1st Group Homework Report

TCP Echo Server and Client

ECEN 602 Network Programming Assignment 1

Mainly, we worked together, but here is the role what we did.

Minhwan Oh : Developed server programming, debugged for integration

Sanghyeon Lee : Developed client programming, tested for integration

File info

==========

For this program, you can see how TCP echo works. There are two main files.

1. echos

Path: cd tcpcliserv/echos.c

Main feature: TCP Server

2. echo

Path: cd tcpcliserv/echo.c

Main feature: TCP Client

3. unp.h

Path: cd tcpcliserv/unp.h

Main feature: Packages for socket programming function *Refers to Unix Network Programming library*

Program scenario

==========

This program includes client and server for a simple TCP echo service, which does the following:

1. Start the server first with the command line: $echos <Port>, where echos is the name of the server program and Port is the port number on which the server is listening. The server must support multiple simultaneous connections.

2. Start the client second with a command line: $echo <IPAdr> <Port>, where echo is the name of the client program, IPAdr is the IPv4 address of the server in dotted decimal notation, and Port is the port number on which the server is listening.

3. Type any word with a command line where client is running, then you can see the echo.

Program details - Echo

==========

1. The client reads a line of text from its standard input and writes the line to the network output to the server.

2. The sever reads the line from its network input and echoes the line back to the client.

3. The client reads the echoed line and prints it on its standard input.

4. When the client reads an EOF from its standard input (e.g., terminal input of Control-D), it closes the socket and exits. When the client closes the socket, the server will receive a TCP FIN packet, and the server child process' read() command will return with a 0. The child process should then exit

File architecture

==========

1. Server

a) Architecture

- Scenario :

  (1) Setup bind/listen/sig_child function, then going to infinite loop until the exit

  (2) Each child process has different PID

- Main function :

  (1) int main(int argc, char **argv): Including main scenario

  (2) void str_echo(int sockfd): Received data, and sending echo data

b) Feature

- Create socket and bind the well-known port of server
- Wait for client and accept the connection
- Make and print out child process to support multiple simultaneous connections from clients

2. Client

a) Architecture

- Scenario :

  (1) Setup socket/connect function, then going to infinite loop until the exit

- Main function :

  (1) int main(int argc, char **argv): Including main scenario

  (2) void str_cli(FILE *fp, int sockfd): Sending data, and received echo data

Test cases

==========

Current program passed below test cases

(1) line of text terminated by a newline - If you type words and enter key, you can see echo data.

(2) line of text the maximum line length without a newline - Even if you type more than number of maximum line, you can see echo data due to running send/read function automatically

(3) line with no characters and EOF - If you type just enter key, you can see enter key is working

(4) client terminated after entering text - If you type terminal input of Control-D, client closes the socket and exit. Server cleans up the client process.

(5) three clients connected to the server - If you run three client programs at same time, you can see all clients are working

Reference

==========

Below functions are referred by network programming library, W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff, Unix Network

Programming, Volume 1, The Sockets Networking API, 3rd Edition

1) echo.c

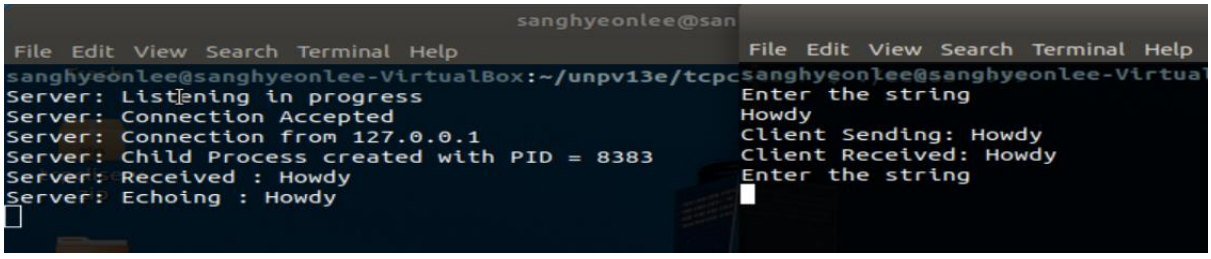
- static ssize_t my_read(int fd, char *ptr)
- ssize_t readline(int fd, void *vptr, size_t maxlen)
- ssize_t Readline(int fd, void *ptr, size_t maxlen)
- ssize_t writen(int fd, const void *vptr, size_t n) /* Write "n" bytes to a descriptor. */
- char * Fgets(char *ptr, int n, FILE *stream)
- void Writen(int fd, void *ptr, size_t nbytes)

