ICT 374 Assignment 2

Tee Yee Kang (CT0347801/34315323)

Lim Wen Chao (CT0360379/34368872)

Table of Content

List of Files:

File name	Remarks/Purpose
makefile	A special file used to automate the process of compilation and linking
server.c	The main program of SFTP client. Responsible for connecting to server and serving the user by executing different commands
client.c	The main program of SFTP server. Responsible for creating connection to client in order to perform different commands
stream.h	Header file of stream (store in the folder “stream”) - contains different functions needed to support file transfer between client and server
stream.c	Implementation file of stream (store in the folder “stream”) - contains different functions needed to support file transfer between client and server
sFunctionHelper.h	Header file of sFunctionHelper (store in the folder “serverFunction”) - contains all the functions needed to execute different commands for the server program
sFunctionHelper.c	Implementation file of sFunctionHelper (store in the folder “serverFunction”) - contains all the functions needed to execute different commands for the server program
logger.h	Header file of logger (store in the folder “logger”) - include functions needed to log all the interactions between the clients and server
logger.c	Implementation file of logger (store in the folder “logger”) - include functions needed to log all the interactions between the clients and server
cFunctionHelper.h	Header file of cFunctionHelper (store in the folder “clientFunction”) - contains all the functions needed to execute different commands for the client program
cFunctionHelper.c	Implementation file of cFunctionHelper (store in the folder “clientFunction”) - contains all the functions needed to execute different commands for the client program
token.h	Header file of tokenizer (store in the folder “clientFunction”) - A tokenizer used to split the given string to tokens
token.c	Implementation file of tokenizer (store in the folder “clientFunction”) - A tokenizer used to split the given string to tokens

myftpd	Server executable program (store in the folder “myftpdExe”) - used to execute the server program. Place in separate folder so that the program is able to run on same machines
sftp_log.txt	logfile (store in the folder “myftpdExe”) - include all the interaction between server and client
myftp	Client executable program (store in the folder “myftpExe”) - used to execute the client program. Place in separate folder so that the program is able to run on same machines
big.txt	text file (store in the folder “myftpExe”) - used to test if the program is able to transfer the text file correctly
bigbig.txt	text file with bigger size/more content (store in the folder “myftpExe”) - used to test if the program is able to transfer the text file correctly
markingguide.doc	binary file (store in the folder “myftpExe”) - used to test if the program is able to transfer the binary file correctly

The project title and a brief description of the project:

Project 2: A Simple File Transfer Protocol

Design and implement a simple network protocol that can be used to download files from a remote site and to upload files to a remote site, and a client and a server programs that communicate using that protocol. The protocol should use TCP as its transport layer protocol. The server must be able to serve multiple client requests simultaneously. For simplicity, do not consider any authentication process in this project, hence the server will provide its service to any client with the right site address and port number.

To simplify project marking, please name your server program myftpd with the following command line syntax (here the letter d in myftpd stands for daemon, since the server should run as a daemon process):

myftpd [initial_current_directory]

The server process maintains a current directory. Its initial value should be the one inherited from its parent process unless the optional

initial_current_directory

is given. In the latter case, the user supplied path should be used to set the initial current directory of the server. This can be done using the function chdir. A client can use the cd command to change the (child) server's current directory later.

The client program should be named myftp with the following command line syntax:

myftp [hostname | IP_address]

where the optional *hostname* (*IP_address*) is the name (address) of the remote host that provides the myftp service. If the *hostname* or *IP_address* is omitted, the local host and the default port number is assumed. Use the port number allocated to you as your default port number.

After the connection between the client and the server is established, the client should display the prompt > and wait for one of the following commands:

- `pwd` - to display the current directory of the server that is serving the client;
- `lpwd` - to display the current directory of the client;
- `dir` - to display the file names under the current directory of the server that is serving the client;
- `ldir` - to display the file names under the current directory of the client;
- cd directory_pathname - to change the current directory of the server that is serving the client; Must support "." and ".." notations.
- `lcd` directory_pathname - to change the current directory of the client; Must support "." and ".." notations.

- `get` filename - to download the named file from the current directory of the remote server and save it in the current directory of the client;
- `put` filename - to upload the named file from the current directory of the client to the current directory of the remove server.
- `quit` - to terminate the myftp session.

The myftp client should repeatedly display the prompt and wait for a command until the quit command is entered.

This project consists of three components:

1. ***Protocol Specification:***
   You must specify a protocol for communication between the myftp client and the myftp server. This protocol will become the *sole* reference to which the client program and the server program can be separately implemented. This means that, by referring to the protocol, the client and server can be implemented independent of each other. The design and implementation of the client (server) should not depend on (1) what strategy and algorithms were used to implement the server (client), (2) what programming language were used to implement the server (client), and (3) what operating system the server (client) is running on. The protocol must be complete, i.e., it contains all the necessary information required by both parties to communicate, such as the format and sequence of data exchanges between the client and the server, the transport layer protocol (TCP or UDP) used to deliver the data, and the server port number. However, it should not contain anything that unnecessarily constrains the implementation of the client or the server. For example, there is no point to limit the implementation language to a particular programming language in the protocol, or to require the client or the server to be implemented with a particular data structure. Before you attempt this project, you should first complete the first two exercises in Lab 10. You may also read the Trivial FTP Protocol given in Appendix 17A of the textbook (Stallings: Chapter 17 of Online Chapters). Note TFTP uses UDP, while myftp protocol must use TCP.
2. ***Client Program:***
   You need to implement a client program according to the myftp protocol. Note that the client should never rely on any undocumented internal tricks in the server implementation. Apart from the myftp protocol, you should not make any other assumption about the server program in your client program.
3. ***Server Program:***
   You need to implement a server program according to the myftp protocol. Note that the server should never rely on any undocumented internal tricks in the client implementation. Apart from the myftp protocol, you should not make any other assumption about the client program in your server program.

Note the following additional requirements/explanations:

- The name of the client executable program must be myftp. The name of the server executable program must be myftpd.
- To avoid potential conflict in the use of the server's ``well known'' port number when several students use the ceto server, you should use the TCP port allocated to you as the default server listening port.
- Tests should show the cases where both the client and the server are on the same machine as well as on different machines (you may want to use ceto.murdoch.edu.au

to do the testing, however ***you must kill your server at the end of the test***). They should also show that more than one client can obtain the service at the same time. You may use the command script to record the terminal I/O. When debugging your network program on a standalone machine (e.g., on your home Linux), you may use localhost as the ``remote'' host name.

- Tests should show that the program can transfer not only small files (e.g., 0, 10 and 100 bytes) but also large files (at least several mega bytes), and not only text files but also binary files. In addition, you must show that the transferred file and the original file are identical not only in their sizes but also in their contents (use diff command).
- The specification of the protocol is an important part of this question. Please provide a full specification of the protocol in a section separated from the client and server implementations. You should complete this specification before starting the implementation of the client and server programs.
- Please note that the server must be implemented as a daemon and must log all of its interactions with the clients.

Self diagnosis and evaluation:

a. a list of features that are fully functional

1. The program is able to run on same machine and different machines.
2. `pwd` – The program is able to display the current directory of the server that is serving the client
3. `lpwd` – The program is also able to display the current directory of the client
4. `quit` – The program can continue running until user enters "quit" command to terminate.
5. `dir` – The program can display the file names under the current directory of the server that is serving the client by using the "dir" command
6. `ldir` - The program is able to display the file names under the current directory of the client by using the "ldir" command.
7. `cd` - to change the current directory of the server that is serving the client
8. `lcd` - to change the current directory of the client
9. The program is able to execute correctly with the address and port number given by the user.
10. The program is also able to execute correctly with the default port number.
11. The program is able to `put` and `get` the big.txt file, bigbig.txt and binary file (markingguide.doc)
12. The program is able to display an error message to tell the user if a wrong command is entered instead of crashing the program
13. The program is also able to display an error message to alert the user when the user tries to put or get a non-existent file instead of crashing the program
14. The program is able to display error message for get, put and cd when user do not give path/file name instead of crashing the program

b. a list of features that are not fully functional (include features that are not fully implemented, or not implemented at all).

—

Discussion of your solution

Most of the data structure or algorithms are using the concepts taught in the given note, such as how to serve the client, creating daemon processes, claiming childs, communication using TCP, etc.

However, in the get function in the “sFunctionHelper”, we added the EINVAL and EACCES to check if the server program is unable to transfer the file to the client due to some reason such as invalid or file does not exist.

Protocol

1. Spec for pwd command

Client sends one message to Server. The message format as follows:

- One byte opcode which is ASCII character E

Server responds with the following message:

- One byte opcode which is ASCII character E followed by
- Four-byte integer (N) in two's complement and in network byte order, which represents the length of the message to be sent followed by
- A sequence of N bytes of ASCII characters which is the message.

2. Spec for dir command

Client sends one message to Server. The message format as follows:

- One byte opcode which is ASCII character F

Server responds with the following message:

- One byte opcode which is ASCII character F followed by
- Four-byte integer (N) in two's complement and in network byte order, which represents the length of the message to be sent followed by
- A sequence of N bytes of ASCII characters which is the message.

3. Spec for cd command

Client sends one message to the Server. The message format as follows:

- One byte opcode which is ASCII character DG followed by
- Two-byte integer (N) value in two's complement and in network byte order, which represents the length of the new working directory path followed by
- The sequence of N ASCII characters representing the new working directory path

Upon receiving the message with opcode DG, the server responds with the following:

- One byte opcode which is ASCII character DG followed by
- One byte acknowledgement code which is one of the following ASCII characters:
    - 0 – the server successfully changed the working directory to the new working directory
    - 1 – the server was unsuccessful in changing the working directory to the new working directory

4. Spec for get command

Client sends one message to the Server. The message format as follows:

- One byte opcode which is ASCII character G followed by
- Two-byte integer value in two's complement and in network byte order, which represents the length of the filename of the file to be sent to the client followed by
- The sequence of ASCII characters representing the filename

Once Server receives the message with opcode G, it responds with the following message:

- One byte opcode which is ASCII character G followed by
- One byte acknowledgement code which is one of the following ASCII characters:
    - 0 – the Server is able to send the named file
    - 1 – Access denied or file do not exist (EACCES)
    - 2 – Invalid file name (EINVAL)
    - 3 – The server cannot accept the file due to other reasons

Client acknowledges the server if it receives the G opcode message with acknowledgement code 0, it responds with the following message:

- One byte opcode which is ASCII character G followed by

- One by acknowledgement code which is one of the following ASCII characters:
    - 0 – The client is able to receive the named file
    - 1 – Server unable to send file, client ends get
    - 2 – Server sent unexpected ack code, client ends get
    - 3 – Server sent unexpected ack code, client ends get
    - 4 – File already exist in client
    - 5 – Permission denied (EACCES)
    - 6 – File name invalid (EINVAL)
    - 7 – Client is unable to receive the named file due to other reason

