CS 4414: Operating Systems – Spring 2018

Machine Problem 5: Simple FTP

Cassie Willis

Completion Level:

The code works to the full extent that it was tested. To the programmer's knowledge, the code is fully functional.

Problem Description:

The goal of this assignment was to develop a program that uses FTP (File Transfer Protocol), and in particular TCP (Transmission Control Protocol) to become familiar with remote procedure calls (RPC). The program uses sockets from different built in C and C++ libraries to initialize connections between a client and a server that can exchange messages between each other. Here, the client was the built in FTP client in Linux, and the server was what was implemented by the designer. The server first establishes connection with the FTP client, then can send and receive messages. The messages sent are based on the messages received and include the error or return code and any output for the given command.

Approach:

The approach to developing this FTP server was to use Berkeley sockets, which is an API for internet sockets and one way to do socket implementation. The initial socket setup for the server was based off of the Wikipedia Berkeley sockets examples that was given as a resource in the writeup. The Wikipedia page gave full C examples of both TCP and UDP clients and servers, and the TCP server in particular was used in this implementation. The server was set up all in main because it kept the functions more clear in their own if statements without having to hunt for different functions for debugging.

For the simple FTP server, several messages had required implementation in order to ensure that the server functioned correctly. These client messages, based on the FTP protocol guide, included USER, PORT, QUIT, TYPE, MODE, STOR, RETR, NOOP, and STRU. In addition to these functions, LIST was also implemented. If the client message received was able to be processed by the server properly, the server sent back a positive return code as well as any output that the message requires (such as LIST returning the output from a ls command). If the request was not successful, the server sends back an error message based on the error that caused the request to be unsuccessful. These message implementations are described below, along with main. The server only handles binary file types in the streaming mode.

USER:

This message is the first message that has to be sent by the FTP client any time a connection is setup. The built in FTP client prompts the user automatically on connection for a name. This message can also be sent using the “user” prompt. The USER message accepts any user name as usernames are not implemented for multiple clients in this implementation. Because any username is acceptable, this function simply sends return code 230 for a successful login.

PORT:

This message has more complex operations than the majority of the messages handled by the server. The purpose is to set up a new socket for the LIST, STOR, and RETR messages to send data on. The PORT first has to take in its arguments for the IP address and port number that the new socket will be set up on. It tokenizes the arguments using spaces and commas, similar to the tokenization in the implementation of Machine Problem 4. These tokens are pushed onto a vector of strings which is used to create the port1 and port2 integers. As specified in the FTP specification document, the two port inputs are added together (overallPort = port1 x 256 + port2) to determine the new port which the new socket will be set up on. The message then sets up this new socket, similar to the client TCP code found on the Wikipedia Berkeley sockets page and connects this socket. If all of this is successful, the PORT sends a 200 return code for a successful command.

QUIT:

This message sends return code 221 for logout and breaks out of the while loop encasing all of the message if-statements in main. Doing this allows the program to enter the shutdown code at the end of main which quits the open FTP client.

TYPE:

This message checks whether the type given is “I” or another value. I-type means that the file is binary, which is the only type supported for this simple FTP system. If the user tries to change the type to binary, the return code 200 is sent for a successful command. If the user tries to change to any other type, the return code 504 command not implemented is sent to the client. Finally, a Boolean value typeI is set to true stating that the type is an I-type if the 200 return code is sent.

MODE:

This message checks whether the mode of the server connection is stream (“S”). If this mode is stream, the server sends a return code 200. If the mode is set to anything else, the server sends a 504 command not implemented error code to show the user that modes other than stream are not implemented for this simple FTP system.

STOR:

The message is used to begin the transmission of a file *to* the remote host to the client. This can only be done if the type was already switched to binary mode, so if the typeI Boolean has not been set to true in the TYPE message command, the server returns a "451 Requested action aborted" code. Otherwise, the server sends code 125 for file handling and tokenizes the arguments from the buff buffer to determine the name of the file being written to or created on the server side. This file is opened and recv and fputc are used to transfer the data from the socket to the server-side file. The file and socket are then closed. If the transmission of a file is done successfully, a 226 code is returned.

RETR:

The message is used to begin the transmission of a file *from* the remote host to the client. This can only be done if the type was already switched to binary mode, so if the typeI Boolean has not been set to true in the TYPE message command, the server returns a "451 Requested action aborted" code. Similarly, if the file cannot be found, a "550 File not found" error code is returned to the client, the socket is closed, and 226 is returned. This is determined based on the FILE descriptor when calling fopen. If the file opens successfully, fgetc and send are used to transfer the data from the server file to the client file over the socket. The file and socket are then closed. If the transmission of a file is done successfully, a 226 code is returned for command okay.