2) echos.c

- Sigfunc * signal(int signo, Sigfunc *func)
- Sigfunc * Signal(int signo, Sigfunc *func) /* for our signal() function */
- ssize_t writen(int fd, const void *vptr, size_t n) /* Write "n" bytes to a descriptor. */
- void Writen(int fd, void *ptr, size_t nbytes)
- void sig_chld(int signo)

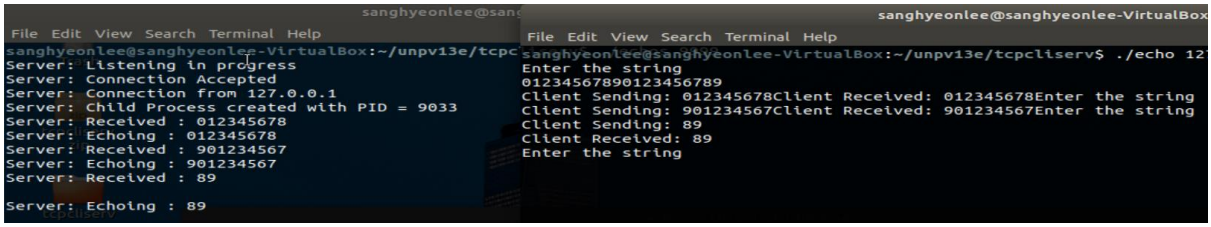
Details - Test Cases

(1) Line of text terminated by a newline



The newline “Client Received: Howdy” is terminated the line of text. And the client is waiting again to get an input. This shows that the TCP echo server can echo the client’s message to the client. In the program, str_echo function sends the received data.

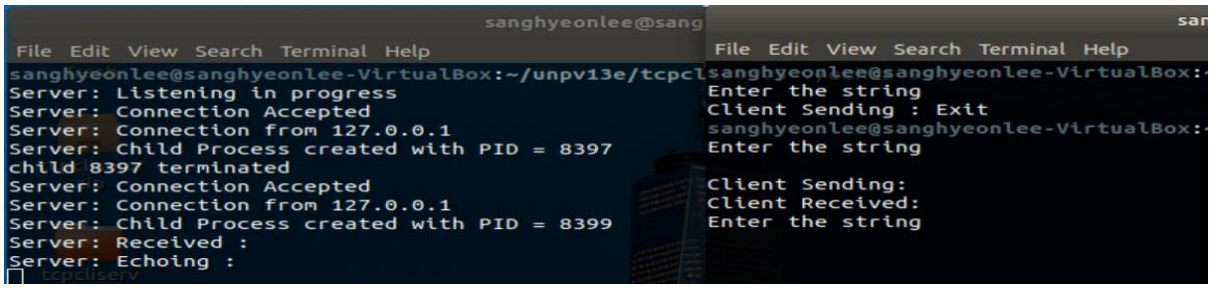
(2) Line of text the maximum line length without a newline




```
#define MAXECHOLINE      10  #define MAXECHOLINE      100
```

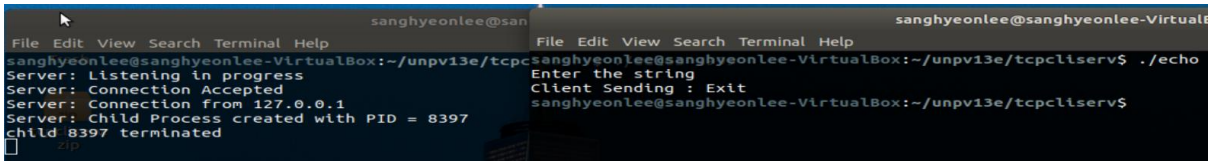
When a message’s length is over the maximum length which is defined as MAXECHOLINE in client and server file, client only send the maximum length of the original message. However, since its last character is not EOF, the Receiving message is printed on the same line not on the new line. Also, client send the remained message to server until the buffer is empty.

