Sunehar Sandhu and Jennifer Rodriguez
Asst3

Read Me

For this project we had to build a simplified version control system. There is a server and client, where the server maintains all the projects and the client maintains all the commits and fetches updates and communicates them to the server. This program implements the use of multithreading in C so that the server can handle multiple clients at once. Any client request deals with only one project at a time, therefore, the client only locks one mutex at a time.

WTF.c is the client side program that contains all of the client functionalities. The commands used in the client are the following. For the commands add, remove, and configure; the client is able to act without connecting to a server. It stores the IP address and port number given from the .configure file. *Add* will fail if the hash of the file hasn't changed since the file was last updated to the .Manifest, and *remove* will fail if the file isn't in the .Manifest.

The command *create* will create a .Manifest file as requested and send it over to the client. The client sets up a folder with the file and places the .Manifest that the server sent. It will then send back a single character representing the project name. This command fails to create the file if the file with the same name already exists or if the client fails to communicate with the server.

The destroy *command* deletes a file by locking the repository and deleting all files and subdirectories. Then it sends back a specific character based on whether it was successful or not. If the project name does not exists or if the client can't communicate with the server, then this command fails.

The *currentversion* command sends the size of the server's .Manifest for a specified project to the client so the client is aware of how many bytes are expected to be received. This is necessary so that when the client receives the file, the client passes through the received input and then prints out the file number and path. This command skips over the hash code. Command fails if the project does not exist on the server.

The *commit* command checks if the .Update file exists and if it does, it will check whether it is empty. It then receives the server's .Manifest and will compare it to that of the client's side and marks changes based on the server. If there are discrepancies between the two files, then the operation stops. The .Commit file will be set up and if the .Commit if empty the command will stop.

The *push* command makes all the changes listed in the .Commit to the server side. It first checks if the .Update file exists and contains an M code. If it does this function will fail. Otherwise, the function will send the .Commit file to the server and read the .Commit's input. Then, it checks 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. A new version of the .Manifest will be implemented but if it fails, it will revert back to the old one. 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 the last whitespace and the new white space to create a token. Since the file that this function is tokenizing is always formatted the same we used a counter to keep track of what each token is.

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

The upgrade method opens the local .Update within the client to do three things: (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. 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 *rollback* command sends a rollback code, project number, and version number all together. This command reverts the project back to the version number requested by the user by deleting all the most recent version’s on the server’s side. The client does not have a local copy of the project. If the project or version number do not exist, or if the client cannot successfully reach the server, the command fails.

The *history* command sends over a .History file from the server with everything that had been done to the project folder by the previous commands. 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 *checkout* command 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 creates the project and its subdirectories, as well as the .Manifest.