FTP functionalities and definitions:

USER NAME (USER) => name: account

The argument field is a Telnet string identifying the user. The user identification is that which is required by the server for access to its file system. This command will normally be the first command transmitted by the user after the control connections are made (some servers may require this). Additional identification information in the form of a password and/or an account command may also be required by some servers. Servers may allow a new USER command to be entered at any point in order to change the access control and/or accounting information. This has the effect of flushing any user, password, and account information already supplied and beginning the login sequence again. All transfer parameters are unchanged and any file transfer in progress is completed under the old access control parameters.

PASSWORD (PASS) => password: user_password

The argument field is a Telnet string specifying the user's password. This command must be immediately preceded by the user name command, and, for some sites, completes the user's identification for access control. Since password information is quite sensitive, it is desirable in general to "mask" it or suppress typeout. It appears that the server has no foolproof way to achieve this. It is therefore the responsibility of the user-FTP process to hide the sensitive password information.

SYSTEM (SYST)

This command is used to find out the type of operating system at the server. The reply shall have as its first word one of the system names listed in the current version of the Assigned Numbers document [4].

PASSIVE (PASV) => ls => get file => put file

This command requests the server-DTP to "listen" on a data port (which is not its default data port) and to wait for a connection rather than initiate one upon receipt of a transfer command. The response to this command includes the host and port address this server is listening on.

LIST (LIST) => ls

This command causes a list to be sent from the server to the passive DTP. If the pathname specifies a directory or other group of files, the server should transfer a list of files in the specified directory. If the pathname specifies a file then the server should send current information on the file. A null argument implies the user's current working or default directory. The data transfer is over the data connection in type ASCII or type EBCDIC. (The user must ensure that the TYPE is appropriately ASCII or EBCDIC). Since the information on a file may vary widely from system to system, this information may be hard to use automatically in a program, but may be quite useful to a human user.

RETRIEVE (RETR) => get file

This command causes the server-DTP to transfer a copy of the file, specified in the pathname, to the server- or user-DTP at the other end of the data connection. The status and contents of the file at the server site shall be unaffected.

STORE (STOR) => put file

This command causes the server-DTP to accept the data transferred via the data connection and to store the data as a file at the server site. If the file specified in the pathname exists at the server site, then its contents shall be replaced by the data being transferred. A new file is created at the server site if the file specified in the pathname does not already exist.

LOGOUT (QUIT) => close => quit (if not closed)

This command terminates a USER and if file transfer is not in progress, the server closes the control connection. If file transfer is in progress, the connection will remain open for result response and the server will then close it. If the user-process is transferring files for several USERs but does not wish to close and then reopen connections for each, then the REIN command should be used instead of QUIT.

An unexpected close on the control connection will cause the server to take the effective action of an abort (ABOR) and a logout (QUIT)

