Team Members: Chris Baldwin & Kaushal Prudhvi
Date: 9/20/2021
Course: ECEN 602 Fall 2021
Assignment: MP1
Professor: Dr. Narasimha Annapareddy
TA: Rishabh Singla

Github: https://github.com/chrisbaldwin2/EchoServer

The project zip contains

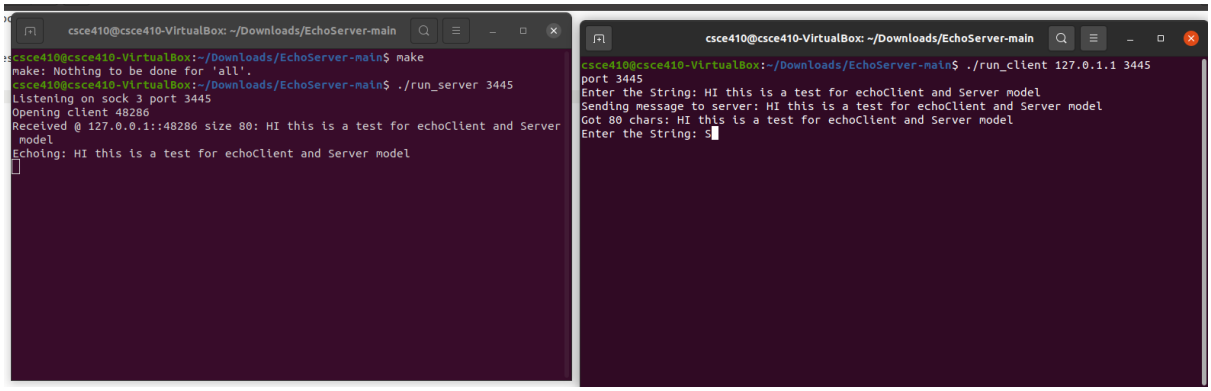
1) The Make File
2) run_server.cpp
3) run_client.cpp
4) README
5) Mp1.h

The Steps for executing the files are :

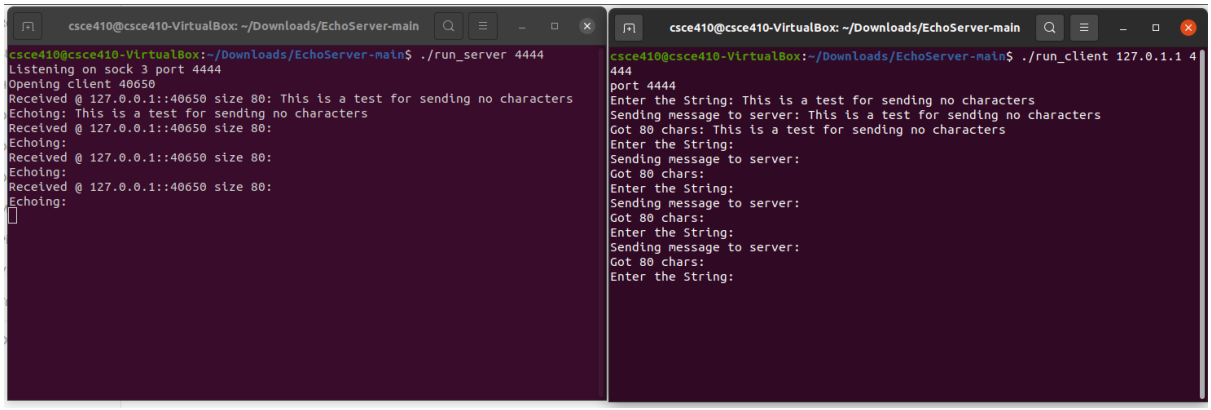
1) Open 2 terminals and change the directory where the files are located
2) Type make and hit enter
3) On first terminal enter ./run_server PORT_NUMBER
4) On Second terminal enter ./run_client <HOST_IP> PORT_NUMBER
5) The code is written in such a way that the port numbers have to be the same on both the sides for a good command line outputs
6) Test the echo client and server models by the following cases
7) All the test cases have been implemented and documented
8) The code is also mentioned after the results

