This program deals with a multi-threaded server that holds a repository of projects that clients can perform different commands on. The commands that a client can call are: configure, checkout, update, upgrade, commit, push, add, remove, history, rollback, create, destroy, and currentversion. Clients are able to request projects or specific files from the server so that the client has its own local copy of them. Each project holds a file called a Manifest which contains a list of files that are in the project along with each file's version number and hash. At the end of each line in the Manifest, a 0 is written so that the user can see if the file has been looked at by the server. Based on differences between the server project and client project manifests, the client and the server can update or commit changes so that the client is up to date with the server or the server is up to date with the client respectively. Below is an explanation of each of the commands that the client can run.

Configure:
The configure function accepts two arguments, the IP address / hostname and the port number. This function stores the inputs in a .configure file for the client to access when it needs to connect to a server. Any command that needs to connect to the server will not run if there is no .configure file.

Checkout:
The checkout function takes in the project name. When this function is run, the client asks the server for a copy of the project requested. This is done by creating one string with the file paths and file contents and another string containing the directories within the project. The way the program stores the file path and contents is as followed: <length of file path> : <file path> : <file size> : <file contents> …. Each file and their contents is listed in the string so that the server can easily send it over to the client. The colon separates the information. The same is done for the string of directories: <length of directory name> : <directory name> …. Once the string are created, the server traverses through the directory string to send the names of the directories to the client so that the client can create them. The server first sends the length of the directory name, and then the directory name by using the colons as delimiters. Once the client has created the directories, the server then sends the file string. The server first sends the length of the file name, then the file name. The client creates the file, and then the server sends the file size and file contents so that the client can write the contents into the file. Once this function is done, the client should now have an exact copy of the project from the server's repository.

Create:
The create function takes in the project name. When the create function is called, we first check to see if the project file already exists in the Server repository. If it does, we alert the user and end the program. If it does not exist, we create an empty directory with the user-given project name. We then create a .Manifest file and start the .Manifest version count at 0. We also create a

hidden file in the Server folder with the title .<project name>_Archive. This is created to store older versions of the project to make rollback easier. Additionally, this function creates a hidden text file named .History in the archive folder to keep track of all successful pushes completed. Once the Server has finished creating the directory, the Client side also creates an empty directory with the project name and creates an empty .Manifest file with version number 0 placed inside of it. The Client will then receive and print a "New project created!" message that lets them know the task has been completed.

Destroy:
The destroy function takes in the project name. When the destroy function is called, we first check to see if the project file exists in the Server repository. If not, we alert the user and end the program. If it does exist, the program locks the repository so that no other threads can have access to the project while its being deleted. After locking the repository, we find the project folder and recursively delete all the files and subdirectories inside it. Then, once the directory is empty we are able to remove it. We repeat this same process for hidden archive file we created for the project as well, which was used to store previous versions of the project. Once both directories have been deleted, the Client receives the success message that lets them know the task has been completed and the project is then unlocked.

Add:
The add function takes in the project name and the file name to be added (file path is also acceptable for files in subdirectories). When this function is called, we first check to see if the project exists in the Client repository. We then open the .Manifest file inside the project repository and append the version number, file path, and the hash all in the same line. We assumed that when adding, the version number would always have to be 0, because it is a new file that has not been updated yet. We write the same file hash provided by the user and compute a live hash of the file. We also add a number 0 at the end of this line to help us keep track of files that have been checked already for other functions (update and commit).

Remove:
The remove function takes in the project name and the file name to be deleted (file path is also acceptable for files in subdirectories). When this function is called, we first check to see if the project exists in the Client repository. We then open the .Manifest file inside the project repository and check to see if the given file path is recorded in the .Manifest using a self-created substring function. When given the full string and the indices of where we want to start and end, this function returns the string in those indices. We use this substring function to find the index of the beginning and end of the line we want to remove. If the file name was found in the .Manifest file, we create a .ManifestTemp file where we copy all the contents up to the beginning of the line and then we copy the contents after the line. Since we are not recopying the line we

want to remove, it is no longer in the .Manifest. We then delete the .Manifest file and rename the .ManifestTemp file to .Manifest.

Update:

The update function takes in the project name. This function fails if the project name does not exist on the server. This function compares the client and server Manifests of the specified project and takes note of any differences between them so that the client can update the project. Each difference is printed out and is recorded in an Update file. The first thing that is checked is the server / client project versions. If they are equal, the server has no updates for the client and "Up to Date" is printed out. Additionally, an empty Update file is created and any Conflict files are deleted. If the project versions are different, then the client needs to be updated with the server. There are four different cases for update. If the client Manifest has files whose version numbers and stored hashes are different from the server Manifest and the live hash of those files match the hash in the client's Manifest, then the client has to modify the file. An M, the file path, and the server's hash is written into the Update file and an M and the file path is printed out. If the client's Manifest does not have a file that is in the server's Manifest, then the client must add the file. An A, the file path, and the server's hash is written into the Update file and an A and the file path is printed out. If the client's Manifest has a file that is not in the server's Manifest, then the client must delete the file. A D, the file path, and the client's hash is written into the Update file and a D and the file path is printed out. If the client's Manifest has a file whose stored hash is different from both the server's Manifest and the live hash of the file, then it is a conflict. If a conflict is found, a Conflict file is created and a C, the file path, and the live hash is stored in it. Additionally, a C and the file path is printed out. The program will still continue to find all updates and conflicts, but at the end, the program will delete the Update file and print out "Conflicts were found and must be resolved".