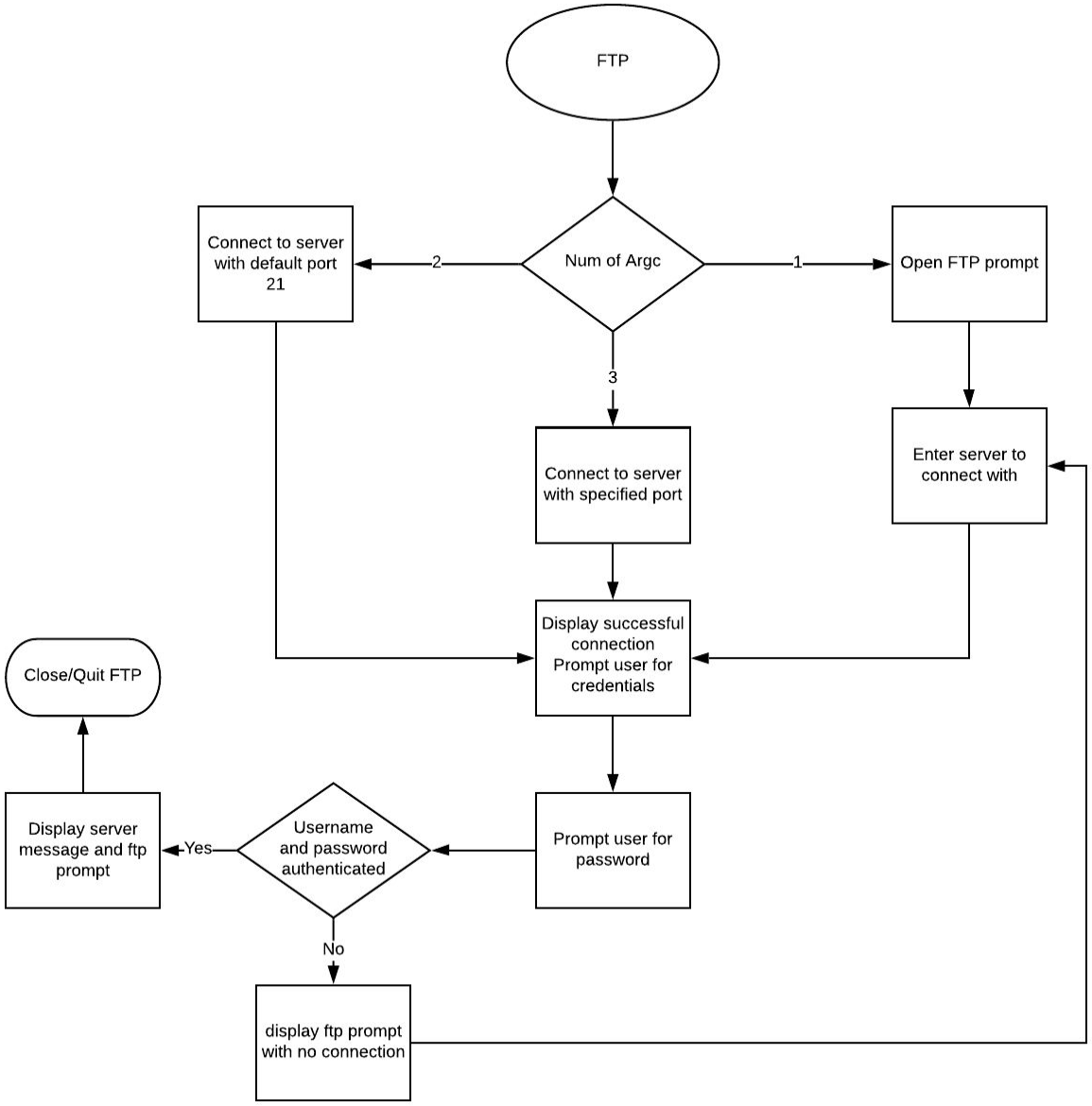

Fig 1. Interface (OPEN, PASS, SYST, QUIT)

This and many of the other methods utilized the strtok method frequently. Most of the data was best handled as char tokens, and keeping the data in this format made implementation the most convenient.

Implementation of **get**:
The first step to this was verifying the connection to a valid server before moving further. After a connection was established the key implement was use of the RETR command. Once inputs

were parsed and assigned a call was made to enter PASV mode. In PASV mode a passive socket connector was opened and functionality was then set so that the server was in listen mode waiting for the next action. Directly after a call was sent to the server to inform it of the data transfer method TYPE I. A helper method ‘receiveFile’ assisted in setting the permissions for local file creation including the `S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH` modes as well as the `O_WRONLY | O_CREAT` options. The special options work as follows:
O_CREAT: Make a file if it does not exist.
O_WRONLY: Open a file to write to
S_IRUSR: Read a file
S_IWUSR: Write a file
S_IRGRP: Read permission of group
S_IROTH: Read permission of other

These commands are common with PASV and RETR.

After the permissions were set a child operation was required to run in parallel with the main socket connection. Using fork() made it possible to perform a subset of actions that allowed for seamless action and then resume the main connection.

Implementation of **put:**
Put operated in nearly an identical manner as get, with a few small permission exceptions. Put operates with the PASV STOR protocols, which means when it is copying a file, it uses O_RDONLY (Open for read only) for the local directory.

Implementation of **cd:**
CD made use of sending a message to the server “CWD “ with an optional argument. The server understood standard command window/unix operators, so that when a path was specified, it was appended to the CWD string and sent to the server to initiate a change in path if the request is valid.

Implementation of **ls:**
The first thing ls needs to work is to initiate a call to PASV mode. After this a child process is made and a call is made by the child operator with the intent of receiving a blocking call. After the blocking call is received another command is sent to the server LIST, this requests that the server return the contents of the current directory. The child process is closed and the information is processed waiting for another command from the user.