Once the Server receives the G opcode message with acknowledgement code 0, it responds by sending the following message:

- One byte opcode which is ASCII Character B followed by
- Four-byte integer (N) in two's complement and in network byte order which represents the length of the file followed by
- A sequence of N bytes which is the content of the file

5. Spec for put command

Client sends one message to the server. The message format is given below:

- one byte opcode which is ASCII character A followed by
- two-byte integer value in two's complement and in network byte order, which represents the length of the filename of the file to be sent to the server followed by
- the sequence of ASCII characters representing the filename

Once the server receives a message with opcode A, it responds with the following message:

- one byte opcode which is ASCII character A followed by
- one byte acknowledgment code which is one of the following ASCII characters:
    - 0 - the server is ready to accept the named file

- 1 – Access denied or file do not exist (EACCES)
- 2 – Invalid file name (EINVAL)
- 3 – The server cannot accept the file due to other reasons

Once the client receives the A message from the server and if the acknowledgment code is 0, it responds by sending the following message:

- one byte opcode which is the ASCII character B followed by
- four-byte integer (let's call it N) in two's complement and in network byte order which represents the length of the file followed by
- a sequence of N bytes which is the content of the file

Test evidence:

Note: We have created 2 separate folders called myftpdExe and myftpExe. These two folders contain the server and client executable program respectively. The purpose of these two folders is used to run the program. In addition, we can also use these two folders to test the program on the same machines. Furthermore, there are two text files (big.txt and bigbig.txt) and one binary file (markingguide.doc) stored in the myftpExe folder. These test files are used to test if the program can successfully transfer the file between the client and server. If you wish to test the "get" command, you can simply shift all the test files to the myftpdExe folder, else you can use it to test the "put" command directly.

In some of the test, we purposely move some of the files into the myftpdExe and myftpExe folders for testing purposes, such as testing command dir and ldir. (in the submitted folder to LMS will not contain such files in the myftpExe and myftpdExe folder).

● Test 1

The purpose of this test is to check if the program is able to get the server current directory using the "pwd" command. After the user has entered command "pwd", the program should display the current directory of the server that is serving the client. In this test, we will run both client and server programs on the same machine. Therefore, the program will use the default address and port number. According to the test output, the program successfully displays the correct current directory of the server.

Command line used:

```
> pwd
```

Test output - client (myftp):

```
> pwd
command: pwd
/home/tee/myftpdExe
```

Test output - server (Terminal):

```
$pwd
/home/tee/myftpdExe
```

- # Test 2

  This test is similar to test 1. However, the purpose of this test is to test on 2 different machines. We will run the server program by using the remote server ceto. Therefore, when running the client program, we will need to provide the ceto address and the port number. In this test, the address for ceto server is "ceto.murdoch.edu.au" and the port number used is 40263. According to the test output, the client program is able to display the correct current directory of the server, which is "/home/student/accounts/34315323/myftpd".

  Command line used – server (myftpd):

  ```
  % ./myftpd 40263
  ```

  Command line used – client (myftp):

  ```
  % ./myftp ceto.murdoch.edu.au 40263
  > pwd
  ```

  Test output – server (myftpd):

  ```
  Daemon pid: 30986
  ```

  Test output – client (myftp):

  ```
  > pwd
  command: pwd
  /home/student/accounts/34315323/myftpdExe
  ```

  Test output - server(Terminal):

  ```
  $pwd

  /home/student/accounts/34315323/myftpdExe
  ```

- # Test 3

  The purpose of test 3 is to test the command lpwd. Command lpwd is used to display the current directory of the client. This command does not require any argument. In this test, we will also run the server and client program on 2 different machines. The server program will be run on the ceto and the client program will run on the virtual box machines. According to the test output, the client program is able to display the current of the client correctly.

  Command line used – server (myftpd):

  ```
  % ./myftpd 40263
  ```

  Command line used – client (myftp):

  ```
  % ./myftp ceto.murdoch.edu.au 40263
  > lpwd
  ```

  Test output – server (myftpd):

  ```
  Daemon pid: 30986
  ```

Test output – client (myftp):

```
> lpwd
command: lpwd
/home/tee/myftpExe
```

Test output - client (Terminal):

```
$pwd
/home/tee/myftpExe
```

- # Test 4

The purpose of this test is to test the "quit" function. According to the question requirement, the client program should repeatedly display the prompt and wait for a command until the quit command is entered. Therefore, test 4 is to test if the program is able to continue executing until the quit command is entered. In this test, we will also run the server and client program on 2 different machines. The server program will be run on the ceto and the client program will run on the virtual box machines. According to the test output, the program is able to continue executing until the command quit is entered. In addition, the program also successfully displays the goodbye message after the command "quit" is entered.

Command line used – client (myftp):

```
% ./myftp ceto.murdoch.edu.au 40263
> lpwd
> pwd
> quit
```

Test output – client (myftp):

```
> lpwd
command: lpwd
/home/tee/myftpExe
> pwd
command: pwd
/home/student/accounts/34315323/myftpdExe
> quit
Bye from the client
```

- # Test 5

  The purpose of this test is to test if the program can successfully display the file names under the current directory of the server that is serving the client. This command does not require any argument. In this test, we will run both server and client programs on the same machines. According to the test output, the program is able to display all the file names under the current directory of the server.

  Command line used:

  ```
  > dir
  ```

  Test output - client (myftp):

  ```
  > dir
  command: dir
  makefile
  stream.h
  myftpd.o
  stream.c
  server
  ..
  big.txt
  bigbig.txt
  markingguide.doc
  .
  myftpd.c
  toClient.txt
  stream.o
  ```

  Test output - server (Terminal):

  ```
  $dir
  makefile        stream.h
  myftpd.o        stream.c
  server          big.txt
  bigbig.txt      markingguide.doc
  myftpd.c        toClient.txt
  stream.o
  ```

- # Test 6

The purpose of this test is to test if the program can successfully display the file names under the current directory of the server that is serving the client. However, in this test we will first change the current directory by using the command “cd” to demonstrate that the command “dir” is working correctly. This command does not require any argument. In this test, we will run both server and client programs on the same machines. According to the test output, the client program is able to display all the file names under the new directory “lab02/c_ex2” correctly.

Command line used:

```
> cd /home/tee/ICT374/lab02/c_ex2
> dir
```

Test output - client (myftp):

```
> dir
command: dir
..
foo2
question2.c
question2
.
```

Test output - server (Terminal):

```
$ dir
foo2            question2.c
question2
```

- # Test 7

The purpose of this test is to test if the program can successfully display the file names under the current directory of the server that is serving the client. This command does not require any argument. In this test, we will run the client and server program on 2 different machines. The client program will run on virtual box and the server program will run on remote server ceto. According to the test output, the client program is able to get the listing of the current directory on the server correctly using the "dir" command.

Command line used – server (myftpd):

```
% ./myftpd 40263
```

Command line used – client (myftp):

```
% ./myftp ceto.murdoch.edu.au 40263
> dir
```

Test output – server (myftpd):

```
Daemon pid: 30986
```

Test output – client (myftp):

```
> dir
command: dir
makefile
token.o
server.o
sftp_log.txt
server.c
cFunctionHelper.o
myftp
myftpd
..
client.o
clientFunction
serverFunction
client.c
sFunctionHelper.o
logger
.
logger.o
stream
stream.o
..
```

Test output - server (Terminal):

```
$ dir
cFunctionHelper.o    server.c
client               serverFunction
client.c             server.o
clientFunction       sftp_log.txt
client.o             sFunctionHelper.o
logger               stream
logger.o             stream.o
makefile             token.o
server
```

- # Test 8

  This test is similar to test 7. The purpose of this test is to test if the program can successfully display the file names under the current directory of the server that is serving the client. However, in this test, we will first change the current directory of the server by using the command “cd”. This is to ensure that the command “dir” can perform correctly. According to the command line, we first change the current directory of the server to “ICT374” by using the command “cd”. As a result, the program can successfully display all the file names of the new directory correctly.

  Command line used – server (myftpd):

  ```
  % ./myftpd 40263
  ```

  Command line used – client (myftp):

  ```
  % ./myftp ceto.murdoch.edu.au 40263
  > pwd
  /home/student/accounts/34315323/myftpdExe
  > cd /home/student/accounts/34315323/ICT374
  > pwd
  /home/student/accounts/34315323/ICT374
  > dir
  ```

  Test output – server (myftpd):

  ```
  Daemon pid: 30986
  ```

  Test output – client (myftp):

  ```
  > dir
  command: dir
  cli4.c
  .
  ser4.c
  lab9
  server4
  server
  ..
  ```

- # Test 9

  The purpose of this test is to test if the program is able to display the file names under the current directory of the client by using the “ldir” command. In this test, we will run both server and client programs on the same machines. This command does not require any argument. According to the test output, the program is able to display all the file names under the directory “myftp” of the client by using the “ldir” command.

  Command line used:

```
> ldir
```

  Test output - client (myftp):

```
> ldir
command: ldir
makefile
token.o
server.o
sftp_log.txt
server.c
cFunctionHelper.o
myftp
myftpd
..
client.o
clientFunction
serverFunction
client.c
sFunctionHelper.o
logger
.
logger.o
stream
stream.o
```

  Test output - client (Terminal):

```
$ dir
makefile            token.o
server.o            sftp_log.txt
server.c            cFunctionHelper.o
myftp               myftpd
client.o            clientFunction
serverFunction      client.c
sFunctionHelper.o   logger
logger.o            stream
stream.o
```

- # Test 10

The purpose of this test is also to test if the program is able to display the file names under the current directory of the client by using the "ldir" command. However, in this test, we will first change the current directory of the client by using the command "lcd". This is to ensure that the command "ldir" is working correctly. In this test, we will run both server and client programs on the same machines. This command does not require any argument. According to the command line, we first change the current directory of the client to directory "lab02/c_ex2. As a result, the client program is able to display all the file names under the new directory correctly.

Command line used:

```
> lpwd
/home/tee/myftpExe
> lcd /home/tee/ICT374/lab02/c_ex2
> lpwd
/home/tee/ICT374/lab02/c_ex2
> ldir
```

Test output - client (myftp):

```
> ldir
command: ldir
..
foo2
question2.c
question2
.
```

Test output - client (Terminal) :

```
$ dir
foo2            question2.c
question2
```

- # Test 11

The purpose of this test is to test if the program is able to change the change the current directory of the server that is serving the client by using the command "cd". This command requires an argument, which is the new directory the user wish to change to. In this test, we will run both server and client program on the same machines. According to the test output, the current directory has successfully change from directory "myftpd" to directory "lab05/c_ex4" by using the command "cd". When calling the command "pwd", we can now see that the new current directory is "lab05/c_ex4".

