return 0;

}

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

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(ntohs((u\_short) pse->s\_port) + portbase);

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

errexit("can't get \"%s\" service entry\n", service);

if ((ppe = getprotobyname(transport)) == 0)

errexit("can't get \"%s\" protocol entry\n", transport);

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

type = SOCK\_DGRAM;

else

type = SOCK\_STREAM;

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

if (s < 0)

errexit("can't create socket: %s\n", strerror(errno));

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;

}

/\*--------------------------------------------------HELPER--------------------------------------------------\*/

void reaper(int sig) {

int status;

while (wait3(&status, WNOHANG, (struct rusage \*) 0) >= 0)

/\* empty \*/;

}

int errexit(const char \*format, ...) {

va\_list args;

va\_start(args, format);

vfprintf(stderr, format, args);

va\_end(args);

exit(1);

}

## *Concurrent Multithreading server with one thread per request*

### Readme

To compile:

1. Open terminal at code folder
2. make

OR

1. Open terminal at code folder
2. gcc -pthread -o server TCPConThreadServer.c

gcc -o client TCPClient.c

To run:

1. make runs
2. Open another terminal at code folder
3. make runc1
4. make runc2

OR

1. Run server: ./server
2. Open another terminal at code folder
3. Run client: ./client file1.txt
4. Run client: ./client file2.txt

To stop server:

1. Ctrl + C

To clean the files:

1. make clean

OR

1. rm -f server client

### Makefile

all: server client

server:

gcc -pthread -o server TCPConThreadServer.c

client:

gcc -o client TCPClient.c

runs:

./server

runc1:

./client file1.txt

runc2:

./client file2.txt

clean:

rm -f server client

### Server Code

(./TCPConThreadServer.c)

//file transfer server for testing FTP hope it will be easy :-)

//This is concurrent multithreading server

/\*\*

author : Omkar M. Rege

This is a simple TCP server program that accepts 2 parameters,

1. IP on which it should run

2. Port which it should use

3. The server return the length of the string of client's greeting message back to the client

4. For example: USAGE : ./tcpserver.o localhost 3333

\*/

//Common header

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdarg.h>

#include <string.h>

#include <strings.h>

#include <errno.h>

#include <stdbool.h>

//Network specific

#include <sys/socket.h>

#include <sys/types.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

//thread library

#include<pthread.h>

int definelisteningsocket(char \*ip);

void \*filehandler(void \*arg);

int getFileSize(FILE \*fp);

int main(int argc, char \*argv[]) {

int lsd, connsocksd, connsocklength;

struct sockaddr\_in connsock;

lsd = definelisteningsocket("localhost");

pthread\_t clientthread;

fflush(stdout);

printf("\nreturned from definelisteningsocket\n");

while (1) {

connsocklength = sizeof connsock;

connsocksd = accept(lsd, (struct sockaddr \*) &connsock, &connsocklength);

if (connsocksd > 0) {

if (pthread\_create(&clientthread, NULL, filehandler, (void \*) &connsocksd) != 0) {

printf("error is spawning thread");

}

printf("A thread has been spawned!!!");

fflush(stdout);

} else {

printf("error in creating client socket by accept");

}

}

}//end of main

int definelisteningsocket(char \*ip) {

int lsocksd;

struct sockaddr\_in listensock;

struct servent \*service;

struct protoent \*protocol;

memset(&listensock, 0, sizeof listensock);

listensock.sin\_family = AF\_INET;

//first we will convert ip from ascii string to network address

if (inet\_aton(ip, &listensock.sin\_addr) != 0) {

printf("Your IP adrress is weird %s", ip);

} else if (strcasecmp("localhost", ip) == 0) {

printf("You have a Localhost");

if (inet\_aton("127.0.0.1", &listensock.sin\_addr) == 0) {

printf("It's better to leave this world!");

exit(1);

}

} else { //listening to any/all incoming address

printf("Now I can accept anything since you do not know IPV4");

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

}

printf("server address is %s", inet\_ntoa(listensock.sin\_addr));

//getting port from service or servent

listensock.sin\_port = htons(9200);

//getting protocol by name

if ((protocol = getprotobyname("tcp")) == 0) {

printf("Can not get protocol");

}

lsocksd = socket(PF\_INET, SOCK\_STREAM, protocol->p\_proto);

//its like connect for client

if (bind(lsocksd, (struct sockaddr \*) &listensock, sizeof listensock) < 0) {

printf("\n Something wrong with bind");

exit(1);

}

if (listen(lsocksd, 5) < 0) {

printf("Something went wrong with lock");

exit(1);

}

return lsocksd;

}

void \*filehandler(void \*arg) {

fflush(stdout);

printf("This is thread");

const char \*const DIR\_PATH = "./files/";

int connsocksd = \*((int \*) arg);

fflush(stdout);

printf("This is thread");

int bytesread = 0, cnt = 0;

char buffer[128];

char \*message, \*temp;

char ch;

bool isFirstAttempt = true;

if ((bytesread = read(connsocksd, buffer, sizeof buffer)) > 0) {

cnt = cnt + bytesread;

if (isFirstAttempt == true) {

message = (char \*) malloc(bytesread + 1);

memcpy(message, buffer, bytesread);

message[bytesread] = '\0';

isFirstAttempt = false;

}

} //end reading

printf("\nCount is %d \n", cnt);

fflush(stdout);

//writting part

printf("File name client sent is %s \n", message);

fflush(stdout);

//resetting buffer and counter

memset(buffer, 0, sizeof buffer);

cnt = 0;

char filename[128];

memset(filename, 0, 128);

strcpy(filename, DIR\_PATH);

strcat(filename, message);

