Samantha Lee (sll155), Matthew Notaro (myn7)

Professor Francisco

Systems Programing

5 May 2020

Assignment 3

Description: WTF is a version control system that stores directories. It contains a multi-threaded server with which multiple clients can interact with, push projects to, get projects from, etc. The server itself keeps track of the history of the project, whereas a client has its own local version.

Command Implementation:

- Configure

Configure saves a .Configure file that contains a host name and IP address. If .Configure already exists, it is updated with the new host and IP. All commands that require server connection rely on setServerDetails(), which reads the host and IP from .Configure and sets the global variables PORT and HOST accordingly.

- Checkout

If the client has already configured and if the project requested does not already exist on the client, the command "checkout:<project>" is sent to the server. The server finds the project directory and calls addDirToLL() to create a linked list containing all files in the project. Then, transferOver() sends the data to the client, following the format "<number of files>:<file name

length>:<file name><file data length>:<file data>...” The project directory is created and a loop reads in the information for each file and adds the files to the project.

- Update

The client connects to the server and receives the server’s .Manifest data, making a temporary file that holds the data. If the client and server .Manifest versions match, the .Conflict and temporary files are removed. Otherwise, a loop examines each entry of the client .Manifest and extracts its version, file name, and stored hash. A nested loop compares the current client entry to every server entry. If a matching file name is found, the file’s live hash is compared to its stored hash, which determines whether a conflict exists (in which case an entry is added to .Conflict) or if the server has modified code (in which case an M entry is added to .Update). If the file is not found, then a D entry is added to .Update. Afterwards, a similar nested loop compares the server’s entries to the client’s, and if the server’s .Manifest has an entry whose file name does not match any within the client’s, An A entry is added to .Update. When the loop is finished, the temporary file is removed.

- Upgrade

Upgrade applies changes listed in .Update to the local copy of the project by extracting each line of .Update — if the entry is flagged with M or A, addFileToLL() adds a file node to a linked list. If the entry is flagged with D, its corresponding entry in .Manifest is deleted (a new .Manifest is created, all text before the entry is written, and then all text after the entry is written). Data from the linked list is then sent over to the server in the form “upgrade:<project>:<number of files>

:<file name length>:<file name>...” The server adds all requested files to a linked list, and sends the linked list data to the server in the form “<number of files>:<file name length>:<file name><file data length>:<file data>...” The client populates the project with files.

- Commit

The client functions in a way similar to how it does for update. After receiving the server’s manifest, nested loops compare entries, adding M, A, and D entries to .Commit accordingly. Then, the client sends the .Commit through a message with the form <length of .Commit data><.Commit data>. The server then reads the information and saves the .Commit to a list of active commits.

- Push

The client sends a hash of the project’s .Commit to the server, which checks if it has that .Commit saved in a .History file, which details previous pushes through entries that include project version numbers and hashes of .Commit. If it does, then it builds a linked list with all the files to be added or modified and requests the files from the client. The client loops through the names of the files and builds its own linked list along the way, storing the files’ information. It then sends the data to the server in a fashion similar to the ones outlined in previous descriptions. The server updates its files in accordance with .Commit and updates .History as well as .Archive, a directory that stores .tar files of old project versions.

- Create

The client sends a request to the server, whose response determines whether the project already exists on the server. If it does not, the client creates a project directory and creates a .Manifest whose version number is 0. The server does the same, creating an additional directory for archived project versions.

- Destroy

The client sends a command including which project it wants destroyed to the server, which recursively deletes files in the directory, then the directory itself. The server then sends confirmation to the server that it either succeeded in destroying the project or failure if the project did not exist.

- Add

The client checks if the file to add exists locally. If it does, it appends an entry to .Manifest for the file, starting at version 0 and including a live hash.

- Remove

The client checks if the file exists locally. If it does, the client finds its corresponding entry in .Manifest and deletes it by storing all text before the entry and all text after the entry, creating a new .Manifest, and writing the stored text.

- CurrentVersion

The client requests all of the files along with the version numbers contained in a project from the server. The server then opens its .Manifest, reads through it, and formats the output to ignore the project version number and hash codes for each of the files before sending the message to the client. The client then outputs a list of all files under the project name, along with their version numbers.

- History

The client requests a project’s history from the server, which opens its .History, reads through entries detailing previous pushes, and formats the intended output before sending the message to the client. Upon a successful push, the server appends the current version’s number to the .History file along with the .Commit file received from the client. The client then outputs a list of all changes.

- Rollback

The client sends a message to the server requesting a project to roll back to a previous version. The server locates the older project in the .Archive directory, moves it into the working directory, deletes the project directory, extracts the contents of the tar file, then deletes the tar file itself. Finally, the server sends confirmation to the client upon success, failure due to invalid version number, or failure due to invalid project name.

Thread Implementation:

Whenever a new client is accepted as a connection to the server, the server creates a new thread that then determines which command to execute based on the first delimiter-separated segment of the bytes read in from the socket. The intended project is then read in from the client socket to determine if the server should lock/unlock the appropriate project mutex. The code contained within the mutex is minimized to only the code that accesses/reads from/writes to the project directory and any of its contents so that the number of threads are running at the same time.