Command line used:

```
> cd /home/tee/ICT374/lab05/c_ex4
```

Test output:

```
> pwd
command: pwd
```

```
/home/tee/myftpdExe
> cd /home/tee/ICT374/lab05/c_ex4
> pwd
command: pwd
/home/tee/ICT374/lab05/c_ex4
```

- # Test 12

This test is similar to test 11. The purpose of test 12 is to test if the program is able to change the change the current directory of the server that is serving the client by using the command "cd". However, in this test, we will run the program on 2 different machines. The client program will run on the local virtual box and the server program will run on the remote server ceto. According to the command line, we change the current directory of the server from directory "myftpd" to "ICT374". After that, we use the command "dir" to display all the file names under the new directory in order to double check the result. As a result, the program can successfully change the current directory to "/home/student/accounts/34315323/ICT374". In addition, we command "dir" also display the correct file names under the new directory.

Command line used – server (myftpd):

```
% ./myftpd 40263
```

Command line used – client (myftp):

```
% ./myftp ceto.murdoch.edu.au 40263
> pwd
/home/student/accounts/34315323/myftpd
> cd /home/student/accounts/34315323/ICT374
> pwd
/home/student/accounts/34315323/ICT374
> dir
```

Test output – server (myftpd):

```
Daemon pid: 30986
```

Test output – client (myftp):

```
> pwd
command: pwd
/home/student/accounts/34315323/myftpdExe
> cd /home/student/accounts/34315323/ICT374
: Success
> pwd
command: pwd
/home/student/accounts/34315323/ICT374
> dir
command: dir
cli4.c
.
ser4.c
lab9
server4
server
..
```

- # Test 13

  The purpose of test 13 is to test if the program is able to change the current directory of the client by using the command “lcd”. This command requires the user to provide an argument, which is the new directory the user wishes to change to. According to the command line, we are trying to change the current directory of the client from directory “myftp” to directory “lab06/c_ex1”. As a result, the program is able to change the current directory to the new passed in directory named “`/home/tee/ICT374/lab06/c_ex1`”. The test output uses the command “lpwd” to check the result. In addition, we also use the command “ldir” to double check if the program has correctly changed the directory. According to all the displayed file names, the program is able to change the new current directory and display the file names under the new directory.

  Command line used:

```
> lpwd
/home/tee/myftpExe
> lcd /home/tee/ICT374/lab06/c_ex1
```

  Test output:

```
> lpwd
command: lpwd
/home/tee/myftp
> lcd /home/tee/ICT374/lab06/c_ex1
: Success
> lpwd
command: lpwd
/home/tee/ICT374/lab06/c_ex1
> ldir
command: ldir
..
c_ex1.c
q1
.
```

- # Test 14

  The purpose of this test is to check if the program is able to handle multiple clients concurrently. According to the question, the program should allow multiple clients to interact with the server concurrently. In this test we will run the server and 2 clients programs on the same machines. According to the rest output below, we run 2 client programs at the same time. In client 1, we change the current directory of client 1 to " `/home/tee/ICT374/lab01`". At the same time, we also changed the current directory of client 2 to "`/home/tee/ICT374/lab02`". As a result, the server program is able to handle multiple clients concurrently.

  Command line used – client 1:

  ```
  % ./myftp
  ```

  Test output – client 1:

  ```
  > lpwd
  /home/tee/myftpExe
  > lcd /home/tee/ICT374/lab01
  > lpwd
  /home/tee/ICT374/lab01
  > quit
  Bye from the client
  ```

  Command line used – client 2:

  ```
  % ./myftp
  ```

  Test output – client 2:

  ```
  > lpwd
  command: lpwd
  /home/tee/myftpExe
  > lcd /home/tee/ICT374/lab02
  : Success
  > lpwd
  command: lpwd
  /home/tee/ICT374/lab02
  > quit
  Bye from the client
  ```

- # Test 15

  The purpose of this test is to check if the program is able to run on different machines by using the given address and port number. In this test, we will use the default port number. Therefore, the user only needs to provide the address of the server and the program will use the default port number. The address used in this test is “ceto.murdoch.edu.au”. According to the command line, the user only provides the address of the ceto server. As a result, the program is working correctly as we perform some commands to test the program such as the pwd and dir.

  Command line used – server (myftpd):

  ```
  % ./myftpd
  ```

  Command line used – client (myftp):

  ```
  % ./myftp ceto.murdoch.edu.au
  ```

  Test output – server (myftpd):

  ```
  Daemon pid: 2751
  ```

  Test output – client (myftp):

  ```
  > pwd
  command: pwd
  /home/student/accounts/34315323/myftpdExe
  > dir
  command: dir
  makefile
  server.o
  sftp_log.txt
  server.c
  cFunctionHelper.o
  myftpd
  ..
  serverFunction
  client.c
  sFunctionHelper.o
  logger
  .
  logger.o
  stream
  stream.o
  > quit
  Bye from the client
  ```

- # Test 16

  The purpose of this test is to check if the program is able to run on different machines by using the given address and port number. In this test, we will provide the program with address and port number. Therefore, the user will need to key in both the address of the server and port number when executing the client. The address used in this test is “ceto.murdoch.edu.au” and the port number is 40263. According to the command line, the user provides both the address of the ceto server and port number. As a result, the program is working correctly as we perform some commands to test the program such as the pwd and dir.

  Command line used – server (myftpd):

  ```
  % ./myftpd 40263
  ```

  Command line used – client (myftp):

  ```
  % ./myftp ceto.murdoch.edu.au 40263
  ```

  Test output – server (myftpd):

  ```
  Daemon pid: 3237
  ```

  Test output – client (myftp):

  ```
  > pwd
  command: pwd
  /home/student/accounts/34315323/myftpdExe
  > dir
  command: dir
  makefile
  server.o
  sftp_log.txt
  server.c
  cFunctionHelper.o
  myftpd
  ..
  serverFunction
  client.c
  sFunctionHelper.o
  logger
  .
  logger.o
  stream
  stream.o
  > quit
  Bye from the client
  ```

- # Test 17

The purpose of this test is to check if the server program can be compiled with the makefile. In this test, we will purposely make some changes to the server program. We will change the expression of daemon pid. After that we will recompile using the makefile and run the server program. In this test, we will run both client and server programs on the same machine. According to the test output, the server program is able to compile using the makefile. In addition, the changes are also updated as the new expression of the daemon pid is shown.

Command line used – server (myftpd):

```
% touch server.c
% make
%./myftpd
```

Test output – server (myftpd):

```
% make
gcc -c server.c
gcc server.o stream.o logger.o sFunctionHelper.o -o myftpdExe/myftpd
% cd myftpdExe
%./myftpd
This is the new expression: 3926
```

- # Test 18

The purpose of this test is to check if the client program can be compiled with the makefile. In this test, we will purposely make some changes to the server program. We will change the symbol of the prompt from '>' to '$'. After that we will recompile using the makefile and run the client program. In this test, we will run both client and server programs on the same machine. According to the test output, the client program is able to compile using the makefile. In addition, the changes are also updated as the prompt now using the new symbol '$'.

Command line used – client (myftp):

```
% touch client.c
% make
%./myftp
```

Test output – client (myftp):

```
% make
gcc server.o stream.o logger.o sFunctionHelper.o -o myftpExe/myftpd
% cd myftpExe
%./myftp
$ pwd
command: pwd
/home/tee/myftpd
$ quit
Bye from the client
```

- # Test 19

  The purpose of this test is to check if the client program is able to run smoothly with the required command line syntax. In this test, we will run both server and client programs on the same machines and use the default address and port number. In order to test the client program, we will run several commands such as pwd, lpwd, cd, dir, etc. The program should continue running until the user enters “quit” to terminate. In the command line below, we have executed several commands in order to test the client program. According to the test output, the program is able to perform the corresponding command correctly and smoothly.

  Command line used – client (myftp):

```
%./client
> pwd
> dir
> cd /home/tee/ICT374/lab02
> pwd
> lpwd
> ldir
> quit
```

  Test output – client (myftp):

```
> pwd
command: pwd
/home/tee/myftpdExe
> dir
command: dir
makefile
server.o
sftp_log.txt
server.c
cFunctionHelper.o
myftpd
..
serverFunction
client.c
sFunctionHelper.o
logger
.
logger.o
stream
stream.o
> cd /home/tee/ICT374/lab02
> pwd
command: pwd
/home/tee/ICT374/lab02
> lpwd
command: lpwd
/home/tee/myftp
> ldir
command: ldir
makefile
stream.h
token.o
stream.c
```

```
..
myftp.o
token.h
myftp.c
token.c
toServer.txt
.
client
stream.o
> quit
Bye from the client
```

- # Test 20

The purpose of this test is to test if the "put" command can successfully transfer the file to the server. In this test, we will run both server and client programs on the same machine and use the default address and port number. In order to perform the "put" command, the user will need to provide the file name he/she wishes to transfer. In this test, we will use a text file called "big.txt" for testing purpose. According to the test output of the client program, the file is successfully transferred from client to server. In order to check if the newly created file in the server is the same as the original, we purposely rename the newly created file to "big2.txt" and check with the original "big.txt" by using the `diff` command. As a result, there is no return value after comparing both files, which means that there are no differences between the 2 files.

Command line used – client(myftp):

```
%./myftp
> put big.txt
> quit
```

Test output – client (myftp):

```
> put big.txt
command: put
Starting to send file to server
Sent: big.txt
> quit
Bye from the client
```

Testing - comparing file

```
% diff -s big.txt big2.txt
Files big.txt and big2.txt are identical
```

- # Test 21

The purpose of this test is to test if the "put" command can successfully transfer the file to the server. In this test, we will run both server and client programs on the same machine and use the default address and port number. In order to perform the "put" command, the user will need to provide the file name he/she wishes to transfer. In this test, we will be using binary files (markingguide.doc) for testing purposes. According to the test output of the client program, the binary file is successfully transferred from client to server. In order to check if the newly created binary file in the server is the same as the original, we purposely rename the newly created file to "`markingguide2.doc`" and check with the original "`marking guide.doc`" by using the `diff` command. As a result, there is no return value after comparing both files, which means that the content of 2 files are the same.