FILE \*fp = fopen(filename, "r");

if (fp == NULL) {

write(connsocksd, "FILE DOES NOT EXIST", 21);

close(connsocksd);

exit(0);

}

int totalwritten = 0, bytesWritten = 0;

while (1) {

ch = fgetc(fp);

if (feof(fp)) {

fflush(stdout);

printf("cnt is %d", cnt);

fflush(stdout);

break;

}

buffer[cnt] = ch;

cnt++;

printf("\n%c %d", ch, cnt);

fflush(stdout);

if (cnt == 128) {

while (cnt != 0) {

bytesWritten = write(connsocksd, buffer + totalwritten, 128 - totalwritten);

totalwritten += bytesWritten;

cnt -= bytesWritten;

}

cnt = 0;

bytesWritten = 0;

totalwritten = 0;

memset(buffer, 0, sizeof buffer); //reset the buffer

}

}

printf("Reached Here");

fflush(stdout);

bytesWritten = 0;

totalwritten = 0;

while (cnt > 0) {

printf("Writting to the client %s:", buffer);

fflush(stdout);

bytesWritten = write(connsocksd, buffer + totalwritten, cnt - totalwritten);

totalwritten += bytesWritten;

cnt -= bytesWritten;

}

fclose(fp);

free(message);

close(connsocksd);

printf("Thread has closed connected socket");

fflush(stdout);

}

## *Concurrent Pre-forked Multiprocessing server*

### Readme

To compile:

1. Open terminal at code folder
2. make

OR

1. Open terminal at code folder
2. gcc -o server TCPConPreForkedServer.c

gcc -o client TCPClient.c

To run:

1. make runs
2. Open another terminal at code folder
3. make runc1
4. make runc2

OR

1. Run server: ./server
2. Open another terminal at code folder
3. Run client: ./client file1.txt
4. Run client: ./client file2.txt

To stop server:

1. Ctrl + C

To clean the files:

1. make clean

OR

1. rm -f server client

### Makefile

all: server client

server:

gcc -o server TCPConPreForkedServer.c

client:

gcc -o client TCPClient.c

runs:

./server

runc1:

./client file1.txt

runc2:

./client file2.txt

clean:

rm -f server client

### Server Code

(./TCPConPreForkedServer.c)

#include <sys/types.h>

#include <sys/signal.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <sys/resource.h>

#include <sys/wait.h>

#include <sys/errno.h>

#include <netinet/in.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <netdb.h>

#include <stdarg.h>

#include <arpa/inet.h>

#define QLEN 5 /\* maximum connection queue length \*/

#define BUFSIZE 4096

#define NUM\_OF\_CHILDREN 5

u\_short portbase = 0; /\* port base, for non-root servers \*/

void reaper(int);

int processRequest(int fd);

int errexit(const char \*format, ...);

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

int main(int argc, char \*argv[]) {

char \*service = "9200"; /\* service name or port number \*/

struct sockaddr\_in fsin; /\* the address of a client \*/

int alen; /\* length of client's address \*/

int msock; /\* master server socket \*/

int ssock; /\* slave server socket \*/

msock = passivesock(service, "tcp", QLEN);

int count = 0;

for (count = 0; count < NUM\_OF\_CHILDREN; count++) {

if (fork() == 0) {

// in the child

while (1) {

alen = sizeof(fsin);

ssock = accept(msock, (struct sockaddr \*) &fsin, &alen);

if (ssock < 0) {

errexit("Could not accept connection");

continue;

}

printf("pid: %d \n", getpid());

processRequest(ssock);

}

}

}

(void) signal(SIGCHLD, reaper);

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

close(msock);

}

int processRequest(int fd) {

char buf[BUFSIZ];

char fileBuf[BUFSIZ];

int n;

int count = 0;

char const \*filepath = "files/";

FILE \*file;

if (read(fd, buf, BUFSIZ) < 0) {

errexit("Fail to read from Socket");

}

char fullpath[sizeof(filepath) + sizeof(buf) + 1];

bzero(fullpath, sizeof(fullpath));

strcat(fullpath, filepath);

strcat(fullpath, buf);

file = fopen(fullpath, "r");

n = 0;

if (file) {

while ((n = fread(fileBuf, 1, sizeof(fileBuf), file)) > 0) {

write(fd, fileBuf, n);

fputs(fileBuf, stdout);

fflush(stdout);

}

close(fd);

fclose(file);

} else {

errexit("File %s does not exist", fullpath);

}

return 0;

}

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

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(ntohs((u\_short) pse->s\_port) + portbase);

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

errexit("can't get \"%s\" service entry\n", service);

if ((ppe = getprotobyname(transport)) == 0)

errexit("can't get \"%s\" protocol entry\n", transport);

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

type = SOCK\_DGRAM;

else

type = SOCK\_STREAM;

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

if (s < 0)

errexit("can't create socket: %s\n", strerror(errno));

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;

}

/\*--------------------------------------------------HELPER--------------------------------------------------\*/

void reaper(int sig) {

int status;

while (wait3(&status, WNOHANG, (struct rusage \*) 0) >= 0)

/\* empty \*/;

}

int errexit(const char \*format, ...) {

va\_list args;

va\_start(args, format);

vfprintf(stderr, format, args);

va\_end(args);

}

## Concurrent Pre-threaded Multithreading server

### Readme

To compile:

1. Open terminal at code folder
2. make

OR

1. Open terminal at code folder
2. gcc -pthread -o server TCPConPreThreadedServer.c

gcc -o client TCPClient.c

To run:

1. make runs
2. Open another terminal at code folder
3. make runc1
4. make runc2

OR

1. Run server: ./server
2. Open another terminal at code folder
3. Run client: ./client file1.txt
4. Run client: ./client file2.txt

