FTP : Server – Client Implementation

Authors :

- **Chetan Anand 11010117**
- **Pushpraj Yadav 11010153**

Description : File Transfer Protocol is a standard network protocol used to transfer files from one host to another host over a TCP-based network, such as the Internet. FTP is built on a client-server architecture and uses separate control and data connections between the client and the server.

FTP may run in ***active*** or ***passive*** mode, which determines how the data connection is established. We are using ***passive*** mode to implement our **FTP SERVER.** In this mode, the client uses the control connection to send a PASV command to the server and then receives a server IP address and server port number from the server, which the client then uses to open a data connection from an arbitrary client port to the server IP address and server port number received

What it does :

- It facilitates the reliable data transfer between Client(s) and the server by establishing a connection between them.
- It allows only following **FTP** commands : **put, get, ls, cd, pwd, !ls, !cd, !pwd, quit.** Otherwise for any other command it says **An Invalid FTP Command.**
- To allow multiple Clients to take part in data transfer simultaneously it uses **Connection Oriented Concurrent Server.**
- **Error Handling :**
    1. If any command other than that are mentioned above are entered, it will give an error "**An invalid FTP Command**".
    2. If a Client tries to "**get**" any file that doesn't exists in Server's directory, it will give an error "**filename : no such file**".
    3. If any Client tries to "**put**" any file to the Server directory which doesn't exist in Client's directory, it will give an error "**filename : no such file**".
    4. If Client enters a valid FTP command followed by an invalid option (eg. Ls –z where z is an invalid option) or an invalid directory name (eg. cd abc where abc is an invalid directory name) then it will give an error.

How does Concurrent Server Model Works :

Algorithm :

- **Parent Step 1 :** Create a socket and bind, leave the socket unconnected.
- **Parent Step 2 :** Place the socket in passive mode, making it ready for use by a server.
- **Parent Step 3 :** Repeatedly call ***accept()*** to receive the next request from a Client and create a new **Slave** process to handle the response.
- **Slave Step 1:** Receive a connection request creation, i.e. the Socket for the connection.

- **Slave Step 2:** Interact with the Client using the connection, read and send back the responses.
- **Slave Step 3 :** Close the connection and Exit. The Slave process Exits after handling all the requests from the Client.