Command line used – client(myftp):

```
%./myftp
> put markingguide.doc
> quit
```

Test output – client (myftp):

```
> put markingguide.doc
command: put
Starting to send file to server
Sent: markingguide.doc
> quit
Bye from the client
```

Testing - comparing file

```
% diff -s markingguide.doc markingguide2.doc
Files markingguide.doc and markingguide2.doc are identical
```

- # Test 22

The purpose of this test is to test if the "put" command can successfully transfer the file to the server. In this test, we will run the server and client program on different machines. The server program will run on the ceto server and the client program will run on the virtual box. Therefore, the user will need to provide an address and port number while running the program. This test will try to transfer the text file from client to server. According to the client command line, we call the put command to sent the "big.txt" file to the server after running the client program. As a result, the file is successfully transferred to the server. In order to check the content of both files, we purposely change the newly created "big.txt" file name to "bigNew.txt". After that we run the program again and transfer the file back to the client by using the `get` command and use the `diff` command to check the content of the files. As a result, we can see that initially there is a file called big.txt after calling command `ls` (means that the file is successfully transferred). After that we use the command `mv` to change the file name. In addition, there is no return value while calling the `diff` command, which means that the content of both text files are the same.

Command line used – server (myftpd):

```
% ./myftpd 40263
```

Command line used – client (myftp):

```
% ./myftp ceto.murdoch.edu.au 40263
> put big.txt
> quit
% ./myftp ceto.murdoch.edu.au 40263
> get bigNew.txt
> quit
```

Test output – server (myftpd):

```
Daemon pid: 3237
```

Test output – client (myftp):

```
> put big.txt
command: put
Starting to send file to server
Sent: big.txt
> quit
Bye from the client
> get bigNew.txt
command: get
Server response: OK
Filesize: 4511345
Received: bigNew.txt
> quit
Bye from the client
```

Testing - comparing file

```
34315323@ceto:~/myftpd$ ls
big.txt   myftpd.c  server    stream.h toClient.txt
makefile  myftpd.o  stream.c  stream.o
34315323@ceto:~/myftpd$ mv big.txt bigNew.txt
34315323@ceto:~/myftpd$ ls
bigNew.txt  myftpd.c  server     stream.h  toClient.txt
makefile    myftpd.o  stream.c   stream.o
34315323@ceto:~/myftpd$

% diff -s big.txt bigNew.txt  → call in client program
Files big.txt and bigNew.txt are identical
```

- # Test 23

The purpose of this test is to test if the "put" command can successfully transfer the file to the server. In this test, we will run the server and client program on different machines. The server program will run on the ceto server and the client program will run on the virtual box. Therefore, the user will need to provide an address and port number while running the program. This test will try to transfer the binary file from client to server. According to the client command line, we call the put command to send the "markingguide.doc" file to the server after running the client program. As a result, the file is successfully transferred to the server. Also the `ls` function also show that the "markingguide.doc" is now in the server side. In order to ensure that the file is transferred correctly, we will use the `diff` command to check for the content of both files. According to the test for comparing files, we first change the name of the binary file to "markingguideNew.doc". After that we run the client program to get back the binary file and check the content of both files using the `diff` command. As a result, there is no return value while calling the `diff` command, which means that the content of both text files are the same.

Command line used – server (myftpd):

```
% ./myftpd 40263
```

Command line used – client (myftp):

```
% ./myftp ceto.murdoch.edu.au 40263
> put markingguide.doc
> quit
% ./myftp ceto.murdoch.edu.au 40263
> get markingguideNew.doc
> quit
```

Test output – server (myftpd):

```
Daemon pid: 3225
```

Test output – client (myftp):

```
> put markingguide.doc
command: put
Starting to send file to server
Sent: markingguide.doc
> quit
Bye from the client
> get markingguideNew.doc
command: get
Server response: OK
Filesize: 150016
Received: markingguideNew.doc
> quit
Bye from the client
```

Testing - comparing file

```
34315323@ceto:~/myftpd$ ls
big.txt   myftpd.c  server    stream.h toClient.txt
makefile  markingguide.doc   myftpd.o  stream.c  stream.o
34315323@ceto:~/myftpd$ mv markingguide.doc markingguideNew.doc
```

```
34315323@ceto:~/myftpd$ ls
bigNew.txt  myftpd.c  server     stream.h  toClient.txt
makefile    markingguideNew.doc  myftpd.o  stream.c      stream.o
34315323@ceto:~/myftpd$

% diff -s markingguide.doc markingguideNew.doc  → call in client program
Files markingguide.doc and markingguideNew.doc are identical
```

- # Test 24

The purpose of this test is to test if the "get" command can successfully transfer the file from the server to client. When calling the command, the user needs to provide the file name of the file he/she wishes to transfer. In this test we will run both server and client programs on the same machines. The file used in this test is the text file "big.txt". According to the test output, the text file is successfully transferred from the server to client. In order to ensure that the text file is correctly transferred, we purposely change the name of the newly created text file in the client side (change from big.txt to bigNew.txt). After that, we will retrieve both the original and newly created text files and check with the `diff` command. As a result, there is no return value while calling the `diff` command, which means that the content of both text files are the same.

Command line used – client(myftp):

```
%./myftp
> get big.txt
> quit
```

Test output – client (myftp):

```
> get big.txt
command: get
Server response: OK
Filesize: 4511345
Received: big.txt
> quit
Bye from the client
```

Testing - comparing file

```
% diff -s big.txt bigNew.txt
Files big.txt and bigNew.txt are identical
```

- # Test 25

The purpose of this test is to test if the “get” command can successfully transfer the file from the server to client. When calling the command, the user needs to provide the file name of the file he/she wishes to transfer. In this test we will run both server and client programs on the same machines. The file used in this test is the binary file “markingguide.doc”. According to the test output, the binary file can successfully be transferred from the server to the client. Similarly, to ensure that the text file is correctly transferred, we purposely change the name of the newly created binary file in the client side (change from markingguide.doc to markingguideNew.doc). After that, we will retrieve both the original and newly created binary files and check with the `diff` command. As a result, there is no return value while calling the `diff` command, which means that the content of both text files are the same.

Command line used – client(myftp):

```
%./myftp
> get markingguide.doc
> quit
```

Test output – client (myftp):

```
> get markingguide.doc
command: get
Server response: OK
Filesize: 150016
Received: markingguide.doc
> quit
Bye from the client
```

Testing - comparing file

```
% diff -s markingguide.doc markingguideNew.doc
Files markingguide.doc and markingguideNew.doc are identical
```

- # Test 26

The purpose of this test is to test if the “get” command can successfully transfer the file from the server to client. When calling the command, the user needs to provide the file name of the file he/she wishes to transfer. In this test, we will run the server and client programs on different machines. The server program will run on the ceto server and the client program will run on the virtual box. Therefore, the user will need to provide an address and port number while running the program. In this test, we will first get the text file “big.txt” followed by a binary file called “markingguide.doc”. This is also to check if the program can transfer the file continuously. According to the test output, both text file and binary file can successfully be transferred from server to client. Similarly, we deliberately change the file name of the text file and binary file for checking the file content. This is to ensure that the files are correctly transferred. As a result, there is no return value while calling the `diff` command, which means that the content of the original files and newly created files are the same.

Command line used – server (myftpd):

```
% ./myftpd 40263
```

Command line used – client (myftp):

```
% ./myftp ceto.murdoch.edu.au 40263
> get big.txt
> get markingguide.doc
> quit
% ls
```

Test output – server (myftpd):

```
Daemon pid: 3225
```

Test output – client (myftp):

```
> get big.txt
command: get
Server response: OK
Filesize: 4511345
Received: big.txt
> get markingguide.doc
command: get
Server response: OK
Filesize: 150016
Received: markingguide.doc
> quit
```

```
tee@tee-VirtualBox:~/myftp$ ls
big.txt  myftp makefile  markingguide.doc  client.c  client.o  stream.c
stream.h  stream.o  token.c  token.h  token.o
tee@tee-VirtualBox:~/myftp$
```

Testing - comparing file

```
% diff -s big.txt bigNew.txt
Files big.txt and bigNew.txt are identical
% diff -s markingguide.doc markingguideNew.doc  → call in client program
Files markingguide.doc and markingguideNew.doc are identical
```

- # Test 27

The purpose of this test is to test if the user enters an invalid command while running the program. The program should be able to display an error message to remind the user instead of crashing the program. In this test, we will run both client and server programs on the same machine. To conduct the test, we will deliberately key in some invalid commands such as “sent”, “phd” and “dlr”. According to the test output, the program is able to recognise the invalid command and display the error message. In addition, the program can continue running after the invalid command instead of crashing.

Command line used – client(myftp):

```
%./myftp
> sent big.txt
> phd
> dlr
> quit
```

Test output – client (myftp):

```
> sent big.txt
Invalid command
> phd
Invalid command
> dlr
Invalid command
> quit
Bye from the client
```

- # Test 28

The purpose of this test is to test if the user enters an non-existence file name while calling the `get` and `put` command. Instead of crashing the program, the program should display an error message to inform the user about the non-existence file. In this test, we will deliberately enter some non-existence file names while calling the `get` and `put` command. According to the test output, the program successfully displays the error message when we try to put and get a non-existence file called hello.c.

Command line used – client(myftp):

```
%./myftp
> put hello.c
> get hello.c
```

Test output – client (myftp):

```
> put hello.c
Error in opening the file: hello.c
> get hello.c
Requested file not found - put
> quit
Bye from the client
```

Files in Server (myftpd):

```
$ ls
server          sftp_log.txt
```

Files in client(myftp):

```
$ ls
bigbig.txt      markingguide.doc
big.txt         small.txt
client
```

- # Test 29

The purpose of this test is to test if the program is able to transfer the "bigbig.txt" file. In this test, we will run both client and server programs on 2 different machines. The servers will run on the ceto server. In the first part of this test, we first sent the "bigbig.txt" file to the server. After that, we use the `mv` command to change the file name of "bigbig.txt" to "bigbig2.txt". The purpose of doing this is to test the `get` command and the content of the file. After that, we will run the client program again and transfer back the "bigbig2.txt' file from server to client by using the `get` command. According to the test output, the program can successfully transfer the "bigbig.txt" and "bigbig2.txt" file by using the `get` and `put` command. To ensure that the text file is transferred correctly, we will execute the `diff` command to compare the "bigbig.txt" and "bigbig2.txt" in the client side (because the "bigbig2.txt" has sent to client). As a result, there is no return value from executing the `diff` command. This means that the content of both files are identical.