To stop server:

1. Ctrl + C

To clean the files:

1. make clean

OR

1. rm -f server client

### Makefile

all: server client

server:

gcc -pthread -o server TCPConPreThreadedServer.c

client:

gcc -o client TCPClient.c

runs:

./server

runc1:

./client file1.txt

runc2:

./client file2.txt

clean:

rm -f server client

### Server Code

(./TCPConPreThreadedServer.c)

//file transfer server for testing FTP hope it will be easy :-)

//This is concurrent multithreading server

/\*\*

author : Omkar M. Rege

This is a simple TCP server program that accepts 2 parameters,

1. IP on which it should run

2. Port which it should use

3. The server return the length of the string of client's greeting message back to the client

4. For example: USAGE : ./tcpserver.o localhost 3333

\*/

//Common header

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdarg.h>

#include <string.h>

#include <strings.h>

#include <errno.h>

#include <stdbool.h>

//Network specific

#include <sys/socket.h>

#include <sys/types.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

//thread library

#include<pthread.h>

#define THREADS 5

int definelisteningsocket(char \*ip);

void \*filehandler(void \*arg);

int main(int argc, char \*argv[]) {

int lsd, connsocksd, connsocklength;

struct sockaddr\_in connsock;

lsd = definelisteningsocket("localhost");

fflush(stdout);

printf("\nreturned from definelisteningsocket\n");

int i = 0;

pthread\_t \*ptr[THREADS];

for (i = 0; i < THREADS; i++) {

pthread\_t clientthread;

int \*newListenSocket = malloc(sizeof(int));

\*newListenSocket = lsd;

if (pthread\_create(&clientthread, NULL, filehandler, (void \*) newListenSocket) != 0) {

printf("error is spawning thread");

}

ptr[i] = &clientthread;

printf("\n A thread has been spawned!!! \n");

fflush(stdout);

}//end of for loop

// for (i = 0; i < THREADS; i++) {

// if (pthread\_join((pthread\_t) ptr[i], NULL) != 0) {

// printf("JOIN HAS FAILED");

// fflush(stdout);

// }

// }

while (1) {

}

}

//end of main

void \*filehandler(void \*arg) {

// fflush(stdout);

// printf("This is thread");

const char \*const DIR\_PATH = "./files/";

int listeningSock = \*((int \*) arg);

while (1) {

struct sockaddr\_in connsock;

int connsocklength = sizeof connsock;

int connsocksd = accept(listeningSock, (struct sockaddr \*) &connsock, &connsocklength);

if (connsocksd <= 0) {

printf("error in creating client socket by accept");

fflush(stdout);

}

int bytesread = 0, cnt = 0;

char buffer[128];

char \*message, \*temp;

char ch;

bool isFirstAttempt = true;

if ((bytesread = read(connsocksd, buffer, sizeof buffer)) > 0) {

cnt = cnt + bytesread;

if (isFirstAttempt == true) {

message = (char \*) malloc(bytesread + 1);

memcpy(message, buffer, bytesread);

message[bytesread] = '\0';

isFirstAttempt = false;

}

} //end reading

printf("File name client sent is %s \n", message);

fflush(stdout);

//writting part

//resetting buffer and counter

memset(buffer, 0, sizeof buffer);

cnt = 0;

char filename[128];

memset(filename, 0, 128);

strcpy(filename, DIR\_PATH);

strcat(filename, message);

FILE \*fp = fopen(filename, "r");

if (fp == NULL) {

write(connsocksd, "FILE DOES NOT EXIST", 21);

close(connsocksd);

exit(0);

}

int totalwritten = 0, bytesWritten = 0;

while (1) {

ch = fgetc(fp);

if (feof(fp)) {

fflush(stdout);

printf("cnt is %d", cnt);

fflush(stdout);

break;

}

buffer[cnt] = ch;

cnt++;

printf("\n%c %d", ch, cnt);

fflush(stdout);

if (cnt == 128) {

while (cnt != 0) {

bytesWritten = write(connsocksd, buffer + totalwritten, 128 - totalwritten);

totalwritten += bytesWritten;

cnt -= bytesWritten;

}

cnt = 0;

bytesWritten = 0;

totalwritten = 0;

memset(buffer, 0, sizeof buffer); //reset the buffer

}

}

printf("Reached Here");

fflush(stdout);

bytesWritten = 0;

totalwritten = 0;

while (cnt > 0) {

printf("Writting to the client %s:", buffer);

fflush(stdout);

bytesWritten = write(connsocksd, buffer + totalwritten, cnt - totalwritten);

totalwritten += bytesWritten;

cnt -= bytesWritten;

}

fclose(fp);

free(message);

close(connsocksd);

printf("Thread has closed connected socket");

fflush(stdout);

}//end of while

}

int definelisteningsocket(char \*ip) {

int lsocksd;

struct sockaddr\_in listensock;

struct servent \*service;

struct protoent \*protocol;

memset(&listensock, 0, sizeof listensock);

listensock.sin\_family = AF\_INET;

//first we will convert ip from ascii string to network address

if (inet\_aton(ip, &listensock.sin\_addr) != 0) {

printf("Your IP adrress is weird %s", ip);

} else if (strcasecmp("localhost", ip) == 0) {

printf("You have a Localhost");

if (inet\_aton("127.0.0.1", &listensock.sin\_addr) == 0) {

printf("It's better to leave this world!");

exit(1);

}

} else { //listening to any/all incoming address

printf("Now I can accept anything since you do not know IPV4");

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

}

printf("server address is %s", inet\_ntoa(listensock.sin\_addr));

//getting port from service or servent

listensock.sin\_port = htons(9200);