NOOP:

This message simply stands as a busy message to send when the server and client communication socket is idle. Therefore, this message always sends a 200 return code for a successful command.

STRU:

This message checks whether the structure is a file ("F"). If the structure is a file, the server sends the 200 return code saying the message was okay. If any other structure type is requested, the server sends the 504 command not implemented error code, letting the user know that this server only includes implementation for file structures.

LIST:

This message is the FTP functionality for the "ls -l" command that would normally be called in the normal kernel command prompt. The server first sends a 125 return code for file status okay. It then uses popen to open the "ls -l" command to a FILE and iterates through each byte of the file, sending the file through the socket set up with PORT to the client whenever a newline is encountered. It then closes the socket and the file and sends 226 to show the client that the file action was successful.

Main():

This function houses the complete functionality of the TCP server. It takes in one argument when running – the port that the server will communicate to the client on. The main function first sets up the socket between the server and the client. The sockaddr_in struct "sa" that was created is memset to all zeroes to ensure the memory space is cleared. The port for sa is set to the argument argv[1] given as the main parameter. The bind function binds the socket to an address, and the listen command prepares for incoming messages. The main function then enters a for() loop where the new socket is created for each connection with the accept function, and the 220 return code is sent saying that the server is ready for the new user.

Main then enters into a while(1) loop where the command and any arguments are received from the client into buff using the recv() function, and the first four characters in buff are copied into the command buffer. This completes the setup for sending messages based on the command given from the FTP client. Each message is handled by comparing the given message to be implemented to the command using strncmp. An else statement at the end sends a "504 command not implemented" return code to handle any commands not implemented by the simple FTP server.

The function exits the while loop and closes the socket connection to the client. It then exits out of the for loop and closes the socket, sending an EXIT_SUCCESS return, however this command should never be reached as the server never exits.

Results:

The program is able to successfully create a socket connection between the server and the client and send messages between them for each of the 10 messages required in a simple FTP. The program can be run using the 'make' command, which creates a my_ftpd.o target executable. The command "./my_ftpd <port>" can be used to run the target.

The program was tested thoroughly by the author by setting up the FTP client and the server on separate command prompt windows. The server and client were set up on the same port, and the help function within FTP was used to determine commands that could be called to send different messages. Multiple of these commands were sent to ensure complete functionality of the server to the author's knowledge.

Analysis:

As evident by the complete functionality, to the author's knowledge, of the FTP server code, this project was a success. This finding was particularly important to the author as she really needs a good grade on this homework to pass this class. Though some errors occurred in developing this program, the author also learned a great deal from this machine problem. With this assignment, the author was able to more deeply understand FTP. The author also learned more about the C++ programming language.

The author did run into a few errors in creating this solution. The biggest setback was in attempting to implement LIST, RETR, and STOR, as these functions are more complex than simply comparing arguments and sending return codes. Office hours helped clear up confusion about the operation of these messages.

Finally, the author continued to learn about time management strategies and planning. This project was completed over a greater period of time, which caused less stress and less feeling of a time crunch. Each of the messages were pseudo-coded and heavily commented to ensure that the author had a clear path of how to implement the functions in the C++ programming language, and the author took advantage of TA office hours, which was a significant help in completing the project.

Conclusion:

FTP servers and clients are a very interesting and somewhat-complex area of operating systems. It is interesting to see how FTP is used and how this can be expanded into more complex data connections with multiple users and security implementations. It would have been helpful to have more instruction on how to go about our implementation of a TCP server outside of the Wikipedia page. The return codes could also have been better defined for what to send when. It was useful to take a hands-on approach to FTP in order to more thoroughly understand it, as the author feels as though much was learned through this process.

Pledge:

On my honor as a student, I have neither given nor received aid on this assignment.