Command line used – server (myftpd):

```
% ./myftpd 40263
```

Command line used – client (myftp):

```
% ./myftp ceto.murdoch.edu.au 40263
> put bigbig.txt
> quit
% ./myftp ceto.murdoch.edu.au 40263
> get bigbig2.txt
> quit
```

Test output – server (myftpd):

```
Daemon pid: 3225
```

Test output – client (myftp):

```
./myftp ceto.murdoch.edu.au
> put bigbig.txt
command: put
Starting to send file to server
Sent: bigbig.txt
> quit
Bye from the client

./myftp ceto.murdoch.edu.au
```

```
> get bigbig2.txt
command: get
Server response: OK
Filesize: 13534035
Received: bigbig2.txt
> quit
Bye from the client
```

Testing - comparing file

```
34315323@ceto:~/myftpd$ mv bigbig.txt bigbig2.txt

% diff -s bigbig.txt bigbig2.txt  → call in client program
Files bigbig.txt and bigbig2.txt are identical
```

• Test 30

The purpose of this test is to check that the program is able to catch when user do not provide path or file name when it is expected in get, put, cd and lcd commands.
Since the program is made to catch the issue before sending any instructions to the server. This test could be done either on the same machine or in multiple machines.
Test the all four commands by using the command without giving e.g. “get”, and also with additional spaces e.g. “get “.

Command line used – client (myftp):

```
$ ./myftp
> put
> get
> cd
> lcd
> put
> get
> cd
> lcd
```

Test output - client (myftp):

```
$ ./myftp
> put
command: put
Error no path given
> get
command: get
Error no path given
> cd
command: cd
Error no path given
> lcd
command: lcd
Failed to change directory
: Bad address
> put
```

```
command: put
Error no path given
> get
command: get
Error no path given
> cd
command: cd
Error no path given
> lcd
command: lcd
Failed to change directory
: Bad address
>
```

Source code listing

server.c

```c
/**
 * @file server.c
 * @brief The main program of SFTP server. Responsible for creating connection
to client and creating daemon process to serve each connected client
 * @version 1.0
 * @date 2022-07-25
 *
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <signal.h>
#include "logger/logger.h"
#include "serverFunction/sFunctionHelper.h"

//#define BUFSIZE 512
#define SERV_TCP_PORT 40263 // given default port number

// /* opcodes */
// #define PUT      'A'
// #define GET      'G'
// #define RESPOND  'B'
// #define PWD      'E'
// #define DIR_OPC  'F'
// #define CD       'D'

void serve_a_client(int sd)
{
    char opcode;

    // read opcode from client first
    while (read(sd, (char *)&opcode, 1) > 0)
    {
```

```
        if (opcode == PUT)
        {
            Logger("Opcode for put received\n");
            put_func(sd);
        }
        else if (opcode == GET)
        {
            Logger("Opcode for get received\n");
            get_func(sd);
        }
        else if (opcode == PWD)
        {
            Logger("Opcode for pwd received\n");
            pwd_func(sd);
        }
        else if (opcode == DIR_OPC)
        {
            Logger("Opcode for dir received\n");
            dir_func(sd);
        }
        else if (opcode == CD)
        {
            Logger("Opcode for cd received\n");
            cd_func(sd);
        }
        else
        {
            Logger("Invalid opcode received\n");
        }
    } // end while

    return;
}

void claim_children()
{
    pid_t pid = 1;
    while (pid > 0)
    { /* claim as many zombies as we can */
        pid = waitpid(0, (int *)0, WNOHANG);
    }
}

void daemon_init(void)
{
    pid_t pid;
    struct sigaction act;
```

```
    if ((pid = fork()) < 0)
    {
        perror("fork");
        exit(1);
    }
    else if (pid > 0)
    {
        /* parent */
        printf("Daemon pid: %d\n", pid);
        exit(0);
    }
    else
    {
        /* child */
        setsid(); /* become session leader */
        umask(0); /* clear file mode creation mask */

        /* catch SIGCHLD to remove zombies from system */
        act.sa_handler = claim_children; /* use reliable signal */
        sigemptyset(&act.sa_mask);       /* not to block other signals */
        act.sa_flags = SA_NOCLDSTOP;     /* not catch stopped children */
        sigaction(SIGCHLD, (struct sigaction *)&act, (struct sigaction *)0);
    }
}

int main(int argc, char *argv[])
{
    int nsd, sd, n;
    pid_t pid;
    unsigned short port; /* server listening port */
    socklen_t cli_addrlen;
    struct sockaddr_in ser_addr;
    struct sockaddr_in cli_addr;

    // remove log file
    remove(logfile);

    // get port number
    if (argc == 1)
    {
        port = SERV_TCP_PORT;
    }
    else if (argc == 2)
    {
        // use user given port
        n = atoi(argv[1]);
        if (n >= 1024 && n < 65536)
        {
```

```
            port = n;
        }
        else
        {
            printf("Error: port number must be between 1024 and 65535\n");
            exit(1);
        }
    }
    else
    {
        printf("Usage: %s [ server listening port ]\n", argv[0]);
        exit(1);
    }

    // set to current directory
    char dir[256] = ".";
    if (chdir(dir) == -1)
    {
        printf("Failed to set initial directory to: %s\n", dir);
        exit(1);
    }

    /* make the server a daemon. */
    daemon_init();

    /* set up listening socket sd */
    if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("server: socket");
        exit(1);
    }

    /* build server Internet socket address */
    bzero((char *)&ser_addr, sizeof(ser_addr));
    ser_addr.sin_family = AF_INET;                // address family
    ser_addr.sin_port = htons(port);              // network ordered port
number
    ser_addr.sin_addr.s_addr = htonl(INADDR_ANY); // any interface

    /* bind server address to socket sd */
    if (bind(sd, (struct sockaddr *)&ser_addr, sizeof(ser_addr)) < 0)
    {
        perror("server bind");
        exit(1);
    }

    /* become a listening socket */
    listen(sd, 5);
```

```
    while (1)
    {
        /* wait to accept a client request for connection */
        cli_addrlen = sizeof(cli_addr);
        nsd = accept(sd, (struct sockaddr *)&cli_addr, &cli_addrlen);
        if (nsd < 0)
        {
            if (errno == EINTR) /* if interrupted by SIGCHLD */
                continue;
            perror("server: accpet");
            exit(1);
        }

        /* create a child process to handle this client */
        if ((pid = fork()) < 0)
        {
            perror("fork");
            exit(1);
        }
        else if (pid > 0)
        {
            close(nsd);
            continue; /* parent to wait for next client */
        }
        else
        {
            /* now in child, serve the current client */
            close(sd);
            InitializeLogger();
            serve_a_client(nsd);
            exit(0);
        }
    }
}
```

client.c

```
/**
 * @file client.c
 * @brief The main program of SFTP client. Responsible for connecting to
server and serving the user by calling other functions
 * @version 1.0
 * @date 2022-07-25
 *
 * @version 1.1
 * @date 2022-07-27
```

```
 * Added checks for cd, put & get for path
 * Token[] will be initialized as NULL
 * Token[1] will be reset to NULL at end of each while
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <dirent.h>
#include <fcntl.h>
#include <netdb.h>

#include "stream/stream.h"
#include "clientFunction/token.h"
#include "clientFunction/cFunctionHelper.h"

int main(int argc, char *argv[])
{
     int sd, nr, n;
     char buf[128], host[60];
     unsigned short port;
     struct sockaddr_in ser_addr;
     struct hostent *hp;
     char *token[2] = {NULL};

     /* get server host name and port number */
     if (argc == 1)
     {
          /* assume server running on the local host and on default port */
          strcpy(host, "localhost");
          port = SERV_TCP_PORT;
     }
     else if (argc == 2)
     { /* use the given host name */
          strcpy(host, argv[1]);
          port = SERV_TCP_PORT;
     }
     else if (argc == 3)
     { /* use given host and port for server */
          strcpy(host, argv[1]);
          n = atoi(argv[2]);

          if (n >= 1024 && n < 65536)
```

```
          {
               port = n;
          }
          else
          {
               printf("Error: server port number must be between 1024 and
65535\n");
               exit(1);
          }
     }
     else
     {
          printf("Usage: %s [ <server host name> [ <server listening port> ]
]\n", argv[0]);
          exit(1);
     }

     /* get host address, & build a server socket address */
     bzero((char *)&ser_addr, sizeof(ser_addr));
     ser_addr.sin_family = AF_INET;
     ser_addr.sin_port = htons(port);
     if ((hp = gethostbyname(host)) == NULL)
     {
          printf("host %s not found\n", host);
          exit(1);
     }
     ser_addr.sin_addr.s_addr = *(u_long *)hp->h_addr;

     /* create TCP socket & connect socket to server address */
     sd = socket(PF_INET, SOCK_STREAM, 0);
     if (connect(sd, (struct sockaddr *)&ser_addr, sizeof(ser_addr)) < 0)
     {
          perror("client connect");
          exit(1);
     }

     while (1)
     {
          printf("> "); /* display prompt */

          /* read user input and tokenise */
          fgets(buf, sizeof(buf), stdin);
          nr = strlen(buf);
          // replace last char
          if (buf[nr - 1] == '\n')
          {
               buf[nr - 1] = '\0';
               nr--;
```

```
        }

        // check if quit
        if (strcmp(buf, "quit") == 0)
        {
             printf("Bye from the client\n");
             exit(0);
        }

        // split command and filename
        tokenise(buf, token);

        printf("command: %s\n", token[0]);

        // execute corresponding function
        if (strcmp(token[0], "put") == 0)
        {
             if(token[1] == NULL)
             {
                  printf("Error no path given\n");
             }
             else
             {
                  put(sd, token[1]);
             }
        }
        else if (strcmp(token[0], "get") == 0)
        {
             if(token[1] == NULL)
             {
                  printf("Error no path given\n");
             }
             else
             {
                  get(sd, token[1]);
             }
        }
        else if (strcmp(token[0], "pwd") == 0)
        {
             pwd(sd);
        }
        else if (strcmp(token[0], "lpwd") == 0)
        {
             lpwd(sd);
        }
        else if (strcmp(token[0], "ldir") == 0)
        {
             ldir(token[1]);
```

```
        }
        else if (strcmp(token[0], "dir") == 0)
        {
            dir(sd);
        }
        else if (strcmp(token[0], "cd") == 0)
        {
            if(token[1] == NULL)
            {
                printf("Error no path given\n");
            }
            else
            {
                cd(sd, token[1]);
            }
        }
        else if (strcmp(token[0], "lcd") == 0)
        {
            lcd(token[1]);
        }
        else if (strcmp(buf, "help") == 0)
        {
            DisplayMenu();
        }
        else
        {
            printf("Invalid command\n");
        }

        token[1] = NULL;
    }
}
```