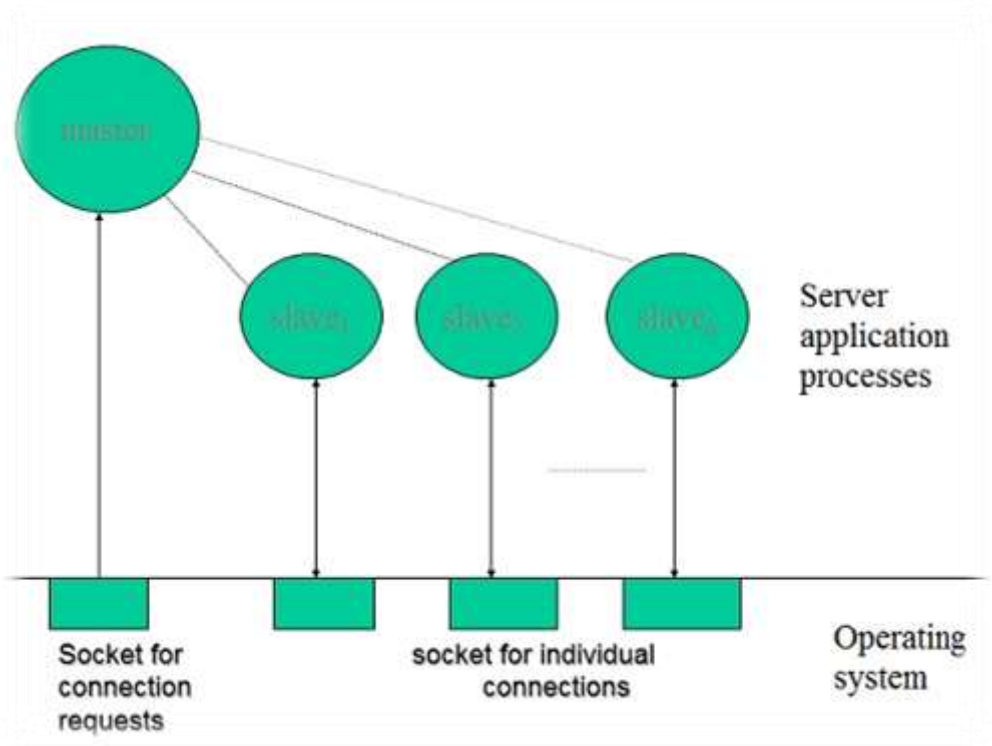

Figure 1 : Concurrent Server Model

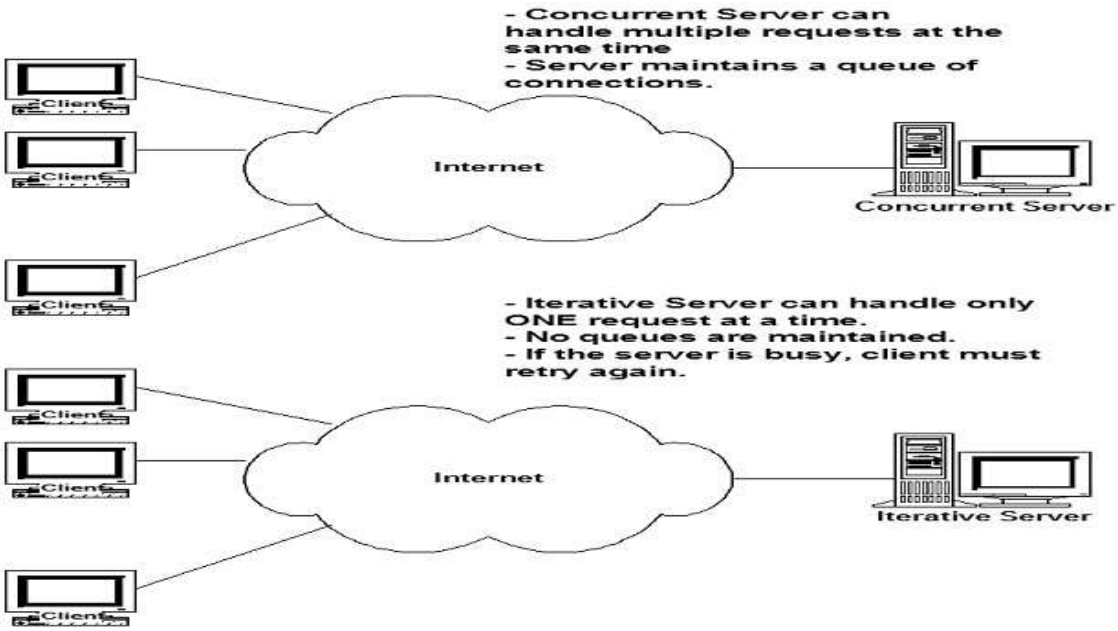

Figure 2 : Multiple Clloent Handling

Project Design：Interaction between Client and Server.