Test cases and Results:

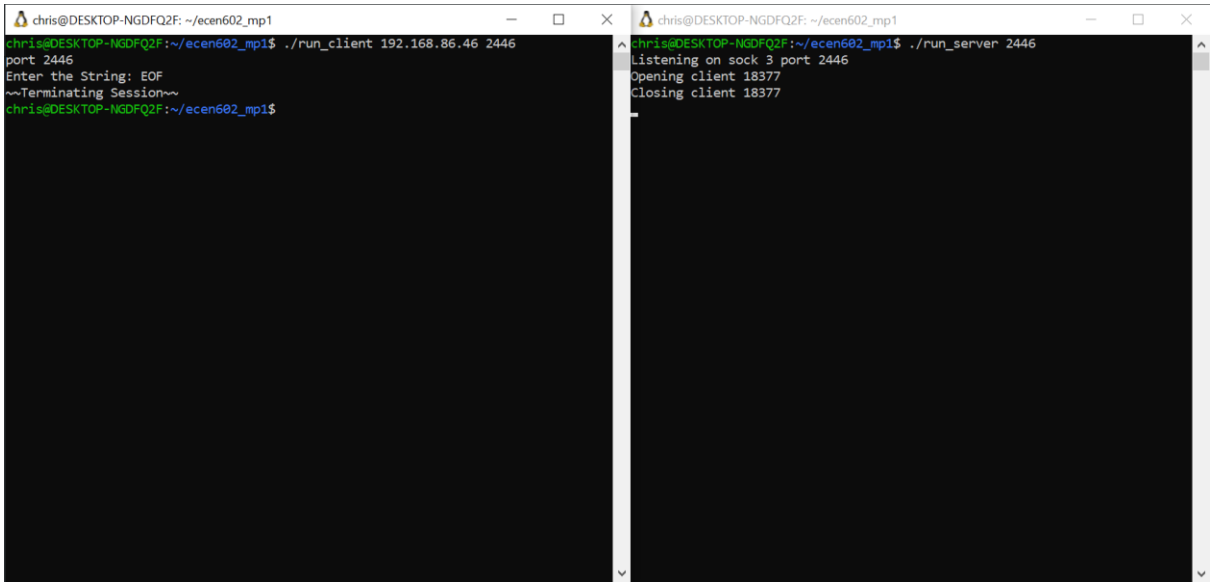
1) Line of text terminated by a newline



2) Line with no characters

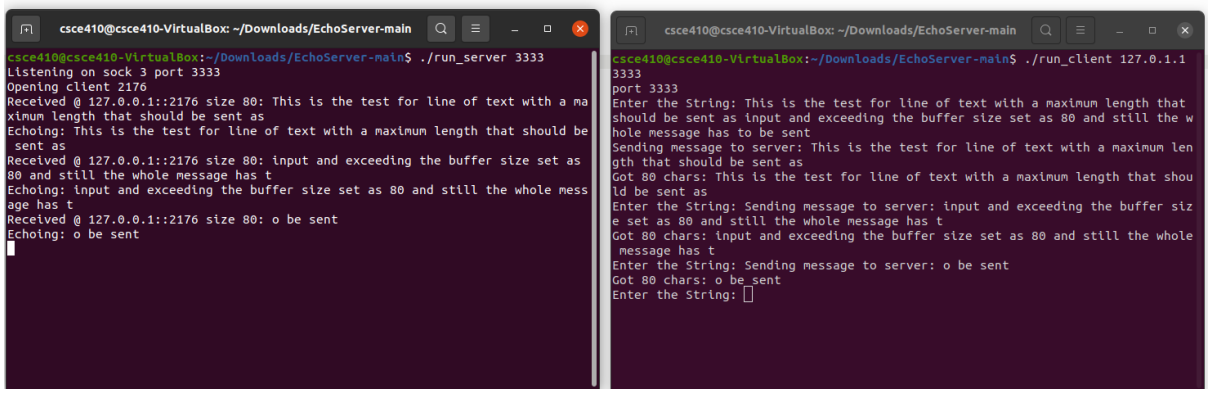


3)When the client sends EOF, the client & server closes the connection

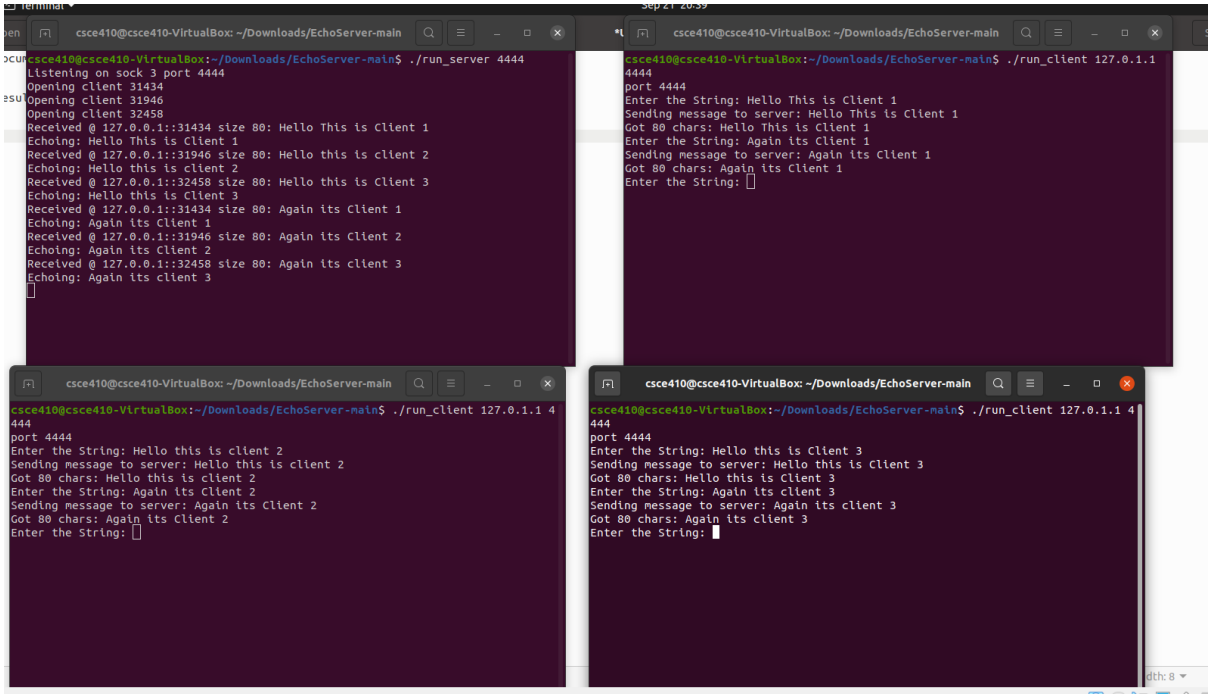


4) Line of text the maximum line length without a newline,

Here we have tried to send the maximum length without a newline exceeding the buffer and the server echoed the full message successfully



5) Three simultaneous clients connecting to the server and sending messages and receiving echoes