stream.h

```
/**
 * @file stream.h
 * @brief Holds functions that help to send or recieve data between client and
server
 * @version 1.0
 * @date 2022-07-25
 *
 *
 */

#include  <unistd.h>
#include  <sys/types.h>
```

```c
#include  <netinet/in.h> /* struct sockaddr_in, htons(), htonl(), */

#define MAX_BLOCK_SIZE 512
//#define MAX_BLOCK_SIZE (1024*5)    /* maximum size of any piece of */
                                   /* data that can be sent by client */

/*
 * purpose:  read a stream of bytes from "fd" to "buf".
 * pre:      1) size of buf bufsize >= MAX_BLOCK_SIZE,
 * post:     1) buf contains the byte stream;
 *           2) return value > 0   : number ofbytes read
 *                           = 0   : connection closed
 *                           = -1  : read error
 *                           = -2  : protocol error
 *                           = -3  : buffer too small
 */
/**
 * @brief read a stream of bytes from "fd" to "buf".
 *
 * @param fd socket descriptor
 * @param buf buffer to hold data
 * @param bufsize size of buffer
 * @return int The number of bytes read
 */
int ReadN(int fd, char *buf, int bufsize);



/*
 * purpose:  write "nbytes" bytes from "buf" to "fd".
 * pre:      1) nbytes <= MAX_BLOCK_SIZE,
 * post:     1) nbytes bytes from buf written to fd;
 *           2) return value = nbytes : number ofbytes written
 *                           = -3     : too many bytes to send
 *                           otherwise: write error
 */

/**
 * @brief write "nbytes" bytes from "buf" to "fd".
 *
 * @param fd socket descriptor
 * @param buf buffer of data to send
 * @param nbytes number of bytes in buf
 * @return int number of bytes written
 */
int WriteN(int fd, char *buf, int nbytes);

/**
 * @brief Writes opcode, filename's length and filename to sd
```

```
 *
 * @param sd socket descriptor
 * @param opcode the opcode
 * @param filename_len the length of the file name
 * @param buf the filename
 * @return int  bytes of filename written
 */
int WritePath(int sd, char opcode, int filename_len, char *buf);
```

stream.c

```
#include "stream.h"

int ReadN(int fd, char *buf, int nbytes)
{
    int n, nr;

    for (n = 0; n < nbytes; n += nr)
    {
        if ((nr = read(fd, buf + n, nbytes - n)) <= 0)
            return (nr); /* error in reading */
    }
    return (n);
}

int WriteN(int sd, char *buf, int bufsize)
{
    int n, nw;

    for (n = 0; n < bufsize; n += nw)
    {
        if ((nw = write(sd, buf + n, bufsize - n)) <= 0)
            return (nw); /* write error */
    }
    return n;
}

int WritePath(int sd, char opcode, int filepath_len, char *buf)
{
    int n, nw;

    // send opcode
    write(sd, (char *)&opcode, 1);

    // send 2-bytes int - length of filename
    short data_size = filepath_len; /* short must be two bytes long */
    data_size = htons(data_size);   /* convert to network byte order */
```

```
    write(sd, &data_size, 2);

    // send filename
    for (n = 0; n < filepath_len; n += nw)
    {
        if ((nw = write(sd, buf + n, filepath_len - n)) <= 0)
            return (nw); /* write error */
    }

    return n;
}
```

sFunctionHelper.h

```
/**
 * @file sFunctionHelper.h
 * @brief Holds functions related to server's SFTP functions such as put and
get files
 * @version 1.0
 * @date 2022-07-25
 *
 *
 */

#include <string.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

#include "../stream/stream.h"

#define BUFSIZE 512

/* opcodes */
#define PUT      'A'
#define GET      'G'
#define RESPOND  'B'
#define PWD      'E'
#define DIR_OPC  'F'
#define CD       'D'

/**
 * @brief Respond to client's put request by receiving client's file and
storing it
 *
```

```
 * @param sd Socket descriptor
 */
void put_func(int sd);

/**
 * @brief Respond to client's get request by retriving file and sending it to
client
 *
 * @param sd Socket descriptor
 */
void get_func(int sd);

/**
 * @brief Respond to client's pwd request by sending client the server's
current directory path
 *
 * @param sd Socket descriptor
 */
void pwd_func(int sd);

/**
 * @brief Respond to client's sd request by sending client the server's list
of file in current directory
 *
 * @param sd Socket descriptor
 */
void dir_func(int sd);

/**
 * @brief Respond to client's cd request by changing the current directory to
what client defined
 *
 * @param sd Socket descriptor
 */
void cd_func(int sd);
```

sFunctionHelper.c

```
#include "sFunctionHelper.h"
#include "../logger/logger.h"

void pwd_func(int sd)
{
    // log file operation
    Logger("PWD start\n");

    char current_directory[BUFSIZE];
```

```
    getcwd(current_directory, sizeof(current_directory)); // get pathname

    // respond to client
    // sent opcode to client first
    char code = (char)PWD;
    if (write(sd, (char *)&code, 1) != 1)
    {
        Logger("Error in sending opcode - pwd\n");
        return;
    }

    // sent 2-bytes of pathname size
    short data_size = strlen(current_directory); /* convert to network byte
order */
    data_size = htons(data_size);
    if (write(sd, &data_size, 2) != 2)
    {
        Logger("Error in sending pathname size - pwd\n");
        return;
    }

    // sent pathname
    if (WriteN(sd, current_directory, strlen(current_directory)) < 0)
    {
        Logger("Error in sending pathname - pwd\n");
        return;
    }

    Logger("PWD completed\n");
    return;
}

void dir_func(int sd)
{
    // log file operation
    Logger("DIR start\n");

    char dirname[256];
    DIR *dp;
    struct dirent *direntp;
    char filenames[BUFSIZE];
    int length;

    // obtain all filenames first

    strcpy(dirname, "."); // current directory

    if ((dp = opendir(dirname)) == NULL)
```

```
    {
        char *logstr;
        strcat(logstr, "Error in opening directory ");
        strcat(logstr, dirname);
        strcat(logstr, "\n");
        Logger(logstr);
        exit(1);
    }

    // store all filenames
    while ((direntp = readdir(dp)) != NULL)
    {
        strcat(filenames, direntp->d_name);
        strcat(filenames, "\n");
    }

    length = strlen(filenames);
    filenames[length - 1] = '\0';
    closedir(dp);

    // respond client
    // sent opcode to client
    char code = (char)DIR_OPC;
    if (write(sd, (char *)&code, 1) != 1)
    {
        Logger("Error in sending opcode - dir\n");
        return;
    }

    // sent 4-bytes length of filenames array
    int data = htonl(strlen(filenames));
    if (write(sd, (char *)&data, 4) != 4)
    {
        Logger("Error in sending length of array - dir\n");
        return;
    }

    // sent content - all filenames
    if (WriteN(sd, filenames, strlen(filenames)) < 0)
    {
        Logger("Error in sending content - dir\n");
        return;
    }

    Logger("DIR completed\n");
}

void cd_func(int sd)
```

```
{
    // log file operation
    Logger("CD start\n");

    int length, check;
    char acknowledgment_code;

    // read client's message
    // read length of new directory name
    short data_size;
    if (read(sd, (char *)&data_size, 2) != 2)
    {
        Logger("Error in reading length of directory name - cd\n");
        return;
    }
    length = (int)ntohs(data_size); /* convert to host byte order */

    char directory[length + 1];

    // read name of new directory
    if (ReadN(sd, directory, length) == -1)
    {
        Logger("Error in reading directory name - cd\n");
        return;
    }

    directory[length] = '\0';

    // change directory
    check = chdir(directory);
    // Create new log in new directory
    InitializeLogger();

    if (check == 0)
    {
        acknowledgment_code = '0';
    }
    else
    {
        acknowledgment_code = '1';
    }

    // respond to client
    // sent opcode to client
    char code = (char)CD;
    if (write(sd, (char *)&code, 1) != 1)
    {
        Logger("Error in sending opcode - cd\n");
```

```
        return;
    }

    // sent acknowledment code
    if (write(sd, (char *)&acknowledgment_code, 1) != 1)
    {
        Logger("Error in sending opacknowledgment code - cd\n");
        return;
    }

    Logger("CD completed\n");
}

void put_func(int sd)
{
    // log file operation
    Logger("PUT start\n");

    int filename_len, filesize;
    char acknowledgment_code;
    char opcode;
    int fd, nr, nw, n;

    // read length of filename
    short data_size;
    if (read(sd, (char *)&data_size, 2) != 2)
    {
        Logger("Error in reading the length of filename - put\n");
        return;
    }
    filename_len = (int)ntohs(data_size); // convert to host byte order

    char filename[filename_len + 1];

    // read filename
    if (ReadN(sd, filename, filename_len) == -1)
    {
        Logger("Error in reading filename - put\n");
        return;
    }
    filename[filename_len] = '\0';

    // create and open the a file to prepare for recieving file from client
    acknowledgment_code = '0';
    if ((fd = open(filename, O_RDONLY)) != -1)
    {
        if(errno == EINVAL)
        {
```

```
            acknowledgment_code = '2';
            Logger("Invalid file name (EINVAL)\n");
        }
        else if(errno == EACCES)
        {
            acknowledgment_code = '1';
            Logger("Access denied or file do not exist (EACCES)\n");
        }
        else
        {
            acknowledgment_code = '3';
            Logger("The server cannot accept the file due to other
reasons\n");
        }
    }
    else if ((fd = open(filename, O_WRONLY | O_CREAT, 0766)) == -1)
    {
        if(errno == EACCES)
        {
            acknowledgment_code = '1';
            Logger("Server unable to create file due to insufficient
permission (EACCES)\n");
        }
        else if(errno == EINVAL)
        {
            acknowledgment_code = '2';
            Logger("Server unable to create file due to file name being
invalid (EINVAL)\n");
        }
        else
        {
            acknowledgment_code = '3';
            Logger("Server unable to create file due to other reasons\n");
        }
    }

    // section b
    // Respond client with require message
    // sent opcode
    char code = (char)PUT;
    if (write(sd, (char *)&code, 1) != 1)
    {
        Logger("Error in sending opcode to client - put\n");
        return;
    }

    // send acknowledgment code
    if (write(sd, (char *)&acknowledgment_code, 1) != 1)
```

```
    {
        Logger("Error in sending acknowledgment code - put\n");
        return;
    }

    if (acknowledgment_code != '0')
    {
        Logger("Error in performing command put\n");
        return;
    }

    // After section c
    // Read client's respond

    // read opcode
    if (read(sd, (char *)&opcode, 1) != 1)
    {
        Logger("Error reading opcode - put\n");
        return;
    }

    // check client respond - opcode
    if (opcode == RESPOND)
    {
        // read filesize
        if (read(sd, (char *)&filesize, 4) != 4)
        {
            Logger("Error in reading file size - put\n");
            return;
        }
        filesize = (int)ntohl(filesize); // convert to host byte order
    }

    // Get number of loops and the remainder bytes needed to get file from
client
    int blocks = filesize / (int)BUFSIZE;
    int remainder = filesize % (int)BUFSIZE;

    // read and write file content
    char content[BUFSIZE];
    // Loops for block + 1 times for the file
    for (int i = 0; i <= blocks; i++)
    {
        // only enters on last loop to get remainding bytes that is less than
BUFSIZE
        if (i == blocks)
        {
            if ((nr = ReadN(sd, content, remainder)) <= 0)
```

```
            {
                Logger("Error in reading filename - put\n");
                close(fd);
                return;
            }
            if ((nw = write(fd, content, nr)) < nr)
            {
                Logger("Error in writing the requested file - put\n");
                close(fd);
                return;
            }
            break;
        }

        // Reads from client and write to created file only runs when
recieving BUFSIZE bytes
        if ((nr = ReadN(sd, content, BUFSIZE)) <= 0)
        {
            Logger("Error in reading filename - put\n");
            close(fd);
            return;
        }
        if ((nw = write(fd, content, nr)) < nr)
        {
            Logger("Error in writing the requested file - put\n");
            close(fd);
            return;
        }
    }

    close(fd);

    Logger("PUT completed\n");
    return;
}

void get_func(int sd)
{
    // log file operation
    Logger("GET start\n");

    int fd, nr, n, nw;
    struct stat fileInfo;
    int filesize;
    int filename_len;
    char acknowledgment_code, acknowledgment_code_from_client;
    char opcode;
```

```
    // After section a
    // read client message

    // read filename
    short data_size; // sizeof (short) must be 2
    if (read(sd, (char *)&data_size, 2) != 2)
    {
        Logger("Error in reading length of filename - get\n");
        return;
    }
    filename_len = (int)ntohs(data_size); // convert to host byte order

    char filename[filename_len + 1];

    // read filename
    if (ReadN(sd, filename, filename_len) == -1)
    {
        Logger("Error in reading filename - get\n");
        return;
    }
    filename[filename_len] = '\0';

    acknowledgment_code = '0';

    // check file
    // if error encountered
    if ((fd = open(filename, O_RDONLY)) == -1)
    {
        if(errno == EINVAL)
        {
            acknowledgment_code = '2';
            Logger("Invalid file name (EINVAL)\n");
        }
        else if(errno == EACCES)
        {
            acknowledgment_code = '1';
            Logger("Access denied or file do not exist (EACCES)\n");
        }
        else
        {
            acknowledgment_code = '3';
            Logger("The server cannot send file due to other reasons\n");
        }
    }

    // send error opcode and ackowledgment code
    opcode = (char)GET;
    if (write(sd, (char *)&opcode, 1) != 1)
```

```
    {
        Logger("Error in sending respond - get\n");
        return;
    }
    if (write(sd, (char *)&acknowledgment_code, 1) != 1)
    {
        Logger("Error in sending acknowledgment - get\n");
        return;
    }

    // read message from client
    if (read(sd, (char *)&opcode, 1) != 1)
    {
        Logger("Error reading opcode - get\n");
        return;
    }
    if (read(sd, (char *)&acknowledgment_code_from_client, 1) != 1)
    {
        Logger("Error reading acknowledgment code from client - get\n");
        return;
    }


    // Checks that opcode and ackcode indicates continue from client
    //Stop function if there is error
    if(opcode == GET)
    {
        if(acknowledgment_code_from_client == '5')
        {
            Logger("Client unable to receive file due to insufficient
permission\n");
            return;
        }
        else if(acknowledgment_code_from_client == '6')
        {
            Logger("Client unable to receive file due to invalid file
name\n");
            return;
        }
        else if(acknowledgment_code_from_client == '7')
        {
            Logger("Client unable to receive file due to it being a
directory\n");
            return;
        }
        else if(acknowledgment_code_from_client == '8')
        {
            Logger("Client unable to receive file due to other reasons\n");
            return;
```

```
        }
    }
    else
    {
        Logger("Unexpected opcode received\n");
        return;
    }
    if(acknowledgment_code != '0' || acknowledgment_code_from_client != '0')
    {
        Logger("Get ended with error\n");
        return;
    }

    // if no error encountered
    // sent opcode
    char code = (char)RESPOND;
    if (write(sd, (char *)&code, 1) != 1)
    {
        Logger("Error in sending respond - get\n");
        return;
    }
    // get file size
    if (fstat(fd, &fileInfo) < 0)
    {
        Logger("Error in fstat - get\n");
        return;
    }
    filesize = (int)fileInfo.st_size; // size of the file

    // reset file pointer
    lseek(fd, 0, SEEK_SET);

    // sent file size
    filesize = htonl(filesize); /* convert to network byte order */
    if (write(sd, &filesize, 4) != 4)
    {
        Logger("Error in sending file size - get\n");
        return;
    }

    // read and write file content
    char buf[BUFSIZE];

    Logger("Sending file to client\n");
    // read content first
    while ((nr = read(fd, buf, BUFSIZE)) > 0)
    {
        // write to server however many bytes was read
```

```
        if (WriteN(sd, buf, nr) == -1)
        {
            Logger("Error in sending file content - get\n");
            return;
        }
    }

    Logger("GET completed\n");
    return;
}
```