The program compares the Manifests by taking a file from the server Manifest and comparing it with the files from the client Manifest. If the server file is found in the client's Manifest, then the 0 at the end of the file line in the client's Manifest becomes a 1. This is so that the program knows that the server and client have the same file. Then, the program checks the server hash, client hash, and the live hash of the file to determine if it is a modify case or if it is a conflict. However, if the server file is not found in the client's Manifest, then it is an add case. After the program checks through each file in the server Manifest for the modify, add, and conflict cases, the program checks through the client's Manifest to see which files need to be deleted. Since files that are part of both the server and client have been marked with a 1 in the client's Manifest, the files that are marked with 0s are the ones that need to be deleted. Once the delete case is checked for, all the 1s in the client Manifest are reset to 0.

Upgrade:

The upgrade function takes in the project name. This function fails if the project name doesn't exist on the server, if there is no Update file, or if there is a Conflict file. If no Update file exists, the client tells the user to run update first. If a Conflict file exists, the client tells the user to resolve all conflicts and then update. If the Update file is empty, then the client tells the user that the project is up to date. However if it is not empty, upgrade reads through the Update file and implements the changes on the client side. Files marked with D will remove the file from the client's Manifest. Files marked with M will request the file contents from the server and overwrite the file in the client. The hash of the file will also be updated in the client's Manifest. This is done by removing the file line from the Manifest, then re-adding it. Files marked with A will request the file contents and the file's version number from the server. The client will then create a new file containing the contents from the server. Then, the file will be added to the Manifest using the add function from above. At the end, the client requests the project version from the server and the project version will be updated in the client's Manifest. This is done by rewriting the Manifest.

Commit:
The commit function takes in the project name. When this function is called, we first check to see if the project exists in the Server repository. If it does not exist in the Server repository, we alert the client and the program. If it does, we then proceed to check if a .Conflict file exists in the project repository. If a .Conflict file does exist, we alert the user that they need to resolve all conflicts first before they can proceed and the program ends. If no .Conflict file exists, we then check to see if a .Update file exists in the project repository. If it does, we check to see that the size is 0. If the size of the .Update file is not 0, we alert the user that a non-empty .Update file exists and the program ends. If no .Update file exists, or if it does exist and the size is 0, we can proceed with the update command. The server then sends over the .Manifest in the project repository. The client receives it and then opens its own project's .Manifest. The .Manifest version numbers are first determined by searching the files for the first newline character and then compared to each other. If the version numbers of the .Manifest files are different, the Client receives the message that it needs to update it's local project first and the program ends. If the versions are equal, a .Commit file is created in the project repository.

The server and client .Manifests are then compared and any changes are recorded in the .Commit file. There are three different cases for commit: add, delete and modify. If the client's .Manifest has a file recorded that the server's .Manifest does not have, then the server has to add the file. An A, the client's file version incremented, the file path, and the client's hash is written into the Commit file and an A and the file path is printed to STDOUT. If the server's .Manifest has a file recorded that the client's .Manifest does not have, then the server needs to delete that file. A D, then server's file version incremented, the file path, and the server's hash is written into the .Commit file and a D and the file path is printed to STDOUT. If the client's and server's .Manifest has the same file with the same hashes, but the live hash of the file in the client repository is different from its stored hash, then the server needs to modify the file. An M, the client's file version incremented, the file path, and the live client hash is written into the .Commit file and an M and the file path is printed to STDOUT. In the instructions it says to write the server's hash to the .Commit file, but we wrote the live client hash into the .Commit file because we assumed the whole point of the pushing function is to modify/change the server file to make it match the current client file. Including the outdated server's hash did not make sense to us as the hash's would later not match after being pushed. If there is a file in both the server's and client's .Manifest, but the hashes do not match, the version numbers are then checked. If the server file's version number is greater than the client file's version number, this is considered a commit failure. In a failure, the .Commit file is deleted, the client is then notified that they have to sync the repository before commiting changes and the program ends.

The program compares the Manifests by taking a file from the server Manifest and comparing it with the files from the client Manifest. If the server file is found in the client's Manifest, then the 0 at the end of the file line in the client's Manifest becomes a 1. This is so that the program knows that the server and client have the same file. Then, the program compares the server's hash with the client's hash to determine if it is a modify case or failure. If hashes are equal, the live client hash is calculated and compared to stored client hash. As mentioned earlier, if the live client hash is different from stored, the server needs to modify. If the hashes are not equal, file version numbers are checked, compared, and if server version number is greater than client version number, the commit fails. If the server file is not found in the client's manifest, it is an add case. After the program checks through each file in the server .Manifest for the modify and add cases, the program checks through the server's .Manifest to see which files need to be deleted. Since files that are part of both the server and client have been marked with a 1 in the server's .Manifest, the files that are marked with 0s are the ones that need to be deleted. Once the delete case is checked for, all the 1s in the server's .Manifest are reset to 0. If the commit was a success, the .Commit file is sent over to the server with the client's IP address in the file's name and a success message is printed.