(3) Line with no characters and EOF



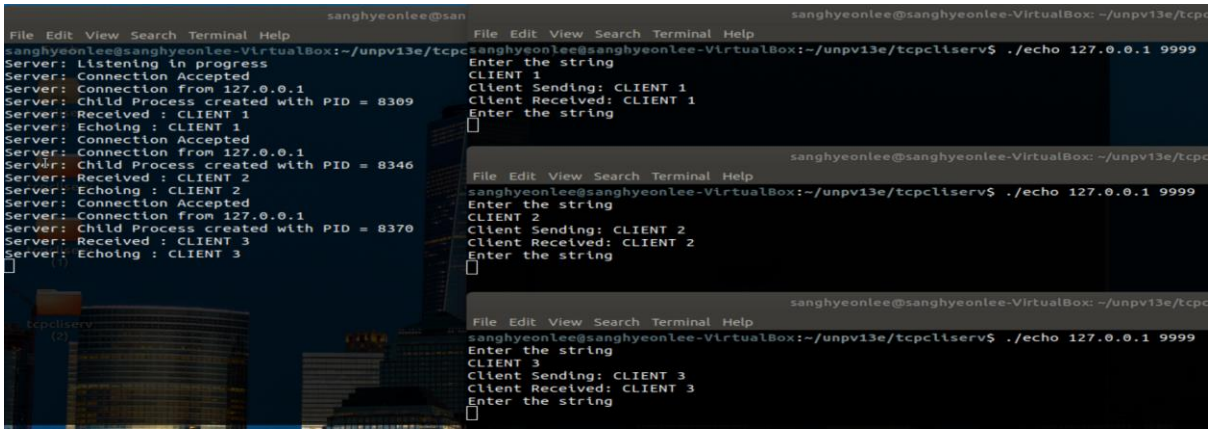
When we put enter value, the server only gets \r\n. The original print code of “Server: Received :” and “Server: Echoing :” doesn’t contain \n. Thus, we can see that \r\n value is actually transmitted and echoed on the server. In this moment we can understand that the enter value is also considered as character on the network.

(4) Client terminated after entering text



When we put “control+D”, the client reads an EOF and closes the socket, and then the client prints "Client Sending : Exit” before it exits the server.

(5) Three clients connected to the server



Through fork function, the server supports multiple simultaneous connections from clients. We accessed the server from three different child clients and checked its different PID in the server. Then, we confirmed each echo connections with server.

Source Code

echo.c

```
#include	"unp.h"

#define MAXECHOLINE		4096

/* Fatal error related to system call
 * Print message and terminate */

static int	read_cnt;
static char	*read_ptr;
static char	read_buf[MAXECHOLINE];

static ssize_t my_read(int fd, char *ptr)
{

	if (read_cnt <= 0) {
again:
		if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
			if (errno == EINTR)
				goto again;
			return(-1);
		} else if (read_cnt == 0)
			return(0);
		read_ptr = read_buf;
	}

	read_cnt--;
	*ptr = *read_ptr++;
	return(1);
}

ssize_t readline(int fd, void *vptr, size_t maxlen)
{
	ssize_t	n, rc;
	char	c, *ptr;

	ptr = vptr;
	for (n = 1; n < maxlen; n++) {
		if ( (rc = my_read(fd, &c)) == 1) {
			*ptr++ = c;
			if (c == '\n')
				break;		/* newline is stored, like fgets() */
		} else if (rc == 0) {
			*ptr = 0;
			return(n - 1);		/* EOF, n - 1 bytes were read */
```

```
                        } else
                                return(-1);             /* error, errno set by read() */
        }

        *ptr = 0;       /* null terminate like fgets() */
        return(n);
}

ssize_t Readline(int fd, void *ptr, size_t maxlen)
{
        ssize_t                 n;

        if ( (n = readline(fd, ptr, maxlen)) < 0)
                printf("readline error");
        return(n);
}

/* include writen */
#include    "unp.h"

ssize_t     writen(int fd, const void *vptr, size_t n) /* Write "n" bytes to a descriptor. */
{
        size_t                  nleft;
        ssize_t                 nwritten;
        const char      *ptr;

        ptr = vptr;
        nleft = n;
        while (nleft > 0) {
                if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
                        if (nwritten < 0 && errno == EINTR)
                                nwritten = 0;                   /* and call write() again */
                        else
                                return(-1);                     /* error */
                }

                nleft -= nwritten;
                ptr   += nwritten;
        }
        return(n);
}
/* end writen */

char * Fgets(char *ptr, int n, FILE *stream)
{
        char            *rptr;

        if ( (rptr = fgets(ptr, n, stream)) == NULL && ferror(stream))
                printf("fgets error");

        return (rptr);
}


void Writen(int fd, void *ptr, size_t nbytes)
{
        if (writen(fd, ptr, nbytes) != nbytes)
                printf("writen error");
}

void str_cli(FILE *fp, int sockfd)
{
        char            sendline[MAXECHOLINE], recvline[MAXECHOLINE];
        int nReadline;

        printf("Enter the string\n");
        //Fgets terminated if buf is NULL
        while (Fgets(sendline, MAXECHOLINE, fp) != NULL)
        {
                printf("Client Sending: %s",sendline);
                //Sending data
                Writen(sockfd, sendline, strlen(sendline));

                //Check the length of received data
```

```
                        nReadline = Readline(sockfd, recvline, MAXECHOLINE);

                        //If the length of received data is 0, print out error msg
                        if( nReadline == 0)
                                printf("str_cli: server terminated prematurely");

                        printf("Client Received: %s",recvline);
                        printf("Enter the string\n");
            }
            //After terminated, print out this to command window
            printf("Client Sending : Exit \n");
}


int
main(int argc, char **argv)
{
            int                                                     sockfd;
            struct sockaddr_in      servaddr;

            if (argc != 3)
                        printf("usage: echo <IPaddress> <Port>");

            sockfd = socket(AF_INET, SOCK_STREAM, 0);

            bzero(&servaddr, sizeof(servaddr));
            servaddr.sin_family = AF_INET;
            ///Input Port data
            servaddr.sin_port = htons(atoi(argv[2]));
            inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

            connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

            str_cli(stdin, sockfd);                 /* do it all */

            exit(0);
}
```