//getting protocol by name

if ((protocol = getprotobyname("tcp")) == 0) {

printf("Can not get protocol");

}

lsocksd = socket(PF\_INET, SOCK\_STREAM, protocol->p\_proto);

//its like connect for client

if (bind(lsocksd, (struct sockaddr \*) &listensock, sizeof listensock) < 0) {

printf("\n Something wrong with bind");

exit(1);

}

if (listen(lsocksd, 5) < 0) {

printf("Something went wrong with lock");

exit(1);

}

return lsocksd;

}

# **Connection-less file servers (UDP)**

## Iterative server

### Readme

To compile:

1. Open terminal at code folder
2. make

OR

1. Open terminal at code folder
2. gcc -o server UDPIterativeServer.c

gcc -o client UDPClient.c

To run:

1. make runs
2. Open another terminal at code folder
3. make runc1
4. make runc2

OR

1. Run server: ./server
2. Open another terminal at code folder
3. Run client: ./client file1.txt
4. Run client: ./client file2.txt

To stop server:

1. Ctrl + C

To clean the files:

1. make clean

OR

1. rm -f server client

### Makefile

all: server client

server:

gcc -o server UDPIterativeServer.c

client:

gcc -o client UDPClient.c

runs:

./server

runc1:

./client file1.txt

runc2:

./client file2.txt

clean:

rm -f server client

### Server Code

(./UDPIterativeServer.c)

#include <sys/types.h>

#include <sys/signal.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <sys/resource.h>

#include <sys/wait.h>

#include <sys/errno.h>

#include <netinet/in.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <netdb.h>

#include <stdarg.h>

#include <arpa/inet.h>

#define QLEN 5 /\* maximum connection queue length \*/

#define BUFSIZE 4096

u\_short portbase = 0; /\* port base, for non-root servers \*/

extern int errno;

typedef unsigned short u\_short;

int getFileSize(FILE \*fp);

int processRequest(int fd, const char \*buf, struct sockaddr\_in sin);

int errexit(const char \*format, ...);

//int passivesock(const char \*service, const char \*transport, int qlen, struct sockaddr\_in \*sin);

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

int passiveUDP(const char \*service);

int main(int argc, char \*argv[]) {

char \*service = "9200"; /\* service name or port number \*/

struct sockaddr\_in fsin; /\* the address of a client \*/

int alen; /\* from-address length \*/

int sock; /\* server socket \*/

char buf[BUFSIZ];

char fileBuf[BUFSIZ];

int n;

char const \*filepath = "files/";

FILE \*file;

sock = passiveUDP(service);

// fputs("a", stdout);

// fflush(stdout);

while (1) {

alen = sizeof(fsin);

if (recvfrom(sock, buf, sizeof(buf), 0, (struct sockaddr \*) &fsin, &alen) < 0) {

errexit("Fail to receive in the server ", strerror(errno), "\n");

}

char fullpath[sizeof(filepath) + sizeof(buf) + 1];

bzero(fullpath, sizeof(fullpath));

strcat(fullpath, filepath);

strcat(fullpath, buf);

file = fopen(fullpath, "r");

int n = 0;

FILE \*ft = fopen(fullpath, "r");

int size = getFileSize(ft);

char sizeSt[128];

sprintf(sizeSt, "%d", size);

fclose(ft);

sendto(sock, sizeSt, strlen(sizeSt), 0, (struct sockaddr \*) &fsin, sizeof(fsin));

if (file) {

while ((n = fread(fileBuf, 1, sizeof(fileBuf), file)) > 0) {

if ((sendto(sock, fileBuf, n, 0, (struct sockaddr \*) &fsin, sizeof(fsin))) < 0) {

errexit("Fail to send file with content: %d", sock);

}

// printf("----- %d", n);

// fputs(fileBuf, stdout);

fflush(stdout);

}

fputs("--------------", stdout);

fflush(stdout);

fclose(file);

} else {

errexit("File %s does not exist", fullpath);

}

}

}

int getFileSize(FILE \*fp) {

fseek(fp, 0L, SEEK\_END);

int res = ftell(fp);

rewind(fp);

return res;

}

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

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(ntohs((u\_short) pse->s\_port) + portbase);

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, SOCK\_DGRAM, 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 passiveUDP(const char \*service) {

return passivesock(service, "udp", QLEN);

}

/\*--------------------------------------------------HELPER--------------------------------------------------\*/

int errexit(const char \*format, ...) {

va\_list args;

va\_start(args, format);

vfprintf(stderr, format, args);

va\_end(args);

exit(1);

}

## Concurrent Multiprocessing server with one process per request

### Readme

To compile:

1. Open terminal at code folder
2. make

OR

1. Open terminal at code folder
2. gcc -o server UDPConProcessServer.c

gcc -o client UDPClient.c

To run:

1. make runs
2. Open another terminal at code folder
3. make runc1
4. make runc2

OR

1. Run server: ./server
2. Open another terminal at code folder
3. Run client: ./client file1.txt
4. Run client: ./client file2.txt

To stop server:

1. Ctrl + C

To clean the files:

1. make clean

OR

1. rm -f server client

### Makefile

all: server client

server:

gcc -o server UDPConProcessServer.c

client:

gcc -o client UDPClient.c

runs:

./server

runc1:

./client file1.txt

runc2:

./client file2.txt

clean:

rm -f server client

### Server Code

(./UDPConProcessServer.c)

#include <sys/types.h>

#include <sys/signal.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <sys/resource.h>

#include <sys/wait.h>

#include <sys/errno.h>

#include <netinet/in.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <netdb.h>