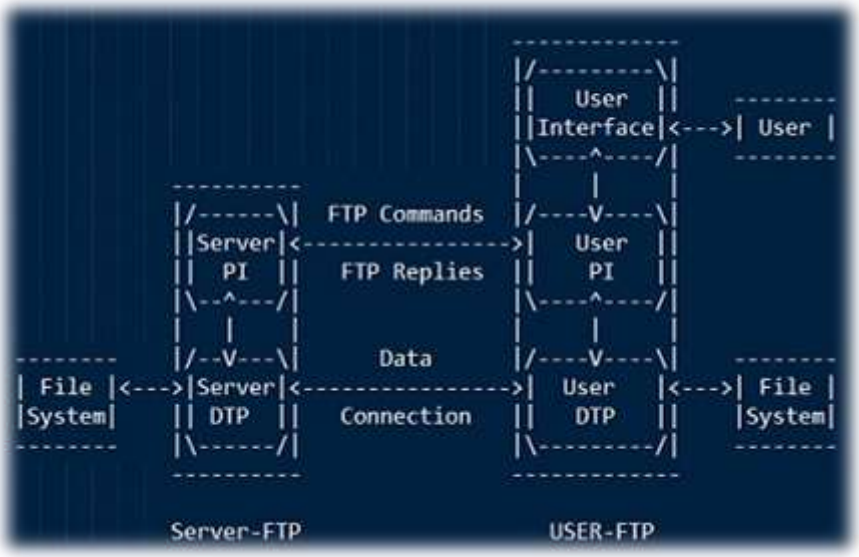

Figure 3 – FTP Model

NOTES:

- The data connection may be used in either direction.
- The data connection need not exist all of the time.

Server Design：

We need the server to support multiple connections at a time. To support this, we process each new client connection in a separate thread. We process both the data and the control connections for a particular user in a single thread, i.e. once a data connection is initialized, no control packets can be sent. We close the data connection on completion of the transfer. Once the client issues a quit command, we close the control connection, thereby terminating all communications.

Client Design：

The client connects to the advertised port of the server. After establishing the connection, the client starts an interactive shell. This shell processes input from the user and sends a request to the server. We print the server's response to the terminal and store/send a file from/to the server. Once the user types the quit command, we close the connection to the server and exit.

Project Modules：

We have the following modules:-

1. **Client Module :** Contains the main code for the client. This module's only function is to send/receive messages to/from the server and call functions to process the messages received from the server. The module also contains an interactive shell which processes user input and transforms them into messages for the client to send.
2. **Server Module :** Contains the main code for the server. This module's only function is to send/receive messages to/from the client(s) and call functions to process the messages.

Module Interaction :

Client Side :

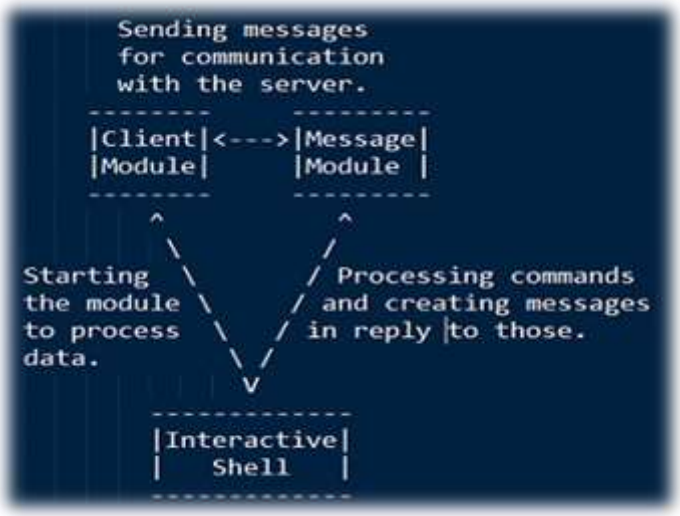

Figure 4 – Client Side Module Interaction

Server Side :

Figure 5 – Server Side Module Interaction

How Server Works :

- Main Process listens on **Port 21** for incoming connections.
- When a connection is received, a new process is **forked** to handle that client.
- This process exits when it is done serving the Client.

Sockets :

- **m-control_sock :** This is the Socket used to listen for incoming connections. Its value is same for main process and the forked process.
- **m-data_sock :** Set up when a **Client** sends a **Port** Request. Each forked Process will have its own data Socket (values will be different).
- **Client_control_sock :** This is passed to the function **serveClient.** It refers to the connection between **Port 21** of the Server and the **Control Port** of the specific **Client**.