Remote File System

Introduction

This document describes the design principles used for creating a three-layered remote file system (RFS) service.

Mode of layering

The three layers are common for both the client and the server. These include:

- File Service Layer (L3) - The top layer in the stack is the file service layer (application layer) which accepts commands on the client side and executes those commands on the server side.
- Crypto Layer (L2) - The second layer (from the top) is the presentation layer where the message payload is encrypted or decrypted and passed on to the lower or upper layer respectively.
- Networking Layer (L1) - The third layer (from the top) is the transport layer responsible for binding/connecting to a port on a particular IP address and sending/receiving messages using the TCP protocol.

Flow of data: L3 → L2 → L1 —→ L1 → L2 → L3 → L2 → L1 —→ L1 → L2 → L3

where indicates the client side and indicates the server side. The connection between the two systems exists only at L1. Layer's L2 and L3 run only on the respective systems. To use the RFS, L1 should be run as it is in the point of entrance and exit from the server, and L3 should be run as it is the point of entrance and exit from the user on the client side.

Protocol Design and Implementation

Client-side stack

client_file_service.py:

The basic functionality is to parse commands and arguments provided by the user and ensure that it is a valid input based on the list of supported commands. In the case of `upd`, it also ensures that the file path entered is valid on the client system.

The supported commands are:

1. cwd
2. ls
3. cd <dir>
4. dwd <file>
5. upd <file>

In case of cd, dwd and upd, there are arguments to be passed along with the command name. A separation token ('`&!$`' is used as the delimiter) is inserted between the command name and the directory/file path argument. This is to ensure that the server understands which part of the message is the command and where the argument begins.

It also prompts the user to enter which mode of encryption he wants to use to send messages to the server. This mode is mapped to an integer from 1 to 3 and passed down to the crypto layer along with the message.

This file is also responsible for processing the decrypted response received from the server. The first bit of the decrypted response is the error flag. In case the error flag is true (1 indicates that there was an error in executing the command on the server side), the rest of the response contains the error message and that is displayed to the user. Since this service runs one command at a time, we already have access to which command was inputted and this data is not sent back from the server.

- For cwd - the rest of the response contains the current working directory and is displayed to the user.
- For ls - the rest of the response contains a list of the files and folders present in the current directory and is displayed to the user.
- For cd - the rest of the response is a message to indicate the successful change of directory and is displayed to the user.
- For dwd - the rest of the response contains a file name followed by a separator token followed by the file content. A new file with the same name is created in the client's current directory and the file content is written to it. The status of this file creation and file write is displayed to the user.

- For upd - the rest of the response is a message to indicate the successful file upload on the server system and is displayed to the user.

The program ends after the result is displayed on the client side.

client_encryption.py:

The supported modes of encryption for message-exchange are:

1. Plaintext - No change in input
2. Caesar cipher - Only alphanumeric characters will be substituted by an offset
3. Transpose - Reverse the contents in a word by word manner

This file receives three arguments - the message, the mode of encryption and a flag to indicate whether to encrypt or decrypt the message. In case of plaintext and transpose mode, this flag is not used because the algorithm is the same for both operations. In case of the Caesar cipher mode, the shift is fixed to be 2 or 26-2=24 based on this flag.

After encrypting the message, it appends a header to this message which contains the encryption mode number and passes this new message to the networking layer. The encryption mode number is not encrypted for the server to understand how to decrypt the data. Ideally, there should be some secret exchanged between the client and server which should be used in the encryption process so that no man-in-the-middle can decrypt the message even if he knows the mode of encryption.

client_echo_server.py:

Using Python's socket API, this file connects to the server's well-known IP address and port. The socket connection uses IPv4 address format (AF_INET). It receives a payload from the encryption layer, encodes it from utf-8 to a bytes format and sends it to the server socket using the TCP protocol (SOCK_STREAM). It is also responsible for receiving the response from the server, decoding it back to the utf-8 format and passing it to the crypto layer for decryption (the first header of the response holds the mode of encryption).