echos.c

```
#include    "unp.h"

#define MAXECHOLINE                     4096

Sigfunc * signal(int signo, Sigfunc *func)
{
            struct sigaction            act, oact;

            act.sa_handler = func;
            sigemptyset(&act.sa_mask);
            act.sa_flags = 0;
            if (signo == SIGALRM) {
#ifdef      SA_INTERRUPT
                        act.sa_flags |= SA_INTERRUPT;   /* SunOS 4.x */
#endif
            } else {
#ifdef      SA_RESTART
                        act.sa_flags |= SA_RESTART;                 /* SVR4, 44BSD */
#endif
            }
            if (sigaction(signo, &act, &oact) < 0)
                        return(SIG_ERR);
            return(oact.sa_handler);
}
/* end signal */

Sigfunc * Signal(int signo, Sigfunc *func)          /* for our signal() function */
{
            Sigfunc     *sigfunc;

            if ( (sigfunc = signal(signo, func)) == SIG_ERR)
                        printf("signal error");
            return(sigfunc);
```

```
}


ssize_t writen(int fd, const void *vptr, size_t n) /* Write "n" bytes to a descriptor. */

{
	size_t		nleft;
	ssize_t		nwritten;
	const char	*ptr;

	ptr = vptr;
	nleft = n;
	while (nleft > 0) {
		if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
			if (nwritten < 0 && errno == EINTR)
				nwritten = 0;		/* and call write() again */
			else
				return(-1);		/* error */
		}

		nleft -= nwritten;
		ptr   += nwritten;
	}
	return(n);
}
/* end writen */

void Writen(int fd, void *ptr, size_t nbytes)
{
	if (writen(fd, ptr, nbytes) != nbytes)
		printf("writen error");
}

void sig_chld(int signo)
{
	pid_t	pid;
	int		stat;

	pid = wait(&stat);
	printf("child %d terminated\n", pid);
	return;
}

void str_echo(int sockfd)
{
	ssize_t		n;
	char		buf[MAXECHOLINE];

again:
	while ( (n = read(sockfd, buf, MAXECHOLINE)) > 0)
	{
		printf("Server: Received : %.*s\n",n, buf);
		//Sending echo data
		Writen(sockfd, buf, n);
		printf("Server: Echoing : %.*s\n",n, buf);
	}

	if (n < 0 && errno == EINTR)
		goto again;
	else if (n < 0)
		printf("str_echo: read error");
}

int main(int argc, char **argv)
{
	int		listenfd, connfd;
	pid_t		childpid;
	socklen_t		clilen;
	struct sockaddr_in	cliaddr, servaddr;
	void		sig_chld(int);
	char		childaddr[14];

	if (argc != 2)
		printf("usage: echos <Port>");
```

```
	listenfd = socket(AF_INET, SOCK_STREAM, 0);

	bzero(&servaddr, sizeof(servaddr));
	servaddr.sin_family      = AF_INET;
	servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
	servaddr.sin_port        = htons(atoi(argv[1]));

	bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

	listen(listenfd, LISTENQ);
	printf("Server: Listening in progress \n");

	//For cleaning up zombie process, add sig_child func
	Signal(SIGCHLD, sig_chld);

	for ( ; ; )
	{
		clilen = sizeof(cliaddr);
		if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0)
		{
			if (errno == EINTR)
				continue;		/* back to for() */
			else
				printf("accept error");
		}
	printf("Server: Connection Accepted \n");

		//Get IP Address
		inet_ntop(cliaddr.sin_family, &cliaddr.sin_addr, childaddr,sizeof(childaddr));
	printf("Server: Connection from %s\n", childaddr);

		if ( (childpid = fork()) == 0)
		{
			/* child process */
			//Get Child PID Number
			printf("Server: Child Process created with PID = %d \n", getpid());
			close(listenfd);		/* close listening socket */
			str_echo(connfd);		/* process the request */
			exit(0);
		}
		close(connfd);				/* parent closes connected socket */
	}
}
```

