CS 4414: Operating Systems

Machine Problem 4: FTP Server

Michael Eller

2 December 2016

I was successful in implementing the bare-bones FTP server.

Introduction

The File Transfer Protocol (FTP) is used to share files in a computer network between a client and server. FTP is an application layer protocol and uses two different TCP (Transmission Control Protocol) connections for control and data between the client and server. This is a bare-bones implementation, meaning that the only functions implemented are the minimum required to have a functioning server.

Implementation

Using POSIX sockets, my program opens a control connection at the port given as a command-line parameter. The server then listens for a client and accepts the connection. All communication between the client and server occurs on this socket. For security and testing purposes, the accepted IP address is hard-coded to be the loopback address (127.0.0.1). This server can only service one client session at a time but can accept consecutive sessions without closing.

The server can perform three basic commands: LIST, RETR, and STOR. In order to send any data to the client, however, the client must send a PORT message, which provides an arbitrary port number on the client to open a data connection. If the client attempts to perform a STOR or a RETR without first having set the TYPE to 'I' (Image), the server will return code 451: "Requested action aborted. Local error in processing." For LIST, the root directory for the server is the same directory in which the program was executed. The only allowable parameters for STRU and MODE are 'F' (File) and 'S' (Stream) respectively. The server simply responds with a 200 response code if it receives a USER command because the client is assumed to be running with the anonymous flag.

LIST

When the client user types 'ls' the client first sends a PORT command to open the data connection. After the server connects, it sends a response code 200 telling the client that the operation was successful. Then the client sends the LIST command. Upon receiving LIST, the server appends whatever path the client wanted to the command string 'ls -l' and uses *system*() to execute the command into a temporary file. In order to only send the filename and file size in compliance with the assignment specification, the temporary file is then parsed line by line and appended to a string stream. The server then sends a return code 125 to the client over the control connection to signal that it is ready to start sending data. Then the server writes the string stream to the data connection and sends a return code 226 over the control connection to signal that it is done sending data.

STOR and RETR

Similar to the LIST command, when the client enters a 'put' or 'get' command, the client sends a PORT command first to tell the server on what port and IP address (loopback address in this case) to connect a socket. The first thing that the server does when it receives a STOR or RETR command is check to make sure the client has set the type to binary. If it is not in binary mode, the server will return code 451, telling the client that there was an error.

For STOR, the objective is to create a file on the server that is a copy of the specified client's file. The server first creates the file locally and returns code 125 to signal the it is ready to receive data from the client. Then the server reads from the data socket 1KB at a time and writes it to the local file. Finally, the server returns code 226 to tell the client that the process is complete. RETR is the same as STOR but reversed. The server opens the specified local file and continually reads 1KB into a buffer and writes that buffer to the client over the data socket.

Problems Encountered

The single biggest problem in completing this assignment was figuring out the messages to send back to the client. I made little to no progress for a long time because I did not realize that the client expects certain response codes before it proceeds. Another issue that I had was in implementing the PORT command, specifically parsing the port number. I neglected the distinction between normal order and network order.

Testing

The testing I performed was fairly straight forward. Since we were not required to implement the client code, I knew that if the output on the client was correct for LIST, it was correct. For STOR and RETR, I simply validated that the transferred files were identical to the originals.

Conclusion

This machine problem was similar to the others in that the hardest part was getting started. Once I figured out the protocol, sending and receiving files was trivial compared to the file manager implemented in machine problem three. My implementation of the bare-bones FTP server is correct and complies with all specifications.