Server-side stack

server_echo_server.py:

This file is responsible for creating a socket object on the server side, binding to a fixed IP address and port, listening to connection requests from the client, accepting the requests and receiving messages from the clients. The implementation of the socket is similar to the client side. However, the server needs to bind to a specific port, unlike the client, as the server's port has to be known to the client to send a request to the server. It decodes the message received back to utf-8 format and passes it to the crypto layer for further processing. When it receives a response from the crypto layer, it encodes it into the byte format and sends it back to the client.

server_encryption.py:

Exact same implementation as client_encryption.py

server_file_service.py:

This file receives the decrypted request from the crypto layer below it. It splits the request based on the separator token and understands the command name by seeing the 0th index after the split. In case of cd, dwd and upd, it takes the text at the 1st index to be the directory/file path argument. In case of upd, the 3rd index is the content of the file to be uploaded. It uses the API's provided by the OS to navigate directories, and create, read and write files.

- For cwd - it uses os.getcwd()
- For ls - it uses os.listdir()
- For cd - it uses os.chdir()
- For dwd - it opens the file (provided as an argument), reads the contents and puts it in the response message
- For upd - it creates a file with the file name provided and writes the content onto it

The server appends an error bit to the response message to indicate if the command was run successfully or not. In case of an error, an error message is appended to the error flag. This response is then passed down to the encryption layer where it is encrypted and appended with an encryption mode number and sent to the network layer to be sent back to the client.

Challenges

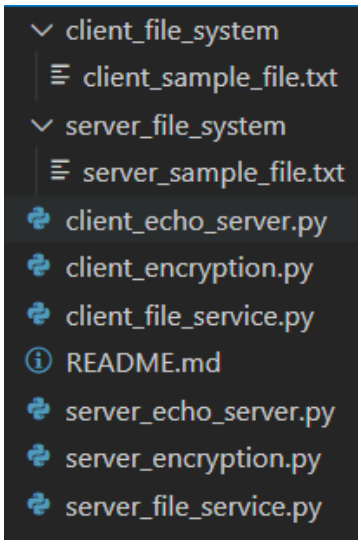
The major challenge was to understand the separation of code and flow of data among the layers. Even when the functionality of each layer was decided, it was difficult to realise where that functionality should be accessed and invoked. On the client side, the file that is active during an interaction is the file_service layer but on the server side, the transport layer file is the one which is always running.

Initially I had mapped each of these commands to a number and was sending that number in the messages. But I realised that the number won't get encrypted in any mode of encryption and thus had to change my design to pass the command names as string. One benefit of layering was that the changes I had to make were restricted to the file service layer of the client and the server stack and the layers below it were not involved in constructing the message data.

Another challenging aspect was to come up with a scheme to handle multiple parameters and separate them when passing messages and sending success/failure information along with the relevant details in either case.

Execution screenshots

Folder structure



cwd command

```
(base) PS D:\Files\Semester7\ComputerNetworks\Assignment1\remote-file-system> python client_file_service.py
What action do you want to perform? (Enter number from 1-5)
    1. cwd
    2. ls
    3. cd
    4. dwd
    5. upd
    1
What mode of encryption do you want? (Enter number from 1-3)
    1. Plaintext
    2. Substitute
    3. Transpose
    2
--
CLIENT sent request ...
CLIENT received response ...
D:\Files\Semester7\ComputerNetworks\Assignment1\remote-file-system
```

ls command

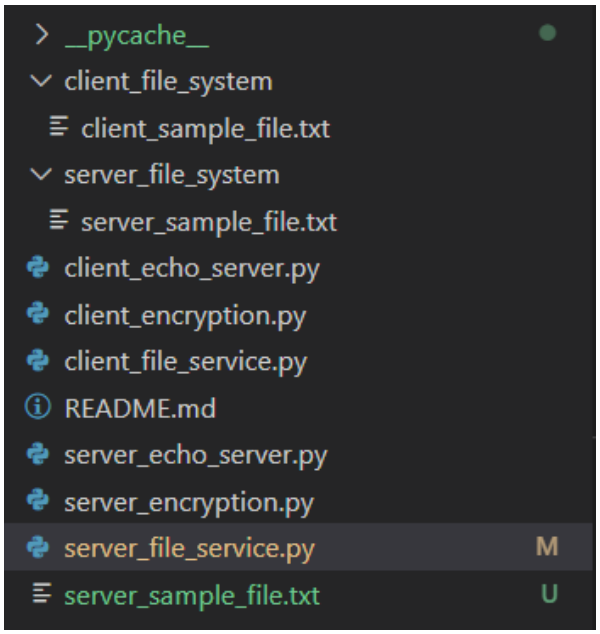
```
(base) PS D:\Files\Semester7\ComputerNetworks\Assignment1\remote-file-system> python client_file_service.py
What action do you want to perform? (Enter number from 1-5)
    1. cwd
    2. ls
    3. cd
    4. dwd
    5. upd
    2
What mode of encryption do you want? (Enter number from 1-3)
    1. Plaintext
    2. Substitute
    3. Transpose
    3
--
CLIENT sent request ...
CLIENT received response ...
['.git', 'client_echo_server.py', 'client_encryption.py', 'client_file_service.py', 'client_file_system', 'README.md', 'server_echo_server.p
y', 'server_encryption.py', 'server_file_service.py', 'server_file_system', '__pycache__']
```