6)When the clients terminated abruptly after echoing messages when terminal is closed.

```
csce410@csce410-VirtualBox:~/Downloads/EchoServer-main$ ./run_server 4444
Listening on sock 3 port 4444
Opening client 31434
Opening client 31946
Opening client 32458
Received @ 127.0.0.1::31434 size 80: Hello This is Client 1
Echoing: Hello This is Client 1
Received @ 127.0.0.1::31946 size 80: Hello this is client 2
Echoing: Hello this is client 2
Received @ 127.0.0.1::32458 size 80: Hello this is Client 3
Echoing: Hello this is Client 3
Received @ 127.0.0.1::31434 size 80: Again its Client 1
Echoing: Again its Client 1
Received @ 127.0.0.1::31946 size 80: Again its Client 2
Echoing: Again its Client 2
Received @ 127.0.0.1::32458 size 80: Again its client 3
Echoing: Again its client 3
Received @ 127.0.0.1::31434 size 80: exit
Echoing: exit
Closing client 31946
Closing client 32458
Closing client 31434
```

(7) Client terminated after entering text. The test is done by pressing ctrl -c on client terminal.

```
csce410@csce410-VirtualBox:~/Downloads/EchoServer-main$ ./run_server 3333
Listening on sock 3 port 3333
Opening client 5248
Received @ 127.0.0.1::5248 size 80: This is the test for client termination afte
r entering a text
Echoing: This is the test for client termination after entering a text
Closing client 5248
```

```
csce410@csce410-VirtualBox:~/Downloads/EchoServer-main$ ./run_client 127.0.1.1 3
333
port 3333
Enter the String: This is the test for client termination after entering a text
Sending message to server: This is the test for client termination after enterin
g a text
Got 80 chars: This is the test for client termination after entering a text
Enter the String: ^C
csce410@csce410-VirtualBox:~/Downloads/EchoServer-main$
```

Makefile

```
SERVERFILE=run_server

CLIENTFILE=run_client
```

```
INSTALLDIR=build
OUTPUTFILE=$(SERVERFILE) $(CLIENTFILE)
# look for .cpp & .h files in ./src
vpath %.cpp ./src
vpath %.h ./src

.PHONY: all
all: $(OUTPUTFILE)



%: %.cpp
    g++ -o $@ $<



.PHONY: install
install:
    mkdir -p $(INSTALLDIR)
    cp -p $(SERVERFILE) $(INSTALLDIR)
    cp -p $(CLIENTFILE) $(INSTALLDIR)

.PHONY: clean
clean:
    rm -f $(INSTALLDIR)/$(SERVERFILE) $(INSTALLDIR)/$(CLIENTFILE)
    rm -f $(OUTPUTFILE)
```