README

TCP Echo Server and Client

TCP Echo Server and Client ECEN 602 Network Programming Assignment 1

Role

Mainly, we worked together, but here is the role what we did. Minhwan Oh : Developed server programming, debugged for integration Sanghyeon Lee : Developed client programming, tested for integration

File info

For this program, you can see how TCP echo works. There are two main files.

1. echos Path: cd tcpcliserv/echos.c Main feature: TCP Server
2. echo Path: cd tcpcliserv/echo.c Main feature: TCP Client
3. unp.h Path: cd tcpcliserv/unp.h Main feature: Packages for socket programming function *Refers to Unix Network Programming library*
4. makefile Path: cd tcpcliserv/makefile

Build info

Go to tcpcliserv folder, then $make

Program scenario

This program includes client and server for a simple TCP echo service, which does the following and:

1. Start the server first with the command line: $echos , where echos is the name of the server program and Port is the port number on which the server is listening. The server must support multiple simultaneous connections.

2. Start the client second with a command line: $echo , where echo is the name of the client program, IPAdr is the IPv4 address of the server in dotted decimal notation, and Port is the port number on which the server is listening.

3. Type any word with a command line where client is running, then you can see the echo.

Program details - Echo

1. The client reads a line of text from its standard input and writes the line to the network output to the server.

2. The sever reads the line from its network input and echoes the line back to the client.

3. The client reads the echoed line and prints it on its standard input.

4. When the client reads an EOF from its standard input (e.g., terminal input of Control-D), it closes the socket and exits. When the client closes the socket, the server will receive a TCP FIN packet, and the server child process' read() command will return with a 0. The child process should then exit

File architecture

1. Server a) Architecture
    - Scenario : (1) Setup bind/listen/sig_child function, then going to infinite loop until the exit (2) Each child process has different PID
    - Main function : (1) int main(int argc, char **argv): Including main scenario (2) void str_echo(int sockfd): Received data, and sending echo data

b) Feature - Create socket and bind the well-known port of server - Wait for client and accept the connection - Make and print out child process to support multiple simultaneous connections from clients

2. Client a) Architecture
    - Scenario : (1) Setup socket/connect function, then going to infinite loop until the exit
    - Main function : (1) int main(int argc, char **argv): Including main scenario (2) void str_cli(FILE *fp, int sockfd): Sending data, and received echo data

Test cases

Current program passed below test cases (1) line of text terminated by a newline - If you type words and enter key, you can see echo data. (2) line of text the maximum line length without a newline - Even if you type more than number of maximum line, you can see echo data due to running send/read function automatically (3) line with no characters and EOF - If you type just enter key, you can see enter key is working (4) client terminated after entering text - If you type terminal input of Control-D, client closes the socket and exit. Server cleans up the

client process. (5) three clients connected to the server - If you run three client programs at same time, you can see all clients are working

Reference

Below functions are referred by network programming library, W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff, Unix Network Programming, Volume 1, The Sockets Networking API, 3rd Edition

1. echo.c
    - static ssize_t my_read(int fd, char *ptr)
    - ssize_t readline(int fd, void *vptr, size_t maxlen)
    - ssize_t Readline(int fd, void *ptr, size_t maxlen)
    - ssize_t writen(int fd, const void *vptr, size_t n) /* Write "n" bytes to a descriptor. */
    - char * Fgets(char *ptr, int n, FILE *stream)
    - void Writen(int fd, void *ptr, size_t nbytes)
2. echos.c
    - Sigfunc * signal(int signo, Sigfunc *func)
    - Sigfunc * Signal(int signo, Sigfunc *func) /* for our signal() function */
    - ssize_t writen(int fd, const void *vptr, size_t n) /* Write "n" bytes to a descriptor. */
    - void Writen(int fd, void *ptr, size_t nbytes)
    - void sig_chld(int signo)