Base Program Subroutines:

netopen(const char *pathname, int flags)
Return value:
netopen() returns the new file descriptor, or -1 in the caller's context if an error occurred (in which case, errno should be set appropriately). In order to avoid error and disambiguate your file descriptors from the system's, make your file descriptors negative (but not -1!).
Errors:
if netserverinit() is never called
If error connecting to the server (i.e server never opened or crashed
If error writing to server (i.e server crashed just as netopen tried to write)
If error reading from server (i.e server crashed just as netopen reads server return value)
If timed out because server took too long to respond (15 seconds)
If file descriptor returned from server is -1 (i.e one of the parameters is bad)

ssize_t netread(int fildes, void *buf, size_t nbyte)
Return value:
Upon successful completion, netread() should return a non-negative integer indicating the number of bytes actually read. Otherwise, the function should return -1 and set errno in the caller's context to indicate the error.
Errors:
if netserverinit() is never called
If error connecting to the server (i.e server never opened or crashed
If error writing to server (i.e server crashed just as netopen tried to write)
If error reading from server (i.e server crashed just as netopen reads server return value)
If timed out because server took too long to respond (15 seconds)

ssize_t netwrite(int fildes, const void *buf, size_t nbyte)
Return value:
Upon successful completion, netwrite() should return the number of bytes actually written to the file associated with fildes. This number should never be greater than nbyte. Otherwise, -1 should be returned and errno set to indicate the error.
Errors:
if netserverinit() is never called
If error connecting to the server (i.e server never opened or crashed
If error writing to server (i.e server crashed just as netopen tried to write)
If error reading from server (i.e server crashed just as netopen reads server return value)
If timed out because server took too long to respond (15 seconds)

int netclose(int fd)
Return value:
netclose() returns zero on success. On error, -1 is returned, and errno is set appropriately
Errors:
if netserverinit() is never called
If error connecting to the server (i.e server never opened or crashed
If error writing to server (i.e server crashed just as netopen tried to write)
If error reading from server (i.e server crashed just as netopen reads server return value)
If timed out because server took too long to respond (15 seconds)

netserverinit(char * hostname)
Return value:
0 on success, -1 on error and h_errno set correctly
Errors:
If hostname is invalid
If server is not open

Subroutines implemented to construct base program subroutines:

int conn_info setconnection()
Return value:
Creates a socket and attempts to connect to the server. Upon a successful connection returns 1 or -1. Will also set the global variables file descriptor and struct addrinfo rp. On error, -1 is returned and h_errno is set appropriately.
Errors:
If getaddrinfo() fails
If connect() fails for every address structure returned by getaddrinfo()

int time_out(int fd)
Return value:
Given no response from the server in a selected amount of time, 0 is returned an errno is set appropriately. If there is a response from the server return 1. If there was an error with select() then return -1.
Errors:
If select() failed
If the server took too long to respond

char *fill_with_spaces(char *message)
Return value:
Given a string of size s, it will append 100 minus s spaces to the end of the string until the size is 100 and then return that string.
Errors:
None

Strategy
We employed a one connection per command strategy. This means that we open and close a connection every time the client calls netopen, netread, netwrite, netclose or netserverinit.

Below is a summary of the plan we used per subroutine on the client side

netopen
We check if netserverinit has been called.
Then we call set_connection to establish a connection with the server.
Next we construct a string to send to the server.
    For netopen all strings will follow this pattern: “open pathname flags”
    In constructing the string we convert flags from a int to a string.
    Then we append everything and send the message to the server
After we write to the server we expect a number to be received
We read the number from the server. It is a string so we convert it to a integer.
If the number is negative the client should expect to receive the errno associated with the error.
Thus we read one more time from the server, receive errno number, set errno and return -1
If the number is positive, we have received a file descriptor. We convert that file descriptor to a negative number and return that.

netread
We check if netserverinit has been called.
Then we call set_connection to establish a connection with the server.
Next we construct a string to send to the server.
    For netread all strings will follow this pattern: “read fd nbytes”
    In constructing the string we convert fd and nbytes from int to string.
    Then we append everything and send the message to the server
After we write to the server we expect a number to be received
We read the number from the server. It is a string so we convert it to a integer.
If the number is negative the client should expect to receive the errno associated with the error.
Thus we read one more time from the server, receive errno number, set errno and return -1
If the number is positive, we have received the number of bytes read. This means read was successful on the server side. Then we must expect the read bytes to be sent to the client. We read one more time to receive what was read and then memcpy() it into the buffer given by the user.

netwrite
We check if netserverinit has been called.
Then we call set_connection to establish a connection with the server.
Next we construct a string to send to the server.
    For netwrite all strings will follow this pattern: “write fd nbytes”
    In constructing the string we convert fd and nbytes from int to string.

Then we append everything. Before we send the message to the server we append spaces to the end of the message until the message is 101 bytes long because the server expects exactly 101 bytes. Then we send the message to the server.
After we send that 101 byte message to the server we send also send the buffer to be written into the file
Thus we write twice to the server.
After we write twice to the server we expect a number to be received
We read the number from the server. It is a string so we convert it to a integer.
If the number is negative the client should expect to receive the errno associated with the error.
Thus we read one more time from the server, receive errno number, set errno and return -1
If the number is positive, we have received the number of bytes written. This means write was successful on the server side and we return that number.

netclose
We check if netserverinit has been called.
Then we call set_connection to establish a connection with the server.
Next we construct a string to send to the server.
For netclose all strings will follow this pattern: “close fd”
In constructing the string we convert fd from int to string.
After we write twice to the server we expect a number to be received
We read the number from the server. It is a string so we convert it to a integer.
If the number is negative the client should expect to receive the errno associated with the error.
Thus we read one more time from the server, receive errno number, set errno and return -1
If the number is positive, it must be 1, thus close succeeded on the server side and we return 1.

Below is a summary of the plan we used for the server
We have a server subroutine that opens a server. The server listens forever. For every connection accepted we spawn a thread that handles the connection. Then the thread will detach once the routine handed to it is done. The routine/function handed to the thread will read whatever message was sent from the client and decipher it. It will tokenize the string. We have made sure that the library functions always send a well packaged message to the server. Thus all messages will mirror one of these strings:
“Open pathname flags”
“Read fd nbytes”
“Write fd nbytes”
“Close fd”
If, somehow, any message other than those described is sent by accident or maliciously, the server does nothing and closes the socket file descriptor as if nothing happened.

Now, after we have tokenize the string, the server will examine what the first token is. However, before we describe what we do with the token, we will describe what our server_open(), server_read()... etc do. They simply parallel the library functions, netopen(), netread()...etc. If netopen was called on the client side and a message was sent to the server, the server calls

server_open() which will then try to open the file and then send a message back to the client. Server_read, server_write, and server_close do the same thing.
Having clarified our server subroutines, we will now explain what we do with the first token. If the token is “open” we call server_open() which will send the right message back to the client after attempting open. If it is “read” we call server_read which will send the right message back to the client after attempting read. And so on for the rest of the net commands.
The server commands are the equivalent of the net commands. Depending on which net command was called, the server will appropriately choose the right server command so that it can send a correctly packaged message back to the client.