File run_client.cpp

```
/* run_client.cpp
 *
 * Original Author: Chris Baldwin
 * Partner:         Kaushal Prudhvi
 * Date:            9/20/2021
 * Course:          ECEN 602 Fall 2021
 * Assignmnet:      MP1
 * Professor:       Dr. Narasimha Annapareddy
 * TA:              Rishabh Singla
```

```
 *
 * History Table
 * ==============================================================
 * Date        :: Author         :: Change
 * -----------++---------------++-----------------------------
 * 9/20/2021  :: Chris Baldwin :: Created run_client.cpp file
 * -----------++---------------++-----------------------------
 * 9/21/2021  :: Chris Baldwin :: Added errno handling
 * -----------++---------------++-----------------------------
 * 9/21/2021  :: Chris Baldwin :: Added function header
 *            ::               :: comments and history table
 * -----------++---------------++-----------------------------
 * 9/21/2021  :: Chris Baldwin :: Added better newline handling
 *            ::               :: and buffer underflow in r/w
 * -----------++---------------++-----------------------------
 * 9/21/2021  :: Chris Baldwin :: Added EOF handling
 * -----------++---------------++-----------------------------
 */

#include<stdio.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<string.h>
#include<strings.h>
#include<errno.h>
#include<unistd.h>
#include<arpa/inet.h>
#include"mp1.h"

/* error
 *
 * Prints the reason for the error and exits the program with code -1
 *
```

```
 * @param err The string describing why the error has occured
 * @return none
 */
void error(const char *err)
{
    printf("%s", err);
    exit(MP1::ERROR);
}

/* bind_socket
 *
 * Acquires a socket
 *
 * @param none
 * @return sfd The socket file descriptor
 */
int get_socket()
{
    int sfd;
    // Attempt to acquire a TCP IPv4 socket
    sfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sfd < 0) error("Error acquiring socket\n");
    // printf("socket descriptor: %d\n", sfd);
    return sfd;
}

/* connect_to_server
 *
 * Connects to a server specified in argv where argv[1] is the ip
 * and argv[2] is the port number
 *
 * @param sfd The open socket file descriptor
 * @param argv The arguments passed in from the command line
 * @return none
 */
```

```c
void connect_to_server(int sfd, char *argv[])
{
    struct sockaddr_in serv_addr;
    char *str[20];
    int port, err;
    // Zero out the memory @ serv_addr
    bzero((char *) &serv_addr, sizeof(serv_addr));
    // Set values for sockaddr_in struct
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
    port = strtol(argv[2], str, 10);
    printf("port %d\n", port);
    serv_addr.sin_port = htons((uint16_t) port);
    // Try to connect to the ip and port number specified
    err = connect(sfd, (struct sockaddr *) &serv_addr,
sizeof(serv_addr));
    if(err < 0) error("Error connecting to server\n");
}

/* readline
 *
 * Reads from stdin and writes it to buf
 *
 * @param[out] buf The buffer to write to
 * @return none
 */
int readline(char *buf)
{
    printf("Enter the String: ");
    // Read from stdin at most the size of the buffer
    if(!fgets(buf, MP1::buf_size, stdin))
    {
        printf("EOF\n");
        return MP1::ERROR;
    }
```

```
    if(!strstr(buf, "\n")) printf("\n");
    return MP1::GOOD;
}

/* writen
 *
 * Writes the buffer passed in to the connected server
 *
 * @param sfd The open socket file descriptor
 * @param buf The buffer to be written to the server
 * @return none
 */
void writen(int sfd, char *buf, int index)
{
    ssize_t size;
    printf("Sending message to server: %s", buf);
    if(!strstr(buf, "\n")) printf("\n");
    // Attempt to send the packet to the server & retry on EINTR
    write_l:
      size = write(sfd, buf + index, MP1::buf_size - index);
      if(size < 0 && errno == EINTR) goto write_l;
    if(size < 0) error("Error writing to socket\n");
    if(size + index < MP1::buf_size) writen(sfd, buf, index + size);
}

int readn(int sfd, char *buf, int index)
{
    int size;
    read_l:
      size = read(sfd, buf + index, MP1::buf_size - index);
      if(size < 0 && errno == EINTR) goto read_l;
    if(size < 0) error("Error reading from socket\n");
    if(size + index < MP1::buf_size) readn(sfd, buf, index + size);
    return size;
}
```

```c
/* listen_for_resp
 *
 * Waits for the echo from the server and prints it to
 * stdout
 *
 * @param[in]  sfd The open socket file descriptor
 * @param[out] buf The buffer to store the echo resp in
 * @return none
 */
void listen_for_resp(int sfd, char *buf)
{
    ssize_t size;
    bzero(buf,MP1::buf_size);
    // Attempt to receive the packet from the server & retry on EINTR
    size = readn(sfd, buf, 0);
    printf("Got %d chars: %s", (int) size, buf);
    if(!strstr(buf, "\n")) printf("\n");
}

int main(int argc, char *argv[])
{
    char buf [MP1::buf_size];
    int sock_fd;
    if(argc != 3) error("Call must have 2 arguments: IPv4_ADDRESS
PORT_NUM\n");
    sock_fd = get_socket();
    connect_to_server(sock_fd, argv);
    while(1)
    {
        if(readline(buf) == MP1::ERROR) break;
        writen(sock_fd, buf, 0);
        listen_for_resp(sock_fd, buf);
    }
    // Close the socket ( sending EOF in the proccess )
```

```
    printf("~~Terminating Session~~\n");
    close(sock_fd);
    return MP1::GOOD;
}
```