#include <stdarg.h>

#include <arpa/inet.h>

#define QLEN 5 /\* maximum connection queue length \*/

#define BUFSIZE 4096

u\_short portbase = 0; /\* port base, for non-root servers \*/

void reaper(int);

int getFileSize(FILE \*fp);

int processRequest(int fd, const char \*buf, struct sockaddr\_in sin);

int errexit(const char \*format, ...);

int passivesock(const char \*service, const char \*transport, int qlen, struct sockaddr\_in \*sin);

int main(int argc, char \*argv[]) {

char \*service = "9200"; /\* service name or port number \*/

struct sockaddr\_in fsin; /\* the address of a client \*/

struct sockaddr\_in ser; /\* the address of a client \*/

socklen\_t addrlen = sizeof(ser); /\* length of addresses \*/

int msock; /\* master server socket \*/

char buf[BUFSIZ];

msock = passivesock(service, "udp", QLEN, &fsin);

(void) signal(SIGCHLD, reaper);

while (1) {

if (recvfrom(msock, buf, sizeof(buf), 0, (struct sockaddr \*) &ser, &addrlen) < 0) {

errexit("Fail to receive in the server ");

}

switch (fork()) {

case 0: /\* child \*/

exit(processRequest(msock, buf, ser));

default: /\* parent \*/

break;

case -1:

errexit("fork: %s\n", strerror(errno));

}

}

}

int processRequest(int fd, const char \*buf, const struct sockaddr\_in sin) {

char fileBuf[BUFSIZ];

int n;

int count = 0;

char const \*filepath = "files/";

FILE \*file;

char fullpath[sizeof(filepath) + sizeof(buf) + 1];

strcat(fullpath, filepath);

strcat(fullpath, buf);

file = fopen(fullpath, "r");

n = 0;

FILE \*ft = fopen(fullpath, "r");

int size = getFileSize(ft);

char sizeSt[128];

sprintf(sizeSt, "%d", size);

fclose(ft);

sendto(fd, sizeSt, strlen(sizeSt), 0, (struct sockaddr \*) &sin, sizeof(sin));

if (file) {

while ((n = fread(fileBuf, 1, sizeof(fileBuf), file)) > 0) {

// fputs(fileBuf, stdout);

if (sendto(fd, fileBuf, n, 0, (struct sockaddr \*) &sin, sizeof(sin)) < 0) {

errexit("Fail to send file with content: %d", fd);

}

}

fclose(file);

} else {

errexit("File %s does not exist", fullpath);

}

return 0;

}

int passivesock(const char \*service, const char \*transport, int qlen, struct sockaddr\_in \*fsin) {

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(ntohs((u\_short) pse->s\_port) + portbase);

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

errexit("can't get \"%s\" service entry\n", service);

if ((ppe = getprotobyname(transport)) == 0)

errexit("can't get \"%s\" protocol entry\n", transport);

s = socket(PF\_INET, SOCK\_DGRAM, ppe->p\_proto);

if (s < 0)

errexit("can't create socket: %s\n", strerror(errno));

if (bind(s, (struct sockaddr \*) &sin, sizeof(sin)) < 0)

errexit("can't bind to %s port: %s\n", service,

strerror(errno));

\*fsin = sin;

return s;

}

/\*--------------------------------------------------HELPER--------------------------------------------------\*/

void reaper(int sig) {

int status;

while (wait3(&status, WNOHANG, (struct rusage \*) 0) >= 0)

/\* empty \*/;

}

int errexit(const char \*format, ...) {

va\_list args;

va\_start(args, format);

vfprintf(stderr, format, args);

va\_end(args);

exit(1);

}

int getFileSize(FILE \*fp) {

fseek(fp, 0L, SEEK\_END);

int res = ftell(fp);

rewind(fp);

return res;

}

## Concurrent Multithreading server with one thread per request

### Readme

To compile:

1. Open terminal at code folder
2. make

OR

1. Open terminal at code folder
2. gcc -pthread -o server UDPConThreadServer.c

gcc -o client UDPClient.c

To run:

1. make runs
2. Open another terminal at code folder
3. make runc1
4. make runc2

OR

1. Run server: ./server
2. Open another terminal at code folder
3. Run client: ./client file1.txt
4. Run client: ./client file2.txt

To stop server:

1. Ctrl + C

To clean the files:

1. make clean

OR

1. rm -f server client

### Makefile

all: server client

server:

gcc -pthread -o server UDPConThreadServer.c

client:

gcc -o client UDPClient.c

runs:

./server

runc1:

./client file1.txt

runc2:

./client file2.txt

clean:

rm -f server client

### Server Code

(./UDPConThreadServer.c)

//file transfer server for testing FTP hope it will be easy :-)

//This is concurrent multithreading server

/\*\*

author : Omkar M. Rege

This is a simple TCP server program that accepts 1 parameter,

1. IP on which it should run

2. The server return the file the client has specified

4. For example: USAGE : ./udpserver.o localhost

\*/

//Common header

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdarg.h>

#include <string.h>

#include <strings.h>

#include <errno.h>

#include <stdbool.h>

//Network specific

#include <sys/socket.h>

#include <sys/types.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

//thread library

#include<pthread.h>

int definelisteningsocket(char \*ip);

void \*filehandler(void \*arg);

int getFileSize(FILE \*fp);

// A struct to pass arguements to the Pthread

struct ThreadPara {

int mainSock;

struct sockaddr\_in \*clientSockPtr;

char \*filename;

};

int main(int argc, char \*argv[]) {

int lsd, clientsocksd, clientsocklength;

char buffer[1024];

struct sockaddr\_in clientsock;

lsd = definelisteningsocket("localhost");

pthread\_t clientthread;

fflush(stdout);

printf("\nreturned from definelisteningsocket\n");

while (1) {

clientsocklength = sizeof clientsock;

if (recvfrom(lsd, buffer, sizeof buffer, 0, (struct sockaddr \*) &clientsock, &clientsocklength) < 0) {

printf("Receive failed");

fflush(stdout);

exit(1);