cFunctionHelper.h

```
/**
 * @file cFunctionHelper.h
 * @brief Holds client's SFTP related functions such as get and put files.
 * @version 1.0
 * @date 2022-07-25
 *
 *
 */

#include <dirent.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>

#include "../stream/stream.h"


#define BUFSIZE 512
#define SERV_TCP_PORT   40263  // given default port number

/* opcodes */
#define PUT         'A'
#define RESPOND     'B'
#define GET         'G'
#define PWD         'E'
#define DIR_OPC     'F'
#define CD          'D'


/**
```

```
 * @brief prints out SFTP command menu in terminal
 *
 */
void DisplayMenu();

/**
 * @brief Get and display the current directory of server
 *
 * @param sd    socket descriptor
 */
void pwd(int sd);

/**
 * @brief Display the current local directory
 *
 * @param sd    socket descriptor
 */
void lpwd(int sd);

/**
 * @brief Get and display directory listing of server
 *
 * @param sd    socket descriptor
 */
void dir(int sd);

/**
 * @brief Display local directory listing
 *
 * @param token directory path input
 */
void ldir(char* token);

/**
 * @brief Change current directory of server
 *
 * @param sd    socket descriptor
 * @param token directory path input
 */
void cd(int sd, char* token);

/**
 * @brief Change current local directory
 *
 * @param token directory path input
 */
void lcd(char* token);
```

```c
/**
 * @brief Send a file to server by filename
 *
 * @param sd    socket descriptor
 * @param filename  filename of file to send to server
 */
void put(int sd, char* filename);

/**
 * @brief Retrieve a file from server by filename
 *
 * @param sd    socket descriptor
 * @param filename  filename of file to be retrieved from server
 */
void get(int sd, char* filename);
```