FILE run_server.cpp

```
/* run_server.cpp
 *
 * Original Author: Chris Baldwin
 * Partner:         Kaushal Prudhvi
 * Date:            9/20/2021
 * Course:          ECEN 602 Fall 2021
 * Assignmnet:      MP1
 * Professor:       Dr. Narasimha Annapareddy
 * TA:              Rishabh Singla
 *
 * History Table
 * ==============================================================
 * Date       :: Author        :: Change
 * -----------++---------------++-----------------------------
 * 9/20/2021  :: Chris Baldwin :: Created run_server.cpp file
 * -----------++---------------++-----------------------------
 * 9/21/2021  :: Chris Baldwin :: Added errno handling
 * -----------++---------------++-----------------------------
 * 9/21/2021  :: Chris Baldwin :: Added function header
 *            ::               :: comments and history table
 * -----------++---------------++-----------------------------
 * 9/21/2021  :: Chris Baldwin :: Added better newline handling
 *            ::               :: and buffer underflow in r/w
 * -----------++---------------++-----------------------------
 * 9/21/2021  :: Chris Baldwin :: Added EOF handling
 * -----------++---------------++-----------------------------
```

```
 */

#include<stdio.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<string.h>
#include<strings.h>
#include<errno.h>
#include<unistd.h>
#include<arpa/inet.h>
#include"mp1.h"
#include<signal.h>

/* error
 *
 * Prints the reason for the error and exits the program with code -1
 *
 * @param err The string describing why the error has occured
 * @return none
 */
void error(const char *err)
{
    printf("%s", err);
    exit(MP1::ERROR);
}

/* bind_socket
 *
 * Acquires a socket, binds it, and begins listening
 *
 * @param address The struct containing the addresses to accept
(INADDR_ANY),
 *                port to bind, and the family (IPv4)
```

```
 * @return sfd The socket file descriptor
 */
int bind_socket(struct sockaddr_in *address)
{
    int sfd, err;
    sfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sfd < 0) error("Error acquiring socket\n");
    // printf("socket descriptor: %d\n", sfd);
    err = bind(sfd, (struct sockaddr *) address, sizeof(struct
sockaddr));
    if(sfd < 0) error("Error binding socket\n");
    // printf("bind: %d\n", err);
    err = listen(sfd, MP1::list_queue_size);
    if(err < 0) error("Error listening on socket\n");
    // printf("listen: %d\n", err);
    return sfd;
}

/* read_socket
 *
 * Reads from the socket file descriptor to th buffer. If
 * the write returns less than the size of the buffer, this
 * wrapper will attempt to write the rest of the buffer.
 *
 * @param sfd The open socket file descriptor
 * @param buf The buffer to be written to the socket
 * @param index The index of the buffer to start writting
 * @return size The size of the buffer written to the sfd
 */
int read_socket(int sfd, char *buf, int index)
{
    int size;
    read_l:
      size = read(sfd, buf + index, MP1::buf_size - index);
      if(size < 0 && errno == EINTR) goto read_l;
```

```
    if(size < 0) error("Error receiving packet\n");

    if(size == 0) return 0;

    if(size + index < MP1::buf_size) read_socket(sfd, buf, size +
index);

    return size + index;

}


/* write_socket

 *

 * Writes the buffer to the socket file descriptor. If the write

 * returns less than the size of the buffer, this wrapper will

 * attempt to write the rest of the buffer.

 *

 * @param sfd The open socket file descriptor

 * @param buf The buffer to be written to the socket

 * @param index The index of the buffer to start writting

 * @return size The size of the buffer written to the sfd

 */

int write_socket(int sfd, char *buf, int index)

{

    int size;

    write_l:

      size = write(sfd, buf + index, MP1::buf_size - index);

      if(size < 0 && errno == EINTR) goto write_l;

    if(size < 0) error("Error sending echo packet\n");

    if(size + index < MP1::buf_size) write_socket(sfd, buf, index +
size);

    return size + index;

}


/* listen_on_socket

 *

 * Listens for a new connenction, accepts it, forks, and echos any
packets which are sent.

 * On sending a blank buffer (EOF), the connection is closed and the
child process ended.