Cassie Willis

Code Dump:

```
/* Cassie Willis
 *
 * CS 4414 - Operating Systems
 * Spring 2018
 * 5-1-18
 *
 * MP5 - Simple FTP
 *
 * The purpose of this project is to become familiar with remote
 * procedure calls, remote file systems, and FTP. The FTP processes
 * all basic user commands along with LIST. The return is any error
 * or return codes as specified in FTP protocol.
 * Refer to the writeup for complete details.
 *
 * Compile with MAKE
 *
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
#include <unistd.h>
#include <iostream>
using namespace std;

//the server-PI interprets commands, sends replies and directs its DTP to set up the data connection and
transfer the data
//The Server has one command-line parameter: the port number upon which to listen and
//accept incoming control connections from Clients.

//Server will include barebones of section 5.1 in FTP documentation except:
// In order to make FTP workable without needless error messages, the
//   following minimum implementation is required for all servers:
//         ~ TYPE - BINARY
//         ~ MODE - Stream
//         ~ STRUCTURE - File
//         ~ COMMANDS - USER, QUIT, PORT, TYPE, MODE, STRU, RETR, STOR, NOOP, LIST

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

```
struct sockaddr_in sa;
int SocketFD = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP); //Create a socket for endpoint
        communication
if (SocketFD == -1) { //Creation failed
  perror("cannot create socket");
  exit(EXIT_FAILURE);
}

memset(&sa, 0, sizeof sa);

sa.sin_family = AF_INET;
sa.sin_port = htons(atoi(argv[1]));  //host to network short -> user input >1024
sa.sin_addr.s_addr = htonl(INADDR_ANY); //only accept from local host

if (bind(SocketFD,(struct sockaddr *)&sa, sizeof sa) == -1) { //Bind the socket to an address
  perror("bind failed");
  close(SocketFD);
  exit(EXIT_FAILURE);
}

if (listen(SocketFD, 10) == -1) { //Prepare for incoming connections (this has 10 max connections)
  perror("listen failed");
  close(SocketFD);
  exit(EXIT_FAILURE);
}

char* buff = new char[256];
char* command = new char[4];
bool typeI = false;
int SocketFD1;

for (;;) {
  int ConnectFD = accept(SocketFD, NULL, NULL); //creates a new socket for each connection (for
        data) and removes is from listen queue
```

```
if (0 > ConnectFD) {
  perror("accept failed");
  close(SocketFD);
  exit(EXIT_FAILURE);
}
else printf("%s\n", "220 Service ready for new user.");

send(ConnectFD, "220\n", 4, 0); //Initial connection
while(1) {
  recv(ConnectFD, buff, 256, 0);
  memcpy(&command, &buff, 4);

  //determine which command was called and handle the commands
  //Return Codes:
  // 200 -> successful command
  // 125 -> file status okay
  // 230 -> user is logged in
  // 220 -> ready for new user
  // 226 -> file action successful
  // 451 -> action aborted
  // 500 -> syntax error, command did not occur
  // 504 -> command not implemented for that parameter
  // Everything sends 200 on success, 500 on failure unless otherwise noted

  //USER is always valid, send 230
  if(strncmp("USER", command, 4) == 0) {
    send(ConnectFD, "230\n", 4, 0);
  }
  //QUIT closes all client sockets, terminating command connection, send 221 because logout
  else if(strncmp("QUIT", command, 4) == 0) {
    send(ConnectFD, "221\n", 4, 0);
    break;
  }
```

```
//PORT specifies the host and port to which the server should connect for the next file transfer, 200 on
    success
//Syntax: "PORT a1,a2,a3,a4,p1,p2" is interpreted as IP address a1.a2.a3.a4, port p1*256+p2.
else if(strncmp("PORT", command, 4) == 0) {
  //separate the arguments
  char* tokens;
  vector<string> arguments;
  tokens = strtok(buff, " ");
  while(tokens != NULL) {
    arguments.push_back(tokens);
    tokens = strtok(NULL, ",");
  }
  // string newIP = arguments[1] + "." + arguments[2] + "." + arguments[3] + "." + arguments[4];
  // const char* ip = newIP.c_str();
  int port1 = atoi(arguments[5].c_str());
  int port2 = atoi(arguments[6].c_str());
  int overallPort = port1 * 256 + port2;

  struct sockaddr_in sa1;

  SocketFD1 = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP); //Create a socket (an endpoint
    for communication)
  //Socket creation failed
  if (SocketFD1 == -1) {
    perror("cannot create socket");
    exit(EXIT_FAILURE);
  }

  memset(&sa1, 0, sizeof sa1);

  sa1.sin_family = AF_INET;
  sa1.sin_port = htons(overallPort);

  if (connect(SocketFD1, (struct sockaddr *)&sa1, sizeof sa1) == -1) { //connect to server
```

```
      perror("connect failed");
      close(SocketFD);
      exit(EXIT_FAILURE);
    }
    send(ConnectFD, "200\n", 4, 0);

