Aditi Singh and Rumeet Goradia (as2811 and rug5)
Asst3
CS214, Section 04

Read Me

In this assignment we designed a program that builds a version control system. The resulting program contains a server side that maintains all the projects, while the client side maintains all the commit and fetches updates and communicates them to the server. This program uses multithreading so that the server can accept multiple clients. We used only one mutex per thread and each thread within the program is handled by "thread_handler()". Any client request that occurs will only deal with one project. Due to this, since each client request is a thread it only locks one mutex at a time.

WTF.c is the main source of client-side functionality. Based on the command-line input of the user, the program will do one of many things. For the commands **add**, **remove**, and **configure**, WTF.c is able to act without connecting to a server. It stores the IP address and port number given with **configure** in a file called .configure. For configure it first ensures that the file is a directory and then ensures that the file exists within the project. It then sets up the .Manifest file path and gets the input of the file in order to hash it. It then adds this information to the .Manifest file and sets up a socket so that the client can communicate with the server. When dealing with **add** or **remove**, it opens the local project's .Manifest and uses helper functions, located in helperfunctions.c, to perform the required task. **add** will fail if the hash of the file hasn't changed since the file was last added to the .Manifest, and **remove** will fail if the file isn't in the .Manifest.

The function **create** will create a .Manifest file as requested and send it over to the client. The client then sets up a folder with the file and places the .Manifest that the server sent. This function will send back a single char representing the project name based on whether the function was successful or not. This function will fail to create the file if the a file with the same name already exists or if the client fails to communicate with the server.

The function **destroy** will delete a file as requested by locking the repository and deleting all files and subdirectories. This function will also send back a specific char based on whether the function was successful or not. If the project name does not exist or if the client cannot reach the server this function will fail.

The function **current version** sends the size of the server's .Manifest file for a specified project to the client so that the client is aware of how many bytes are expected to be received. After this, it then receives the .Manifest file itself from the server. The function then sends the input of the same .Manifest file to the client. The client then passes through the received input and then subsequently prints out the file number and file path. The function skips over the hash code, however, because it is not necessary for this function. If the project does not exist on the server the function fails.

The function **commit** will first check if the .Update file exists and if it does it will then check whether it is empty. It then receives the server's .Manifest and will compare it to that of the client's side, however unlike **update** it will be marking changes based on the server. If there are discrepancies between the two files the operation will cease. From this, the .Commit file will be set up and if the .Commit is empty this function will stop. In terms of error checking, we ensured that there was a checker to see whether the server was able to create a .Commit file.

The function **push** will first check if the .Update exists, is not empty and contains an (M) code. If it does this function will fail. If not, then this function will sent the .Commit file to the server and read the .Commit's input. From this, it will check if the server found its matching .Commit file. On the client side a copy of the current .Manifest will be created and the version number will be updated. The new .Manifest will be implemented immediately, but if this fails it will revert to the old .Manifest file. We then used a tokenizer to keep track of where the last whitespace was and whenever we encountered a new white space, we took the part of the string between those the last whitespace and the new white space and created a token out of it. Since the file that this function is tokenizing is always the formatted the same we used a counter to keep track of what each token is.

The method **update** only needs the servers .Manifest file in order to work properly. It will then open the local .Manifest and the client will then compare the .Manifest file to that present in the server, recording all the differences in a .Update file which will be stored with the client. The files will be labeled with specific tag to signal what the changes are.

The method **upgrade** opens the local .Update file and sends it over to the server. It then tokenizes the .Update data and uses the .Update within the client to do three tasks: (M): fetch the file from the server and replace the client sides file with it; (A): is the same as (M); (D): will delete the client sides file. For anything that was an (M) or (A) code it will get the file's contents from the server. Since the new .Manifest file will be the same as the server's we can just fetch it from the server side. From this it successfully opens the local .Manifest file.

The command **rollback** will send rollback code, the project number, and version number all in one go. This command will revert a project back to the version number requested by the user by deleting all the most recent versions on the server's side. The client does not need to have a local copy of the project in order for this function to work. If the project does not exist, the version number does not exist, or if the client cannot successfully communicate to the server this function will fail to work.

The function **history** sends over a .History file from the server with all the operations performed to the project folder. The client does not need to have a local copy of the project in order for this function to work. If the project does not exist or if the client cannot successfully communicate to the server this function will fail to work.

The method **checkout** lets the client request the project from the server. The server will then search for the project within the server repository and will send the latest version. The client will then create the project, its subdirectories, as well as the .Manifest.

This program has comprehensive error checking that checks all different types of errors, including, but not limited to, errors that would be a result of miscommunication between the client and the server as well as scenarios in which the project folder does not exist or if the same project is trying to be created.