 *
```

```
 * @param sfd The open socket file descriptor
 * @param port The port number passed in from command line
 * @return none
 */
void listen_on_socket(int sfd, int port)
{
    printf("Listening on sock %d port %d\n", sfd, port);
    sockaddr_in cli_addr;
    bzero((char *) &cli_addr, sizeof(sockaddr_in));
    socklen_t clilen = sizeof(sockaddr);
    int newsockfd, pid;
    signal(SIGCHLD, SIG_IGN);
    while(1)
    {
        newsockfd = accept(sfd, (struct sockaddr *) &cli_addr,
&clilen);
        if(newsockfd < 0) error("Error accepting connection\n");
        printf("Opening client %d\n", cli_addr.sin_port);
        pid = fork();
        if(pid < 0) error("Error forking proccess\n");
        if(pid == 0){
            // Child Process
            char buf[MP1::buf_size];
            char ip_str[INET_ADDRSTRLEN];
            ssize_t size;
            // Stores the string of the client ip address into ip_str
            inet_ntop(AF_INET, &(cli_addr.sin_addr), ip_str,
INET_ADDRSTRLEN);
            while(1)
            {
                // Zero the buffer to prevent reading stale values
                bzero(buf, MP1::buf_size);
                size = read_socket(newsockfd, buf, 0);
                if(size == 0) break;
                printf("Received @ %s::%d size %d: %s", ip_str,
cli_addr.sin_port, (int) size, buf);
```

```
                if(!strstr(buf, "\n")) printf("\n");
                printf("Echoing: %s", buf);
                if(!strstr(buf, "\n")) printf("\n");
                size = write_socket(newsockfd, buf, 0);
            }
            // After receiving EOF, Close the socket and end the child
process
            printf("Closing client %d\n", cli_addr.sin_port);
            close(newsockfd);
            exit(MP1::GOOD);
        }
        // Parent Process
        close(newsockfd);
    }
}

int main(int argc, char *argv[])
{
    int port, sfd;
    char *port_str[20];
    // Handle issues with port number
    if(argc != 2) error("Call must have 1 argument: PORT_NUM\n");
    port = strtol(argv[1], port_str, 10);
    if(port <= 1024) error("Error port cannot be less than 1024\n");
    // Assign socket struct variables
    sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons((uint16_t) port);
    // Bind the socket
    sfd = bind_socket(&serv_addr);
    // Listen to socket ( infinite loop )
    listen_on_socket(sfd, port);
    return MP1::GOOD;
}
```

mp1.h

```
namespace MP1
{
    static const int buf_size = 80;
    static const int list_queue_size = 32;
    static const int ERROR = -1;
    static const int GOOD = 0;
};
```

README

```
# EchoServer
## Authors
- Chris Baldwin
- Kaushal Prudhvi

## Roles
Chris Baldwin
* Source Code
* Makefile
* README

Kaushal Prudhvi
* Code Review
* Testcases
* Report

__To Install__
1. Clone the repo
2. Open a terminal (server) in the repository folder
3. run `make` to make the project
4. run `hostname -I` to get your local ip address
5. run `./run_server <PORT_NUM>`
6. Open a terminal (client) in the repository folder
7. run `./run_client <IP_ADDR> <PORT_NUM>` (Note that
   the PORT_NUM should match the server port number)

__Running The Server__
* The buffer size is 80 Bytes
* The client can type Cntrl-D or Cntrl-C to end their session
```