Code

makefile

```
## Michael Eller mbe9a
## OS Machine Problem 4
## 2 December 2016
## makefile

OBJS = ftp.o
CC = g++
DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)

my_ftpd: $(OBJS)
	$(CC) $(LFLAGS) $(OBJS) -o my_ftpd

ftp.o: ftp.h ftp.cpp
	$(CC) $(CFLAGS) ftp.cpp

clean:
	\rm *.o *~ my_ftpd

tar:
	tar cfv mbe9a.tar my_ftpd ftp.h ftp.cpp makefile mbe9a.pdf
```

ftp.h

```
// Michael Eller mbe9a
// OS Machine Problem 4
// 29 November 2016
// ftp.h (includes, defs, and protos)

#ifndef FTP_H
#define FTP_H

#define SRC125        "125 Data connection already open; transfer starting.\n"
#define SRC125LEN        53
#define SRC150        "150 File status okay; about to open data connection.\n"
#define SRC150LEN        53
#define SRC200        "200 The requested action has been successfully completed.\n"
#define SRC200LEN        58
#define SRC220        "220 Service ready for new user.\n"
#define SRC220LEN        32
#define SRC221        "221 Service closing control connection.\n"
#define SRC221LEN        40
#define SRC226        "226 Closing data connection. Requested file action successful (for example, file transfer or file abort).
#define SRC226LEN        106
#define SRC451        "451 Requested action aborted. Local error in processing.\n"
#define SRC451LEN        57
#define SRC504        "504 Command not implemented for that parameter.\n"
#define SRC504LEN        48

#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <sstream>
```

```
#include <string.h>
#include <list>
#include <errno.h>
#include <dirent.h>

using namespace std;

#endif
```

ftp.cpp

```
// Michael Eller mbe9a
// OS Machine Problem 4
// 29 November 2016
// ftp.cpp (main)

#include "ftp.h"

/*
 * This is a bare-bones implementation of an FTP server.
 * For security and testing purposes, the accepted IP address is hard-coded to be the loopback address (127.0.0.1).
 * The port that the server operates on is set by the only command-line parameter.
 * This server can only service one client session at a time but can accept consecutive sessions without closing.
 *
 * Implementation:
 *                TYPE - ASCII Non-print
 *                MODE - Stream
 *                STRUCTURE - File, Record
 *                COMMANDS - QUIT, PORT, TYPE,
 *                                MODE, STRU, USER (for default values)
 *                        and
 *                                RETR, STOR, NOOP, and LIST
 *
 * It was assumed that the ftp client would be operated with the anonymous flag, so the server's response
 * to USER is simply a return code of 200. Similarly for MODE and STRU, the assumption is that the mode will only be 'S'
 * (Stream) and the file structure will only be 'F' (File), so they return code 504 if the client attempts to set them to
 * anything else.
 *
 * Additionally, if the client attempts to perform a STOR or a RETR without first having set the TYPE to 'I' (Image),
 * the server will return code 451: Requested action aborted. Local error in processing. For LIST, the root directory for
 * the server is the same directory in which the executable was started.
 */

int main(int argc, char *argv[])
{
        // declare server variables
        char buffer[100];
        char* token;
        int command;
        bool TYPEI = false;
        struct sockaddr_in control_addr, client_addr;
        int socket_control_fd = socket(AF_INET, SOCK_STREAM, 6);
        int socket_client_fd;

        // set all bytes of the socket address struct to 0
        memset(&control_addr, 0, sizeof control_addr);

        // intialize the local control address struct
        control_addr.sin_family = AF_INET;
        control_addr.sin_port = htons(atoi(argv[1]));
        control_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

```
// check and see if they were initialized correctly
if (socket_control_fd == -1)
{
        perror("cannot create socket");
        exit(EXIT_FAILURE);
}
// bind and the control socket to the control address struct, check if unsuccessful
if (bind(socket_control_fd, (struct sockaddr *)&control_addr, sizeof control_addr) == -1)
{
        perror("control bind failed");
        close(socket_control_fd);
        exit(EXIT_FAILURE);
}
// listen on the control socket, check if unsuccessful
if (listen(socket_control_fd, 1) == -1)
{
        perror("control listen failed");
        close(socket_control_fd);
        exit(EXIT_FAILURE);
}

