Asst3 Documentation

With Asst3 being a complex project, there are a lot of parts working together to make the project one cohesive application of a mock-git functionality. One key aspect of the project is how we set up and use the manifest file to keep track of different project directories and manage the information for various commands. Our manifest file was straightforward, the format being <file version number> <tab> <project name/path><tab><hash code><newline>. The hash code was created by using SHA-256 hash generation on a string and inserted into the manifest for each file entry. An implementation of a linked list was also used to be able to traverse the manifest in an efficient manner, rather than scanning through the whole file every time. Instead, a linked list with a node consisting of the version number, file name, and hash code was used. The implementation of the linked list allowed us to more easily read and write between files and fetch metadata concerning various files and projects, names, version numbers, and hash codes. This functionality made many different processes in various commands a lot simpler to execute and saved a lot of time in our coding.

For communication to work seamlessly between the client and server, a bit of socket programming was needed. We first set up the socket on both the client and server side, using socket(), bind(), listen(), and accept(). The socket works like a file descriptor, and messages can be sent between the server and the client. A problem arose in that it is not only one client connecting to the server, but in fact it can be multiple client connections at the same time. This would not necessarily be an issue, but the conflict was if two clients were trying to edit the same file or project directory, which would be the first one and which the last? What if there were more than two clients accessing the same data? This is the case of deadlocking, where our program becomes out of sync due to multiple clients working on the same file. The issue was resolved by implementing mutex locks on each project. Mutex locks enable multiple threads to work on the same data, while avoiding the deadlock problem that arises. Thus, the multi-threaded server is capable of handling multiple clients at a time, and makes a more functional program as a whole.

Concerning the implementation of all the commands, there were various steps to each command, and they had to be completed systematically, while also accounting for thread-safe mechanics and ensuring no errors arise. The first command was config, which saves the IP and port of the client and server and makes it into a file. All commands that need to communicate with the server first reference this file before executing any other code. Checkout first sends a project name to the server. The server returns the manifest file for that project as well as all files under that project and the client builds the project up again locally. The way the server returns all the files is in the format of <number of files> <:> <file name size> <:> <file name> <:> <file contents size> <:> <file contents> <:> <next file>, etc. The client reads this giant string from the socket and proceeds to analyze it and do the appropriate file building. Update was a longer process to code, as there were many working aspects and necessary steps to arrive at the result. First the client requests the manifest from the server, the hashes in the local manifest are

recomputed, and then the necessary steps are applied for each hash/file-entry. There are four distinct markings, U, M, A, and D. If the client and server's manifests are the same version and the file entry is in the client's manifest but not the server's, or the client and server's manifests versions are the same and the file entry is the same in both, but the hashes are different, then the entry is marked with a U, signifying to upload. If the file entry is in both the server's and client's manifests and the version numbers are different, but the hashes are the same, the entry is marked with M, signifying to modify. For a file that is in the server's manifest but not in the client's and the version numbers of the manifests are different, the entry is marked with an A, signifying to add. For an entry that is not in the server's manifest but is in the client's and the server's manifest is a different version number, the entry is marked with a D, signifying deletion. All these marks and file entries are written out to an .Update file, apart from U. All these are then printed to STDOUT. Upgrade builds off of the previous update command, in that upgrade enacts the changes written in the .Update file. Entries marked with D are removed from the local version. Entries marked with M/A are requested from the server, as well as the manifest. After the client receives these from the server, it proceeds to rebuild the files on the local side. M signifies the client to replace the existing file and A signifies the client to add the file. The local (client's) manifest is then replaced by the one sent by the server, and the local .Update file is deleted, with the necessary changes being made to the project. First the Commit starts off by the client requesting the server's manifest file. After receiving the server's manifest for a given project, the client remakes all the hash codes for the files in its local manifest. Then commit builds a .Commit file with the following changes to be implemented. The hashes between the local and server manifest are then compared. For each hash/file entry, if it is in the server's manifest but not in the client's, then the file is to be deleted from the manifest, marked with a D. If it is not in the server's manifest but is in the client's, then it is marked with an A, as it is to be added to the manifest. If the file/hash is in both the server and the client, but the client's version number is higher, then the file needs to be updated in the server's manifest. After writing all these markings and updated hashes to the .Commit file, the client sends it to the server. Finally, the server sends back a success message to the client stating that the commit was a successful operation. Push was one of the bigger commands to be written in the project, and rightfully so as it is a large-scale operation requiring the sending of multiple files and lots of information. Push begins with the client sending the .Commit file from commit to the server, along with the file contents of those files marked with a U or an A. The server is then responsible for making the changes marked in the commit file to the project/manifest. The server starts by copying the current project directory over, saving the previous version of the project before the updates are implemented. The way the previous versions are saved is that we compress the old version of the project using system calls to then compress into a .tgz file and then move it into a hidden directory titled prevVersions. Then the files marked with a D are removed, and the files marked with U/A are added/modified. The manifest is then updated to reflect the changes made, with the version number of the project being incremented as well. The server then sends the new manifest file to the client. The client takes the server's manifest and replaces the local manifest with it, also deleting the local .Commit, as its changes have been made by the server. Create and destroy are simple commands and are similar in their workings. The project name is sent to the server and it is subsequently created or destroyed, respectively. Add and remove are also simple in that

the file name given is either added or removed from the local manifest. Currentversion requests from the server the current state of the project name given. The server sends over a list of all the files within the project name along with their version numbers, and the client simply lists them to the screen. History starts by requesting from the server a project name's history. Namely, the server sends over a file that contains the history of all operations completed upon the project (on successful pushes) since its creation. The result is listed to STDOUT in a similar fashion to update, but with version numbers and newlines separating each push's log of changes to the project. Rollback is a straightforward command. A project name and a version number are given, and the server reverts the current version back to the version number specified in the parameters. All versions between the current version and the version number requested are deleted off the server's hidden directory prevVersions, where all previous versions of a project are stored (one is stored every time a push is made).

All in all, the complete functionality of a mock-git application required a lot of different working parts coming together. There were many commands and the foundation of the project was built on socket programming and understanding how to work with threading/files. Careful notice was given to avoid any deadlock conditions, as that would ruin the purpose of having a multi-threaded server. Mutexes were the main contributor in avoiding these deadlock conditions, and were a very important aspect of this project, ensuring that a multi-threaded server that accepts multiple clients while simultaneously accessing the same server repository and project directories was even possible.