}

struct ThreadPara myPara;

myPara.mainSock = lsd;

myPara.clientSockPtr = &clientsock;

myPara.filename = buffer;

if (pthread\_create(&clientthread, NULL, filehandler, (void \*) &myPara) != 0) {

printf("\nerror is spawning thread\n");

}

printf("\nA thread has been spawned!!!\n");

fflush(stdout);

}

}//end of main

void \*filehandler(void \*arg) {

fflush(stdout);

printf("\nThis is thread");

const char \*const DIR\_PATH = "./files/";

struct ThreadPara param = \*((struct ThreadPara \*) arg);

fflush(stdout);

printf("\nThis is thread");

char \*message = param.filename;

printf("\nFile name client sent is %s \n", message);

fflush(stdout);

int mainSock = param.mainSock;

struct sockaddr\_in \*clientSocket = param.clientSockPtr;

//send filesize to client

char filename[128];

memset(filename, 0, 128);

strcpy(filename, DIR\_PATH);

strcat(filename, message);

FILE \*ft = fopen(filename, "r");

int size = getFileSize(ft);

char sizeSt[128];

sprintf(sizeSt, "%d", size);

// printf("\n Size is %d", size);

fclose(ft);

sendto(mainSock, sizeSt, strlen(sizeSt), 0, (struct sockaddr \*) clientSocket, sizeof \*(clientSocket));

int bytesread = 0, cnt = 0;

char buffer[128];

char ch;

//writting part

printf("\nFile name client sent is %s \n", message);

fflush(stdout);

//resetting buffer and counter

memset(buffer, 0, sizeof buffer);

cnt = 0;

FILE \*fp = fopen(filename, "r");

if (fp == NULL) {

sendto(mainSock, "FILE DOES NOT EXIST", 21, 0, (struct sockaddr \*) clientSocket, sizeof \*(clientSocket));

exit(0);

}

int totalwritten = 0, bytesWritten = 0;

while (1) {

ch = fgetc(fp);

if (feof(fp)) {

fflush(stdout);

printf("cnt is %d", cnt);

fflush(stdout);

break;

}

buffer[cnt] = ch;

cnt++;

printf("\n%c %d", ch, cnt);

fflush(stdout);

if (cnt == 128) {

while (cnt != 0) {

bytesWritten = sendto(mainSock, buffer + totalwritten, 128 - totalwritten, 0,

(struct sockaddr \*) clientSocket, sizeof \*(clientSocket));

totalwritten += bytesWritten;

cnt -= bytesWritten;

}

cnt = 0;

bytesWritten = 0;

totalwritten = 0;

memset(buffer, 0, sizeof buffer); //reset the buffer

}

}

// printf("Reached Here"); fflush(stdout);

bytesWritten = 0;

totalwritten = 0;

while (cnt > 0) {

printf("Writting to the client %s:", buffer);

fflush(stdout);

bytesWritten = sendto(mainSock, buffer + totalwritten, cnt, 0, (struct sockaddr \*) clientSocket,

sizeof \*(clientSocket));

totalwritten += bytesWritten;

cnt -= bytesWritten;

}

fclose(fp);

// free(message);

// printf("Thread has closed connected socket");

// fflush(stdout);

}

int getFileSize(FILE \*fp) {

fseek(fp, 0L, SEEK\_END);

int res = ftell(fp);

rewind(fp);

return res;

}

int definelisteningsocket(char \*ip) {

int lsocksd;

struct sockaddr\_in listensock;

struct servent \*service;

struct protoent \*protocol;

memset(&listensock, 0, sizeof listensock);

listensock.sin\_family = AF\_INET;

//first we will convert ip from ascii string to network address

if (inet\_aton(ip, &listensock.sin\_addr) != 0) {

printf("Your IP adrress is weird %s", ip);

} else if (strcasecmp("localhost", ip) == 0) {

printf("You have a Localhost");

if (inet\_aton("127.0.0.1", &listensock.sin\_addr) == 0) {

printf("It's better to leave this world!");

exit(1);

}

} else { //listening to any/all incoming address

printf("Now I can accept anything since you do not know IPV4");

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

}

printf("server address is %s", inet\_ntoa(listensock.sin\_addr));

//getting port from service or servent

listensock.sin\_port = htons(9200);

//getting protocol by name

if ((protocol = getprotobyname("udp")) == 0) {

printf("Can not get protocol");

}

lsocksd = socket(PF\_INET, SOCK\_DGRAM, protocol->p\_proto);

//its like connect for client

if (bind(lsocksd, (struct sockaddr \*) &listensock, sizeof listensock) < 0) {

printf("\n Something wrong with bind");

exit(1);

}

return lsocksd;

}

## Concurrent Pre-forked Multiprocessing server

### Readme

To compile:

1. Open terminal at code folder
2. make

OR

1. Open terminal at code folder
2. gcc -o server UDPConPreForkedServer.c

gcc -o client UDPClient.c

To run:

1. make runs
2. Open another terminal at code folder
3. make runc1
4. make runc2

OR

1. Run server: ./server
2. Open another terminal at code folder
3. Run client: ./client file1.txt
4. Run client: ./client file2.txt

To stop server:

1. Ctrl + C

To clean the files:

1. make clean

OR

1. rm -f server client

### Makefile

all: server client

server:

gcc -o server UDPConPreForkedServer.c

client:

gcc -o client UDPClient.c

runs:

./server

runc1:

./client file1.txt

runc2:

./client file2.txt

clean:

rm -f server client

### Server Code

(./UDPConPreForkedServer.c)