// outer loop, this enables multiple sessions (consecutively not simultaneously)
while (1)
{
        // accept the control connection, check if unsuccessful
        int control_connection_fd = accept(socket_control_fd, NULL, NULL);
        if (control_connection_fd < 0)
        {
                perror("control accept failed");
                close(socket_control_fd);
                exit(EXIT_FAILURE);
        }
        // return code 220 - service ready for new user
        write(control_connection_fd, SRC220, SRC220LEN);

        // inner loop - service current session
        while(1)
        {
                // get command from client
                command = read(control_connection_fd, &buffer, 100);
                if (command < 0)
                {
                        perror("receive on control socket failed");
                        close(control_connection_fd);
                        close(socket_control_fd);
                        exit(EXIT_FAILURE);
                }
                // ignore carriage returns and new line characters
                for (int x = 0; (unsigned)x < strlen(buffer); x++)
                        if ( buffer[x] == '\n' || buffer[x] == '\r' ) buffer[x] = '\0';
                cout << buffer << endl;
                // split on spaces
                token = strtok((char*)buffer, " ");

                // quit session if command is exit
                if(!strcmp(token, "QUIT")) break;

                // return 200 if USER is received, shouldn't happen but here just in case
                if(!strcmp(token, "USER")) write(control_connection_fd, SRC200, SRC200LEN);
```

```
// port command to open the data connection
else if (!strcmp(token, "PORT"))
{
        // reinitialize the socket
        socket_client_fd = socket(AF_INET, SOCK_STREAM, 6);
        if (socket_client_fd == -1)
        {
                perror("cannot create data socket");
                exit(EXIT_FAILURE);
        }
        // retrieve port number for client data connection
        token = strtok(NULL, ",");
        for (int i = 0; i < 4; i++) token = strtok(NULL, ",");

        unsigned int client_port[2];
        for (int i = 0; i < 2; i++)
        {
                client_port[i] = atoi(token);
                if (i == 1) break;
                token = strtok(NULL, ",");
        }

        // build client address struct
        memset(&client_addr, 0, sizeof client_addr);
        client_addr.sin_family = AF_INET;
        client_addr.sin_port = htons((unsigned short)((client_port[0] * 256) + client_port[1]));
        client_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

        // return success code for PORT command
        write(control_connection_fd, SRC200, SRC200LEN);

        // connect to the client socket
        if (connect(socket_client_fd, (struct sockaddr *)&client_addr, sizeof client_addr) == -1)
        {
                perror("connection to client failed");
                close(socket_client_fd);
                exit(EXIT_FAILURE);
        }
}
// ls
else if (!strcmp(token, "LIST"))
{
        // build command
        string command = "ls -l";
        string command_suffix = " > .results";
        string path;
        token = strtok(NULL, " ");
        if (token != NULL)
        {
                path = string(token);
                command += " " + path;
        }
        command += command_suffix;

        // call ls and write it to results (a temporary file)
        remove(".results");
        system(command.c_str());

        // open the file and read from it
        FILE* results = fopen(".results", "r");
        char buffer[256];
        char* size;
        char* filename;
        stringstream ss;

        //burn the first line
        fgets(buffer, 256, results);
```

```
        // continue with the rest of the file
        while(fgets(buffer, 256, results))
        {
                // ignore carriage returns and new line characters
                for (int x = 0; (unsigned)x < strlen(buffer); x++)
                        if ( buffer[x] == '\n' || buffer[x] == '\r' ) buffer[x] = '\0';
                size = strtok(buffer, " ");
                for (int i = 0; i < 4; i++) size = strtok(NULL, " ");
                for (int i = 0; i < 4; i++) filename = strtok(NULL, " ");

                // add to the string stream
                ss << filename << "\t" << size << "\r\n";
        }
        fclose(results);
        remove(".results");

        // return code 125 - data connection already open; transfer starting
        write(control_connection_fd, SRC125, SRC125LEN);

        // write the string to the client
        write(socket_client_fd, ss.str().c_str(), ss.str().length());
        close(socket_client_fd);

