Ashwin Haridas, ah1058 (Section 06)
Ritin Nair, rrn32 (Section 04)

Assignment Last: Where's The File?

Purpose:

The purpose of this program is to create a version control system where the user can make changes to a local copy of a project and have the changes they make be stored in a repository of projects on a server. The user is also able to update local versions of their project by upgrading to the version on the server, or they are able to get a local copy of a project from the server by using checkout. This version control system ensures that there is consistency when multiple users are working on a shared project and want to guarantee that they are all working on the same version without creating conflicts with each other.

Build Instructions:

To build the necessary files and directories necessary to run this program the user must first call make in the current working directory. This will create a client directory, a server directory, any required object files for libraries, an executable named WTF in the client directory, and an executable named WTFserver in the server directory. For any client side operation the user must first cd into the client directory and then run the operation with ./WTF <operation> <args …>. To use the server the user must first cd into the client and run the command ./WTF configure <IP Address/hostname> <port number>. Then, they must cd into the server directory in a new terminal and run ./WTFserver <port number>, where port number is specified in a .configure file in the client directory.

WTFtest:

To build the necessary files and executable for WTFtest, run "make test". After this, running "./WTFtest" will execute some basic tests for our project. More details about WTFtest are located in testplan.txt

<u>*IMPORTANT*:</u>

- To run client commands first cd into the client directory and run the command with ./WTF <command><args…>
- To use the server in any commands that require communication between the client and server first cd into the server directory and then run ./WTFserver <port number> in the server directory, where port number is specified in a .configure file in the client directory

Optimizations (+10 Extra Credit):

We successfully implemented the first extra credit option, compressing old versions of projects at the repository. For any previous versions of a project, we compress (using tar) and place it in a backups folder. When "rollback" is called, we identify the project version and decompress using system calls.

Input:

- *For any commands that need to communicate information between the client and server:*
    - Open a new terminal and move into the server directory with cd server. Then, run ./WTFserver <port number> where port number is specified by a .configure file in the client's directory.
    - Do not need to do this for configure, add, or remove to work
    - After establishing a connection between the client and server, this command does not need to be called again in between two operations that need to communicate between the client and server
- In general, running "./WTF <args>" on the client side with the appropriate arguments will connect the client to the server and run the selected command.

Output:

- In general, after executing "./WTF <args>", the repository is updated with changes that are stated in the assignment description.

Thread Synchronization:

Everytime a thread is created, which occurs whenever a client connects to the server, we add this thread to a linked list of threads. We use a linked list of threads in order to keep track of every thread so that at the end, when the server receives a SIGINT signal, every thread that was created during the duration of the server's existence is appropriately "joined" (ended). We use mutexes to lock and unlock the entire repository whenever we need to call a function because it could have its shared data modified by a different client running commands separately. It made more sense for us to lock the entire repository rather than each individual project because some commands make changes to the entire repository, so it is best to be safe and just lock the entire thing. This ensures that no shared data is changed during these operations where we need the project being changed to not be affected by other clients.

File Structure:

The .Manifest metadata file has its version number stored as the first line. All subsequent lines represent files in the project and metadata about them. Each line has a file's status, path name, version number, and hash. These are all separated by tabs so that when parsing through the manifest we are easily able to assign these values into fields of a node. A similar format was

also used for the .Commit and .Update files to ensure that we could use the same function on all these files to convert them into linked lists.

The .History file contains a log of all operations performed on a project on successful pushes of the project. It has the version number of the project and the operations performed during the push, all separated by newlines. It does not keep track of if the project was rollbacked to a previous version however, and instead just adopts the .History file that the specified version of the project had.

Data Structures:

We used linked lists to make it easier to parse through the metadata files in this project. This included converting .Manifest, .Commit, and .Update into linked lists. The linked lists had the head represent the version number of the .Manifest, the clientid of the .Commit file, or a "0" for .Update simply so we could use the manifest_to_LL function on it as well. All the following nodes of the linked list contained the status, path name, version number, and hash of the file corresponding to the node.

We also made a linked list of FileNodes that stored a file's name, its size, and its contents. This was used to make it easier to search through relevant files and find out information such as what the contents of these files were.

We also used a linked list to store a list of pending commits. This list is made up of CommitNodes that store a project name, client ID, and the contents of the .Commit file. It is used to see if the server has a pending commit that matches with the commit the client requests, and to expire any pending commits of that project during push and destroy.

The linked lists can also be used to write back modified data to a metadata file. For example, we use the LL_to_manifest function to write the newly modified manifest contents back to a .Manifest file.

Messaging Protocol:

In general, messages between the client and server are prepended with a two character string representing how the message should be interpreted. They have the length of the information to read from the message followed by a colon and the actual information itself. We could not simply use a colon as the delimiter to separate information in the message in case the information being obtained from the message contained a colon as well. Using this protocol allowed us to send messages of dynamic sizes between the client and server.

Final Notes:

Our project is fairly efficient overall; by using the same structure for all of our file contents, it is very easy to make modifications to files based on changes. It appears that our program works in O(n*s) time, n being number of files and s being being the size of each file. This is because our program is based on traversing through a buffer and tokenizing.