cFunctionHelper.c

```c
#include "cFunctionHelper.h"

void DisplayMenu()
{
    printf("---Command List---\n");

    printf("put <filename>    - Sends file to server\n");
    printf("get <filename>    - Retrieve file from server\n");

    printf("pwd     - Display current directory path on server\n");
    printf("lpwd    - Display current local directory path\n");

    printf("dir    - Display current directory listing on server\n");
    printf("ldir    - Display current local directory listing\n");

    printf("cd <path>     - Change current directory on server\n");
    printf("lcd <path>     - Change current local directory\n");

    printf("help    - Display functions menu\n");

    printf("quit    - quit program\n");

    printf("---End List---\n");
    return;
}

void pwd(int sd)
{
    int msg_len;
```

```
    char opcode;

    // sent opcode to server
    char code = (char)PWD;
    if (write(sd, (char *)&code, 1) != 1)
    {
        printf("Error in sending opcode - pwd\n");
        return;
    }

    // read server's respond
    if (read(sd, (char *)&opcode, 1) != 1)
    {
        printf("Error reading opcode - pwd\n");
        return;
    }

    if (opcode == PWD)
    {
        // read length of message
        short data_size;
        if (read(sd, (char *)&data_size, 2) != 2)
        {
            printf("Error in reading length of message - pwd\n");
            return;
        }
        msg_len = (int)ntohs(data_size); // convert to host byte order
    }

    char message[msg_len + 1];

    // read message content
    if (ReadN(sd, message, msg_len) < 0)
    {
        printf("Error in reading directory - pwd\n");
        return;
    }

    message[msg_len] = '\0';
    printf("%s\n", message);
    return;
}

void lpwd(int sd)
{
    char current_directory[BUFSIZE];
    if (getcwd(current_directory, sizeof(current_directory)) == NULL) // get
pathname
```

```
    {
        perror("Failed to get current directory\n");
        return;
    }
    else
    {
        printf("%s\n", current_directory);
        return;
    }
}

void dir(int sd)
{
    char opcode;
    int msg_len;

    // send 1 byte opcode to server to inform the operation
    char code = (char)DIR_OPC;
    if (write(sd, (char *)&code, 1) != 1)
    {
        printf("Error in sending opcode - dir\n");
        return;
    }

    // read server's respond
    if (read(sd, (char *)&opcode, 1) != 1)
    {
        printf("Error reading opcode - dir\n");
        return;
    }

    if (opcode == DIR_OPC)
    {
        // read length of message
        int data;
        if (read(sd, (char *)&data, 4) != 4)
        {
            printf("Error in reading length of message - pwd\n");
            return;
        }
        msg_len = (int)ntohl(data); // convert to host byte order
    }

    char message[msg_len + 1];
    // read message content
    if (ReadN(sd, message, msg_len) < 0)
    {
        printf("Error in reading directory - dir\n");
```

```
        return;
    }

    message[msg_len] = '\0';
    printf("%s\n", message);
    return;
}

void ldir(char *token)
{
    DIR *dp;
    struct dirent *direntp;

    if (token == NULL)
    {
        token = "."; // current directory
    }

    // open the client's current dicectory
    if ((dp = opendir(token)) == NULL)
    {
        perror("Error in opening client's current directory\n");
        return;
    }

    // read and print file names
    while ((direntp = readdir(dp)) != NULL)
    {
        printf("%s\n", direntp->d_name);
    }
    closedir(dp);
    return;
}

void cd(int sd, char *token)
{
    char opcode;
    char acknowlegment_code;
    int n, nw;

    // sent required message to server
    if (nw = WritePath(sd, CD, strlen(token), token) <= 0)
    {
        perror("Error in sending message to server - cd\n");
        return;
    }

    // read server's respond
```

```
    if (read(sd, (char *)&opcode, 1) != 1)
    {
        perror("Error reading opcode - cd\n");
        return;
    }

    if (opcode == CD)
    {
        // read acklodgment code
        if (read(sd, (char *)&acknowlegment_code, 1) != 1)
        {
            perror("Error reading ackowledgment code - cd\n");
            return;
        }
    }

    // if get 1 - means error
    // else no issue
    if (acknowlegment_code == '1')
    {
        perror("Server failed to change the working directory\n");
        return;
    }
    else if (acknowlegment_code == '0')
    {
        printf("Server directory changed\n");
        return;
    }
    else
    {
        perror("Unknown acknowledgement code\n");
        return;
    }
}

void lcd(char *token)
{
    if (chdir(token) == -1)
    {
        perror("Failed to change directory\n");
        return;
    }
    return;
}

void put(int sd, char *filename)
{
    int fd, nw, n, nr;
```

```c
    struct stat fileInfo; // obtain file size
    int filesize;
    int filename_len = strlen(filename);
    char opcode;
    char acknowledgment_code = '0';

    // open requested file
    if ((fd = open(filename, O_RDONLY)) == -1)
    {
        perror("Error in opening the file: \n");
        return;
    }

    /* reset file pointer */
    lseek(fd, 0, SEEK_SET);

    // section a
    // client sent message to server
    if (nw = WritePath(sd, PUT, filename_len, filename) <= 0)
    {
        perror("Error in sending message to server - put\n");
        return;
    }

    // now read server's respond to see if filename has any error
    // read opcode
    if (read(sd, (char *)&opcode, 1) != 1)
    {
        perror("Error reading opcode - put\n");
        return;
    }

    // check opcode
    if (opcode != PUT)
    {
        printf("Unexpected opcode received from server - put\n");
        exit(1);
    }

    // read acknowledgment code
    if (read(sd, (char *)&acknowledgment_code, 1) != 1)
    {
        perror("Error in reading acknowledgment code - put\n");
        return;
    }

    // check acknowledgment code received
    if (acknowledgment_code == '1')
```

```
    {
        printf("Server unable to receive file due to insufficient permission
(EACCES)\n");
        return; // end function
    }
    else if (acknowledgment_code == '2')
    {
        printf("Server unable to receive file due to file name being invalid
(EINVAL)\n");
        return; // end function
    }
    else if(acknowledgment_code == '3')
    {
        printf("Server unable to receive file due to other reasons\n");
        return; // end function
    }
    else if(acknowledgment_code != '0')
    {
        printf("Unknown ack code received\n");
        return;
    }

    // section c
    // if acknowledgement code = 0
    // sent respond

    // sent opcode
    char code = (char)RESPOND;
    if (write(sd, (char *)&code, 1) != 1)
    {
        perror("Error in sending opcode - put\n");
        return;
    }

    // get file size
    if (fstat(fd, &fileInfo) < 0)
    {
        perror("Error in fstat\n");
        return;
    }
    filesize = (int)fileInfo.st_size; // size of the file

    // send 4-bytes int - file size
    int data_size = htonl(filesize); // convert to network byte order
    if (write(sd, &data_size, 4) != 4)
    {
        perror("Error sending filesize - put\n");
        return;
```

```
    }

    // sent content of the file
    char buf[BUFSIZE];

    printf("Starting to send file to server\n");
    // write to server
    while ((nr = read(fd, buf, BUFSIZE)) > 0)
    {
        // write to server
        if (WriteN(sd, buf, nr) == -1)
        {
            perror("Error in sending file content - get\n");
            return;
        }
    }
    printf("Sent: %s\n", filename);
    return;
}

void get(int sd, char *filename)
{
    int filename_len = strlen(filename);
    char acknowledgment_code, acknowledgment_code_to_server = '0';
    char opcode;
    int fd, nw, nr, n;
    int filesize;

    // section a
    // client send message to server to request the file
    if (nw = WritePath(sd, GET, filename_len, filename) <= 0)
    {
        perror("Error in sending message to server - get\n");
        exit(1);
    }

    // check server respond
    // get opcode from server
    if (read(sd, (char *)&opcode, 1) != 1)
    {
        perror("Error reading opcode - get\n");
        exit(1);
    }

    if (opcode == GET)
    {
        // obtain acknowledgment code from server
        if (read(sd, (char *)&acknowledgment_code, 1) != 1)
```

```
        {
            printf("Error reading acknowledgment code - get\n");
            acknowledgment_code_to_server = '1';
        }

        // check opcode & acknowledgment code
        if (acknowledgment_code == '1')
        {
            printf("Access denied or file do not exist (EACCES)\n");
            acknowledgment_code_to_server = '1';
        }
        else if(acknowledgment_code == '2')
        {
            printf("Invalid file name (EINVAL)\n");
            acknowledgment_code_to_server = '1';
        }
        else if(acknowledgment_code == '3')
        {
            printf("The server cannot accept the file due to other
reasons\n");
            acknowledgment_code_to_server = '1';
        }
        else if(acknowledgment_code != '0')
        {
            printf("Unknown ack code received from server\n");
            acknowledgment_code_to_server = '2';
        }
    }
    else
    {
        acknowledgment_code_to_server = '3';
        printf("Unknown opcode received from server\n");
    }

    // if acknowledgment code from server is 0 - means ok
    // check is there is same file exist
    if ((fd = open(filename, O_RDONLY)) != -1)
    {
        printf("Requested file already exist: %s\n", filename);
        acknowledgment_code_to_server = '4';
    }

    // if no same file, create a new file
    if ((fd = open(filename, O_WRONLY | O_CREAT, 0766)) == -1)
    {
        if(errno == EACCES)
        {
            acknowledgment_code_to_server = '5';
```

```
            printf("Permission denied for %s\n", filename);
        }
        else if(errno == EINVAL)
        {
            acknowledgment_code_to_server = '6';
            printf("File name invalid: %s\n", filename);
        }
        else
        {
            acknowledgment_code_to_server = '7';
            printf("Unable to create: %s\n", filename);
        }
    }

    // respond server
    char code = (char)GET;
    if (write(sd, (char *)&code, 1) != 1)
    {
        perror("Error in sending opcode - get\n");
        return;
    }
    // sent acknowlegment code
    if (write(sd, (char *)&acknowledgment_code_to_server, 1) != 1)
    {
        perror("Error in sending acknowledgment code - get\n");
        return;
    }

    // if no issue - acknowledgment code is 0
    //else stop function
    if(acknowledgment_code_to_server != '0')
    {
        printf("Ending Get due to Error\n");
        return;
    }

    // start reading receiving the file
    // read opcode from server
    if (read(sd, (char *)&opcode, 1) != 1)
    {
        perror("Error reading opcode - get\n");
        exit(1);
    }
    // read file size
    int data_size;
    if (opcode == RESPOND)
    {
        printf("Server response: OK\n");
```

```
        if (read(sd, (char *)&data_size, 4) != 4)
        {
            perror("Error in reading filesize - put\n");
            exit(1);
        }
        filesize = (int)ntohl(data_size); // convert to host byte order
        printf("Filesize: %d\n", filesize);
    }
    else
    {
        printf("Server response: Opcode error\n");
        return;
    }

    // Get number of loops and the remainder bytes needed to get file from
server
    int blocks = filesize / (int)BUFSIZE;
    int remainder = filesize % (int)BUFSIZE;

    // read and write file content
    char content[BUFSIZE];

    for (int i = 0; i <= blocks; i++)
    {
        if (i == blocks)
        {
            if ((nr = ReadN(sd, content, remainder)) <= 0)
            {
                perror("Error in reading filename - put\n");
                close(fd);
                return;
            }
            if ((nw = write(fd, content, nr)) < nr)
            {
                perror("Error in writing the requested file - put\n");
                close(fd);
                return;
            }
            break;
        }

        if ((nr = ReadN(sd, content, BUFSIZE)) <= 0)
        {
            perror("Error in reading filename - put\n");
            close(fd);
            return;
        }
        if ((nw = write(fd, content, nr)) < nr)
```

```
        {
            perror("Error in writing the requested file - put\n");
            close(fd);
            return;
        }
    }

    // // read and write content
    // char content[BUFSIZE];
    // int blockSize = BUFSIZE;

    // while(filesize > 0){
    //  if(blockSize > filesize){
    //      blockSize = filesize;
    //  }
    //  if((nr = ReadN(sd,content,blockSize))<=0){
    //      printf("Error in reading the requested file - put\n");
    //      close(fd);
    //      return;
    //  }
    //  if( (nw = write(fd, content, nr)) < nr ){
    //      printf("Error in writing the requested file - put\n");
    //      close(fd);
    //      return;
    //  }
    //  filesize -= nw;
    // }

    close(fd);
    printf("Received: %s\n", filename);
}
```

token.h

```
/**
 * @file token.h
 * @brief A tokenizer that will split given string to tokens
 * @version 1.0
 * @date 2022-07-25
 *
 *
 */

#define MAX_NUM_TOKENS  100
#define tokenSeparators " \t\n"    // characters that separate tokens

/**
```

```
 * @brief  breaks up an array of chars by whitespace characters into
individual tokens.
 *
 * @param line a string that will be split into tokens
 * @param token array that will house the tokens
 * @return int  >=0: largest index of token array
 *              -1: on failure
 */
int tokenise (char line[], char *token[]);
```

token.c

```
#include <string.h>
#include "token.h"

int tokenise(char line[], char *token[])
{
    char *tk;
    int i = 0;

    tk = strtok(line, tokenSeparators);
    token[i] = tk;

    while (tk != NULL)
    {

        ++i;
        if (i >= MAX_NUM_TOKENS)
        {
            i = -1;
            break;
        }

        tk = strtok(NULL, tokenSeparators);
        token[i] = tk;
    }
    return i;
}
```

logger.h

```
/**
 * @file logger.h
 * @brief Helps manage logging program activity into a file
```

```
 * @version 1.0
 * @date 2022-07-25
 *
 *
 */

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>

#define logfile "sftp_log.txt"


void InitializeLogger();

void Logger(char* e);
```

logger.c

```
#include "logger.h"

void InitializeLogger()
{
    FILE *fd;

    fd = fopen(logfile, "a");
    if (fd == NULL)
    {
        perror("Error writing to log file");
        exit(0);
    }
    chmod(logfile, 0766);

    fclose(fd);
    return;
}

void Logger(char *e)
{
    time_t rawtime;
    struct tm *timeinfo;
    FILE *fd;
    char *output;
```

```
    fd = fopen(logfile, "a");

    if (fd == NULL)
    {
        perror("Error writing to log file");
        exit(0);
    }

    time(&rawtime);
    timeinfo = localtime(&rawtime);

    fprintf(fd, "%s  %s", asctime(timeinfo), e);

    fclose(fd);

    return;
}
```

Policy on the Reuse of the Third Party Source Code

- The structure of the server and client program are following the structure of given example from topic 8 example code 6, including functions such as serve_a_client(int sd), claim_children() and daemon_init.
- The tokenize function is taken from ICT374 lab 6