        // return code 226 - closing data connection. Requested file action successful
        write(control_connection_fd, SRC226, SRC226LEN);
}
// check the type parameter and set the type boolean accordingly
else if (!strcmp(token, "TYPE"))
{
        token = strtok(NULL, " ");
        if (!strcmp(token, "I")) TYPEI = true;
        else TYPEI = false;
        write(control_connection_fd, SRC200, SRC200LEN);
}
// check MODE
else if (!strcmp(token, "MODE"))
{
        token = strtok(NULL, " ");
        if (!strcmp(token, "S")) write(control_connection_fd, SRC200, SRC200LEN);
        else write(control_connection_fd, SRC504, SRC504LEN);
}
// check STRU
else if (!strcmp(token, "STRU"))
{
        token = strtok(NULL, " ");
        if (!strcmp(token, "F")) write(control_connection_fd, SRC200, SRC200LEN);
        else write(control_connection_fd, SRC504, SRC504LEN);
}
// no operation
else if (!strcmp(token, "NOOP")) write(control_connection_fd, SRC200, SRC200LEN);
```

```
// retrieve
else if (!strcmp(token, "RETR"))
{
        // check that the type is binary first
        if (!TYPEI)
        {
                write(control_connection_fd, SRC451, SRC451LEN);
                continue;
        }
        // open the file
        token = strtok(NULL , " \n\r");
        FILE* retr_file = fopen(token, "rb");
        if (retr_file == NULL)
        {
                perror("could not open server-side file");
                fclose(retr_file);
                exit(EXIT_FAILURE);
        }
        // get the file size
        fseek(retr_file, 0, SEEK_END);
        int filesize = ftell(retr_file);
        fseek(retr_file, 0, SEEK_SET);

        // return code 125
        write(control_connection_fd, SRC125, SRC125LEN);

        // read the file into a buffer
        char retr_buffer[1024];
        while (filesize > 1024)
        {
                fread(retr_buffer, 1024, 1, retr_file);

                // write to the client
                write(socket_client_fd, retr_buffer, 1024);

                // update filesize
                filesize -= 1024;
        }

        if (filesize != 0)
        {
                fread(retr_buffer, filesize, 1, retr_file);

                // write to the client
                write(socket_client_fd, retr_buffer, filesize);
        }
        // close the stuffs
        fclose(retr_file);
        close(socket_client_fd);

        // send 226 back to the client
        write(control_connection_fd, SRC226, SRC226LEN);
}
```

```
                        // store file on the server
                        else if (!strcmp(token, "STOR"))
                        {
                                // check that the type is binary first
                                if (!TYPEI)
                                {
                                        write(control_connection_fd, SRC451, SRC451LEN);
                                        continue;
                                }
                                // open the file
                                token = strtok(NULL , " \n\r");
                                int stor_file = open(token, O_CREAT | O_TRUNC | O_RDWR, S_IRWXU);
                                if (stor_file == -1)
                                {
                                        perror("could not create server-side file");
                                        close(stor_file);
                                        exit(EXIT_FAILURE);
                                }
                                // start reading from the client
                                char buffer[1024];

                                // return code 125
                                write(control_connection_fd, SRC125, SRC125LEN);
                                int bytes;
                                while ((bytes = read(socket_client_fd, buffer, 1024)) > 0) write(stor_file, buffer, bytes);

                                // close and finish
                                close(stor_file);
                                close(socket_client_fd);

                                // send 226 back to the client
                                write(control_connection_fd, SRC226, SRC226LEN);
                        }
                }
                // return code 221 - service closing control connection
                write(control_connection_fd, SRC221, SRC221LEN);

                // shutdown the control and data connections, check if unsuccessful
                if (shutdown(control_connection_fd, SHUT_RDWR) == -1)
                {
                        perror("shutdown failed");
                        close(control_connection_fd);
                        close(socket_control_fd);
                        exit(EXIT_FAILURE);
                }
                // set the socket as reusable to the kernel
                int i = 1;
                setsockopt(socket_control_fd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(int));
        }
        // close the stuffs, return
        close(socket_control_fd);
        return EXIT_SUCCESS;
}
```