#include <sys/types.h>

#include <sys/signal.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <sys/resource.h>

#include <sys/wait.h>

#include <sys/errno.h>

#include <netinet/in.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <netdb.h>

#include <stdarg.h>

#include <arpa/inet.h>

#define QLEN 5 /\* maximum connection queue length \*/

#define BUFSIZE 4096

#define NUM\_OF\_CHILDREN 5

u\_short portbase = 0; /\* port base, for non-root servers \*/

void reaper(int);

int getFileSize(FILE \*fp);

int processRequest(int fd, const char \*buf, struct sockaddr\_in sin);

int errexit(const char \*format, ...);

int passivesock(const char \*service, const char \*transport, int qlen, struct sockaddr\_in \*sin);

int main(int argc, char \*argv[]) {

char \*service = "9200"; /\* service name or port number \*/

struct sockaddr\_in fsin; /\* the address of a client \*/

struct sockaddr\_in ser; /\* the address of a client \*/

socklen\_t addrlen = sizeof(ser); /\* length of addresses \*/

int msock; /\* master server socket \*/

char buf[BUFSIZ];

msock = passivesock(service, "udp", QLEN, &fsin);

int count = 0;

for (count = 0; count < NUM\_OF\_CHILDREN - 1; count++) {

if (fork() == 0) {

// in the child

while (1) {

if (recvfrom(msock, buf, sizeof(buf), 0, (struct sockaddr \*) &ser, &addrlen) < 0) {

errexit("Fail to receive in the server");

continue;

}

printf("pid: %d \n", getpid());

fflush(stdout);

processRequest(msock, buf, ser);

}

}

}

(void) signal(SIGCHLD, reaper);

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

close(msock);

}

int processRequest(int fd, const char \*buf, const struct sockaddr\_in sin) {

char fileBuf[BUFSIZ];

int n;

int count = 0;

char const \*filepath = "files/";

FILE \*file;

char fullpath[sizeof(filepath) + sizeof(buf) + 1];

bzero(fullpath, sizeof(fullpath));

strcat(fullpath, filepath);

strcat(fullpath, buf);

file = fopen(fullpath, "r");

n = 0;

FILE \*ft = fopen(fullpath, "r");

int size = getFileSize(ft);

char sizeSt[128];

sprintf(sizeSt, "%d", size);

fclose(ft);

sendto(fd, sizeSt, strlen(sizeSt), 0, (struct sockaddr \*) &sin, sizeof(sin));

if (file) {

while ((n = fread(fileBuf, 1, sizeof(fileBuf), file)) > 0) {

if (sendto(fd, fileBuf, n, 0, (struct sockaddr \*) &sin, sizeof(sin)) < 0) {

errexit("Fail to send file with content: %d", fd);

}

}

fclose(file);

} else {

errexit("File %s does not exist", fullpath);

}

return 0;

}

int passivesock(const char \*service, const char \*transport, int qlen, struct sockaddr\_in \*fsin) {

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(ntohs((u\_short) pse->s\_port) + portbase);

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

errexit("can't get \"%s\" service entry\n", service);

if ((ppe = getprotobyname(transport)) == 0)

errexit("can't get \"%s\" protocol entry\n", transport);

s = socket(PF\_INET, SOCK\_DGRAM, ppe->p\_proto);

if (s < 0)

errexit("can't create socket: %s\n", strerror(errno));

if (bind(s, (struct sockaddr \*) &sin, sizeof(sin)) < 0)

errexit("can't bind to %s port: %s\n", service,

strerror(errno));

\*fsin = sin;

return s;

}

/\*--------------------------------------------------HELPER--------------------------------------------------\*/

void reaper(int sig) {

int status;

while (wait3(&status, WNOHANG, (struct rusage \*) 0) >= 0)

/\* empty \*/;

}

int errexit(const char \*format, ...) {

va\_list args;

va\_start(args, format);

vfprintf(stderr, format, args);

va\_end(args);

}

int getFileSize(FILE \*fp) {

fseek(fp, 0L, SEEK\_END);

int res = ftell(fp);

rewind(fp);

return res;

}

## Concurrent Pre-threaded Multithreading server

### Readme

To compile:

1. Open terminal at code folder
2. make

OR

1. Open terminal at code folder
2. gcc -pthread -o server UDPConPreThreadedServer.c

gcc -o client UDPClient.c

To run:

1. make runs
2. Open another terminal at code folder
3. make runc1
4. make runc2

OR

1. Run server: ./server
2. Open another terminal at code folder
3. Run client: ./client file1.txt
4. Run client: ./client file2.txt

To stop server:

1. Ctrl + C

To clean the files:

1. make clean

OR

1. rm -f server client

### Makefile

all: server client

server:

gcc -pthread -o server UDPConPreThreadedServer.c

client:

gcc -o client UDPClient.c

runs:

./server

runc1:

./client file1.txt

runc2:

./client file2.txt

clean:

rm -f server client

### Server Code

(./UDPConPreThreadedServer.c)

//file transfer server for testing FTP hope it will be easy :-)

//This is concurrent multithreading server

/\*\*

author : Omkar M. Rege

This is a simple TCP server program that accepts 1 parameter,

1. IP on which it should run

2. The server return the file the client has specified

4. For example: USAGE : ./udpserver.o localhost

\*/

//Common header

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdarg.h>

#include <string.h>

#include <strings.h>

#include <errno.h>

#include <stdbool.h>

//Network specific

#include <sys/socket.h>

#include <sys/types.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

//thread library

#include<pthread.h>

#define THREADS 5

int definelisteningsocket(char \*ip);

void \*filehandler(void \*arg);

int getFileSize(FILE \*fp);

struct ThreadPara {

int mainSock;

struct sockaddr\_in \*clientSockPtr;

char \*filename;