Push:

The push function takes in the project name. It fails if it cannot connect with the server, the project name doesn't exist on the server, or if there is no .Commit file in the client's project folder. If the project exists on the server, the project is then locked so that no other thread can get to the project while the project is being changed. Push reads through the .Commit file and implements the changes to the server side. If there are multiple .Commit files in a case with multiple users, we make sure to check which .Commit files have the IP address that matches the IP address of the user trying to push. If a .Commit file is not found from the client that is requesting the push, then the server sends an error. However, if an active .Commit file is found and matches the .Commit file on the server side, all other .Commit files are deleted. Next, the program checks to see if the contents of the server and client Commit files are the same. If the

.Commit found on the server side did not match the .Commit on the client side, the user would be notified that the commits do not match and the program would end. If they do match, the program continues. Next, the project creates a duplicate and stores the old version of the project in the archive folder. The project is first renamed with the project's version number. For example, <project name> _ <version number>. The project is then copied into the archive folder and is compressed using TAR. Then the project in the server directory is renamed to the original project name. Next, the project version in the Manifest is incremented. Once this is done, the program begins reading through the Commit file to perform the actions indicated. Files marked with D will remove the file from the server's Manifest and delete its contents on the server side. Files marked with M will request the file contents from the client and overwrite the file in the server. The hash of the file will also be updated in the server's Manifest using information in the .Commit file. This is done by removing the file line from the Manifest, then re-adding it. Files marked with A will request the file contents and the file's version number from the client. The server will then create a new file containing the contents from the client. Then, the file will be added to the Manifest using the add function from above. Once all of the actions are performed and the push is successful, the client receives a success message from the server and the client then deletes its Commit file. The repository is then unlocked.

Current Version:
Current version takes in the project name. It will fail if the project name does not exist on the server. This function has the client request for the server's Manifest. Once the client has the server's Manifest, the program traverses through the file and outputs all the files and their version numbers.

History:
The history function takes in the project name. It will fail if the project name does not exist on the server. The client requests the server's History file for the project. This is stored in the project's archive folder on the server side. Once the server retrieves the file, it sends the contents of it to the client. The client then prints out the contents of the History file. This file contains a list of all of the operations performed on all successful pushes since the project was created. The project's version numbers separate each successful push. Additionally, the History file also contains information about rollbacks. If a rollback was performed, the History will have a line that says: "Rollback to project version __".

Rollback:
The rollback function takes in 2 inputs: the project name and the version number to rollback to. This function will fail if the project name does not exist on the server or if the version number is invalid. The function first checks to see if the project exists on the server. If it does, then the project is locked. This is so that no other client can request information from the project while it

is being reverted back to an older version. Next, it checks if the version number is invalid. It does this by comparing the server's project version with the input. If the input is greater than or equal to the server's project version, then the function fails since the version number inputted is invalid and can not be rolled back to. However, if the input version is less than the server's project version, then the program continues. The program then goes into the project's archive folder to find the project that matches the input version. Each old project in the archive is named as followed: <project name> _ <version number>.tar.gz. The program can easily find the version it is looking for just by reading the name of the old project TAR file. Once the old project is found, all the recent versions of the project will be deleted, the old project will be moved out of the archive directory, it will be untared, and will be renamed to the project name (without the version number in the name). After this is done, the program writes "Rollback to project version __" in the project's History file.

The server and client were set up using sockets. For the server program, the accept function was put in an infinite while loop so that multiple clients could connect to it. Additionally, a thread was created for each client that connected to the server. This allows for multiple clients to access the server at the same time. A thread was only created for a client if the project was not locked. If the project is locked, then an error message will be displayed telling the user that the repository is locked. However, if the project is unlocked, then a new thread is created for the client. This was done by creating an array of threads. When a server connects to a client and the project is unlocked, a new thread is created in the thread array, the index for this array is then incremented. The server can be disconnected by using CTRL-C. The program catches this signal, closes the socket file descriptors, free up any memory, and exits. For the client program, the connect function stays in a loop until the function returns a 0. This allows the client to be invoked before the server. The client will try to find a server every 3 seconds. Once a server is found, the while loop is exited and the rest of the program continues. However, if the client does not find a server, the user can exit the program using CTRL-C. The program will catch this signal, close all socket file descriptors, free up any memory, and then exit. In order to implement locking for destroy, push, and rollback, we create a linked list that stores the project name and the respective mutex. When create is called, a new node is created containing the project name and the initialized mutex. Then, when destroy, push, or rollback is called, the project name is searched for in the linked list, and the mutex for the project is found. The mutex is then locked at the start of the command's function and unlocked at the end of the command's function. The reason we use mutex to lock the repository for these three commands is because we don't want any other clients trying to request information from the server or try to change the project on the server while the project is being changed or deleted by the first client that calls one of the three commands.