  }
  //TYPE supports TYPE I command when "binary" types into prompt, send 200 if I, else send 504
  else if(strncmp("TYPE", command, 4) == 0) {
    if(strncmp("I", &buff[5], 1) == 0) {
      send(ConnectFD, "200\n", 4, 0);
      typeI = true;
    }
    else send(ConnectFD, "504 Command not implemented for that parameter\n", 47, 0);
  }
  //MODE should be stream, send 200 if S (stream), 504 otherwise
  else if(strncmp("MODE", command, 4) == 0) {
    if(strncmp("S", &buff[5], 1) == 0) send(ConnectFD, "200\n", 4, 0);
    else send(ConnectFD, "504 Command not implemented for that parameter\n", 47, 0);
  }
  //STRU with file (F) send 200, STRU R (else) returns “504 Command not implemented for that
parameter”
  else if(strncmp("STRU", command, 4) == 0) {
    if(strncmp("F", &buff[5], 1) == 0) send(ConnectFD, "200\n", 4, 0);
    else send(ConnectFD, "504 Command not implemented for that parameter\n", 47, 0);
  }
  //RETR Begins transmission of a file from the remote host, send “451 Requested action aborted: local
        error in processing” if not in image mode
  //550 file not found error code
  else if(strncmp("RETR", command, 4) == 0) {
    if(typeI == false) send(ConnectFD, "451 Requested action aborted: local error in processing\n", 56,
        0);
    else {
      send(ConnectFD, "125\n", 4, 0);
```

```
char* tokens;
vector<string> arguments;
tokens = strtok(buff, " ");
while(tokens != NULL) {
  arguments.push_back(tokens);
  tokens = strtok(NULL, " ");
}

char result[1000000];

FILE *fd = fopen(arguments[1].c_str(), "r");
if(fd == NULL) {
  send(ConnectFD, "550 File Not Found\n", 19, 0);
  close(SocketFD1);
  send(ConnectFD, "226\n", 4, 0);
  break;
}
else {
  int i = 0;
  int c;
  while((c = fgetc(fd)) != EOF){
    if(c == '\n'){
      result[i++] = '\n';
      send(SocketFD1, result, i, 0);
      i = 0;
    }
    else result[i++] = c;
  }
  if(i > 0) {
    send(SocketFD1, result, i, 0);
  }
}
fclose(fd);
```

```
        close(SocketFD1);
        send(ConnectFD, "226\n", 4, 0);
      }
    }
    //STOR Begins transmission of a file to the remote site, send “451 Requested action aborted: local error in processing” if not in image mode
    else if(strncmp("STOR", command, 4) == 0) {
      if(typeI == false) send(ConnectFD, "451 Requested action aborted: local error in processing\n", 56, 0);
      else {
        send(ConnectFD, "125\n", 4, 0);

        char* tokens;
        vector<string> arguments;
        tokens = strtok(buff, " ");
        while(tokens != NULL) {
          arguments.push_back(tokens);
          tokens = strtok(NULL, " ");
        }

        char result[1000000];

        string open = arguments[1];
        FILE *fd = fopen(open.c_str(), "w");

        int info = recv(SocketFD1, result, 10000000, 0);

        for(int i = 0; i < info; i++) {
          fputc(result[i], fd);
        }

        fclose(fd);
        close(SocketFD1);
        send(ConnectFD, "226\n", 4, 0);
    }
  }
  //NOOP does nothing but return a response 200
  else if(strncmp("NOOP", command, 4) == 0) {
    send(ConnectFD, "200\n", 4, 0);
  }
  //LIST lists remote files (LS), 200, 500
  else if(strncmp("LIST", command, 4) == 0) {
    send(ConnectFD, "125\n", 4, 0);
    char result[1000];
    FILE *ls = popen("ls -l", "r");

    int i = 0;
    int c;
    while((c = getc(ls)) != EOF){
      if(c == '\n'){
        result[i++] = '\r';
        result[i++] = '\n';
        send(SocketFD1, result, i, 0);
        i = 0;
      }
      else result[i++] = c;
    }
    pclose(ls);
    close(SocketFD1);
    send(ConnectFD, "226\n", 4, 0);
  }
  //Not implemented, send 504
  else send(ConnectFD, "504 Command not implemented for that parameter\n", 47, 0);
  //clear the buffers for the next command
  memset(buff, 0, 256);
  memset(command, 0, 5);
}
```

```
    if (shutdown(ConnectFD, SHUT_RDWR) == -1) {
      perror("shutdown failed");
      close(ConnectFD);
      close(SocketFD);
      exit(EXIT_FAILURE);
    }
    close(ConnectFD);
  }

  close(SocketFD);
  return EXIT_SUCCESS;
}
```