};

int main(int argc, char \*argv[]) {

int lsd;

lsd = definelisteningsocket("localhost");

fflush(stdout);

printf("\nreturned from definelisteningsocket\n");

int i = 0;

pthread\_t \*ptr[THREADS];

for (i = 0; i < THREADS; i++) {

pthread\_t clientthread;

int \*newListenSocket = malloc(sizeof(int));

\*newListenSocket = lsd;

if (pthread\_create(&clientthread, NULL, filehandler, (void \*) newListenSocket) != 0) {

printf("\nerror is spawning thread\n");

}

ptr[i] = &clientthread;

printf("\n A thread has been spawned!!! \n");

fflush(stdout);

}//end of for loop

// for (i = 0; i < THREADS; i++) {

// if (pthread\_join((pthread\_t) ptr[i], NULL) != 0) {

// printf("JOIN HAS FAILED");

// fflush(stdout);

// }

// }

while (1) {}

}//end of main

void \*filehandler(void \*arg) {

fflush(stdout);

printf("This is thread");

const char \*const DIR\_PATH = "./files/";

int mainSock = \*((int \*) arg);

fflush(stdout);

printf("This is thread");

char message[128];

struct sockaddr\_in clientSocket;

int clientsocklength = sizeof clientSocket;

memset(message, 0, sizeof message);

if (recvfrom(mainSock, message, sizeof message, 0, (struct sockaddr \*) &clientSocket, &clientsocklength) < 0) {

printf("Receive failed");

fflush(stdout);

exit(1);

}

printf("File name client sent is %s \n", message);

fflush(stdout);

//send filesize to client

char filename[128];

memset(filename, 0, 128);

strcpy(filename, DIR\_PATH);

strcat(filename, message);

FILE \*ft = fopen(filename, "r");

int size = getFileSize(ft);

printf("\n Size is %d", size);

char sizeSt[128];

sprintf(sizeSt, "%d", size);

fclose(ft);

sendto(mainSock, sizeSt, strlen(sizeSt), 0, (struct sockaddr \*) &clientSocket, sizeof(clientSocket));

int bytesread = 0, cnt = 0;

char buffer[128];

char ch;

//writting part

printf("File name client sent is %s \n", message);

fflush(stdout);

//resetting buffer and counter

memset(buffer, 0, sizeof buffer);

cnt = 0;

FILE \*fp = fopen(filename, "r");

if (fp == NULL) {

sendto(mainSock, "FILE DOES NOT EXIST", 21, 0, (struct sockaddr \*) &clientSocket, sizeof(clientSocket));

exit(0);

}

int totalwritten = 0, bytesWritten = 0;

while (1) {

ch = fgetc(fp);

if (feof(fp)) {

fflush(stdout);

printf("cnt is %d", cnt);

fflush(stdout);

break;

}

buffer[cnt] = ch;

cnt++;

// printf("\n%c %d",ch,cnt);

// fflush(stdout);

int j = 1;

if (cnt == 128) {

while (cnt != 0) {

bytesWritten = sendto(mainSock, buffer + totalwritten, 128 - totalwritten, 0,

(struct sockaddr \*) &clientSocket, sizeof(clientSocket));

totalwritten += bytesWritten;

cnt -= bytesWritten;

printf("-------------> %d", bytesWritten);

}

cnt = 0;

bytesWritten = 0;

totalwritten = 0;

memset(buffer, 0, sizeof buffer); //reset the buffer

}

}

bytesWritten = 0;

totalwritten = 0;

printf("------------->Reached Here and cnt is %d", cnt);

fflush(stdout);

while (cnt > 0) {

printf("Writting to the client %s:", buffer);

fflush(stdout);

bytesWritten = sendto(mainSock, buffer + totalwritten, cnt, 0, (struct sockaddr \*) &clientSocket,

sizeof(clientSocket));

totalwritten += bytesWritten;

cnt -= bytesWritten;

}

// sendto(mainSock,"EOF",4,0,(struct sockaddr \*)&clientSocket,sizeof(clientSocket));

fclose(fp);

// free(message);

// printf("Thread has closed connected socket");

// fflush(stdout);

}

int getFileSize(FILE \*fp) {

fseek(fp, 0L, SEEK\_END);

int res = ftell(fp);

rewind(fp);

return res;

}

int definelisteningsocket(char \*ip) {

int lsocksd;

struct sockaddr\_in listensock;

struct servent \*service;

struct protoent \*protocol;

memset(&listensock, 0, sizeof listensock);

listensock.sin\_family = AF\_INET;

//first we will convert ip from ascii string to network address

if (inet\_aton(ip, &listensock.sin\_addr) != 0) {

printf("Your IP adrress is weird %s", ip);

} else if (strcasecmp("localhost", ip) == 0) {

printf("You have a Localhost");

if (inet\_aton("127.0.0.1", &listensock.sin\_addr) == 0) {

printf("It's better to leave this world!");

exit(1);

}

} else { //listening to any/all incoming address

printf("Now I can accept anything since you do not know IPV4");

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

}

printf("server address is %s", inet\_ntoa(listensock.sin\_addr));

//getting port from service or servent

listensock.sin\_port = htons(9200);

//getting protocol by name

if ((protocol = getprotobyname("udp")) == 0) {

printf("Can not get protocol");

}

lsocksd = socket(PF\_INET, SOCK\_DGRAM, protocol->p\_proto);

//its like connect for client

if (bind(lsocksd, (struct sockaddr \*) &listensock, sizeof listensock) < 0) {

printf("\n Something wrong with bind");

exit(1);

}

return lsocksd;

}