cd command

```
(base) PS D:\Files\Semester7\ComputerNetworks\Assignment1\remote-file-system> python client_file_service.py
What action do you want to perform? (Enter number from 1-5)
    1. cwd
    2. ls
    3. cd
    4. dwd
    5. upd
    3
Enter path to change directory to:../
What mode of encryption do you want? (Enter number from 1-3)
    1. Plaintext
    2. Substitute
    3. Transpose
    1
--
CLIENT sent request ...
CLIENT received response ...
Directory changed successfully
```

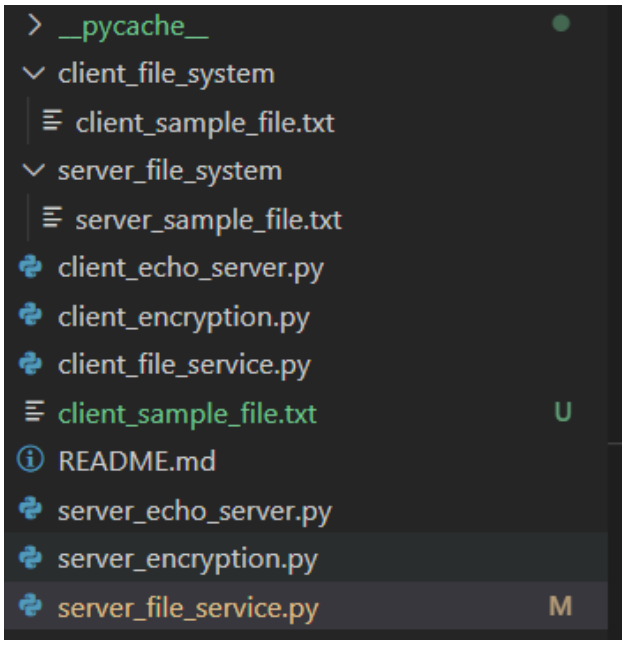
dwd command

```
(base) PS D:\Files\Semester7\ComputerNetworks\Assignment1\remote-file-system> python client_file_service.py
What action do you want to perform? (Enter number from 1-5)
    1. cwd
    2. ls
    3. cd
    4. dwd
    5. upd
    4
Enter filename to download:server_file_system/server_sample_file.txt
What mode of encryption do you want? (Enter number from 1-3)
    1. Plaintext
    2. Substitute
    3. Transpose
    2
--
CLIENT sent request ...
CLIENT received response ...
server_sample_file.txt downloaded successfully.
```



upd command

```
(base) PS D:\Files\Semester7\ComputerNetworks\Assignment1\remote-file-system> python client_file_service.py
What action do you want to perform? (Enter number from 1-5)
    1. cwd
    2. ls
    3. cd
    4. dwd
    5. upd
    5
Enter filename to upload:./client_file_system/client_sample_file.txt
What mode of encryption do you want? (Enter number from 1-3)
    1. Plaintext
    2. Substitute
    3. Transpose
    3
--
CLIENT sent request ...
CLIENT received response ...
client_sample_file.txt uploaded successfully
```



The server terminal shows the same response (the port number is different every time).

```
(base) PS D:\Files\Semester7\ComputerNetworks\Assignment1\remote-file-system>python server_echo_server.py
Connected by ('127.0.0.1', 62303)
SERVER recieved request ...
SERVER sent response ...
```