Assignment 2

Karan Amin
Saavi Dhingra

Intro:
This project began through basic communication between the server and the client. By connecting them through the port number, we were able to send and receive messages. Based on the knowledge of this, we implemented additional features, which included a multithreaded server. This ensures that multiple connections can be formed, and that multithreading is taking place.

This project is a basic simulation and implementation of git and other version control systems. While it does not have functionality like branches in github, it simulates the process of having a central repository where all the projects are stored and maintained. With a main central repository we have a canonical main version of every file. Multiple clients can connect to the main central repository and push their changes so that other clients and pull those changes and work on their version of the project. This way multiple people can work and contribute to the code while making sure they are working on the same version. The central server is multithreaded which greatly improves performance while allowing for multiple clients to connect all at the same time. The server makes sure no two clients access the same project by using mutexes to lock and unlock separate projects. A detailed explanation of our design principles and implementation is described below, but first let's describe how to use our version of git.

On Start:
First we will describe what happens in the main function of our server and client. They both initialize a TCP socket but the server does it in 4 steps while the client does it in 2. The server first creates the socket, bind it to the port number given as the second command-line argument, calls listen which prepares the socket to accept connections and become a server socket. Listen also specifies a backlog of 100 connections requested to be queued before any further requests are refused. We could have chosen a backlog of 1 but then making the server multithreaded would be useless because the server cannot accept any new connections. We chose a backlog of 100 as a safe number so that our server will almost almost never fail to accept a client. Once listen is called, we call accept and wait for a client to connect. Once a client connects, we create a new pthread_t and malloc some memory to hold the client socket that the server will receive from accept(). Once that is done, it will create a new pthread passing in that socket and a function handle_connection() to handle that connection. We do not join the thread with pthread_join because the server will continuously be running in the background since it will be accepting connections in an infinite while loop. The only way to stop the server is to kill the process by giving the program a SIGINT or a SIGKILL signal.

Now let's talk about the client side and how it starts up. When the client is started it first checks if the command-line argument given to it is a valid one. A valid one meaning it is specified and described in the project description. If it deems the arguments to be invalid, it will immediately terminate the process. If it deems the command to be valid, it first checks if the command was “configure”, if so it calls the configure function which will create the .Configure file we will describe later. If the command given was not “configure” then it will check if it was either “add” or “remove”, because these two commands don't require the client to connect to the server, therefore it can do it’s functionality without creating a TCP connection. If the call was neither of those, it will connect to the server in 2 steps. First it will open the .Configure file stored inside it’s repository. If it cannot find a .Configure fail it will print an error and tell the client to run configure first. If a configure file was found, it will extract the IP address and port number specified inside the file. It will then use them to first create the socket and then create the sockaddr_in struct and initialize it with the IP and port numbers. Once that is done it will call connect to attempt to connect to the server, if it succeeds it will call handle_connection to handle the connection.

Configure:
Now let's talk about the first three commands that don’t require the client to connect to the socket. The first of which is configure. This command is fairly simple and straightforward. It will create a .Configure file inside the client repository if it doesn’t already exist and empty the file if it does exist. Then it will first write the 3rd command-line argument which contains either the hostname/IP address and it will copy it over to the .Configure file, followed by a space, and then the 4th command-line argument which contains the port number. Once both are copied over the file descriptor is closed and the program is terminated.

Add:
This command is also very simple and straightforward. As arguments it will receive the project name and the filename/filepath given by command-line arguments. It will first check if a project with that project name exists in the client repository. If so, it will check if there is a .Manifest file inside the project. If both are true then it will attempt to add the file to the manifest. It does this by creating a string that we will insert into the manifest file. The first character in the string will always be a 1 because every file that is added to the manifest will begin on version 1. Then the project name is appended to the string followed by the filepath of the file. Once all those are done it will call a function to calculate the hash of the file, which for this project we will use SHA1. It will append that hash to the string encoded in hex. Once the string is appended to the manifest it will close the manifest and end the program. A couple things to note here is that the file path given must either begin with a “./” so ./WTF add project1 ../project1/filename.txt would be a valid command. The file path can also exclude the “./” and it would be fine. It will also work if you just give the filename instead of the relative path. So for example ./WTF add project1 filename.txt would also work, but ONLY IF the filename.txt is immediately under the project name. If the file is in a sub directory inside the project it will fail. Just to avoid ambiguity use the relative file path for this argument. Also note that the file must exist for add to work because the client needs to compute the hash for the file in order to add it to the manifest. Also make note that multiple calls to “add” of the same file will overwrite the previous entry for that file with the new entry with a version number 1, and the live hash of the file.

Remove:

This command is also pretty simple and straightforward and follows similar steps as add. It first checks if the project exists on the server, if not, the program will exit. If it does, then it will check if the .Manifest file exists. If both are true it will call a function which will delete the entry for that file. Note that as opposed to “add”, “remove” does not need the file to exist in the project. The function will simply use the filename/filepath and the project name to find the line in the manifest which contains both of these and delete this line from the manifest. Also note that the same rules apply to the convention for file names as they did for the “add” command. Once the file is removed from the manifest the file descriptor is closed and the program terminates.

Handle_connection:
Both the server and the client handle this command very similarly, but the server goes through it a little differently. This function receives the socket it will use to communicate to the other side as an argument. The first thing the client sends is the project name. Once the server gets the project name, it will call a function to choose which mutex to the server should lock. This mutex is chosen by the following algorithm.

To pick a mutex to lock, the select_mutex function receives a project name as argument. Then it searches a global linked list which contains a [project_name, mutex] pair for each node. If the project_name finds a match it returns the pthread_mutex_t associated with that project name. If it doesn’t find a match it will create a new node for the list and initializes a new pthread_mutex_t and stores it in the node in the linked list, and then returns the mutex it just created. This works so that next time a client calls a command on a project_name the mutex associated with that project has already been stored in the list and the function can simply return that mutex. One other special thing we implemented is that when a call to this select_mutex function is made, it locks a global command mutex pthread_mutex_t list_lock. This mutex named list_lock locks another other thread from trying to check the mutex list to find a match. We need to do this because let's say two threads check for a mutex associated with project1 and project1 simultaneously. It finds that there is no match for the mutex in the list and attempts to create a new node and add it to the list. This creates a race condition where both threads are trying to create a new mutex for that project and add it to the list. If both threads attempt to add it at the same time then one of the nodes may become disjoint. This also creates the situation where both threads create two seperate mutexes for the same project and then proceeds to work on that project while another thread also works on that project. This global list_lock mutex prevents this race condition from exposing itself. It also helps in later commands like “push” when we need to “lock” the repository so that no other command can be run. When we run push we can lock this list_lock mutex so that no other thread can select a mutex for a project therefore not run their command before push unlocks list_lock. This implementation of selecting a lock also avoids deadlocks. Deadlocks can occur when thread A locks mutex A, and thread B locks mutex B, thread A tries to lock mutex B, and then thread B tries to lock mutex A, since neither A or B can be unlocked since they are waiting for the other to be unlocked, both threads are in a deadlock. This problem can be mitigated or avoided by establishing some kind of ordering or hierarchy in which locks are locked. If you want to lock the mutex above you in the hierarchy you must first unlock your lock or have the lock above you guaranteed to be unlocked, but if you want to move down in the lock hierarchy you can lock without having to unlock your mutex. This "lock hierarchy" is created by list_lock being on top and all the project mutex locks being immediately underneath it. Since our program selects locks in a specific order every time, this hierarchy is maintained and deadlocks are avoided. When a thread locks list_lock, it will always unlock list_lock at the end of the function before calling any project mutex. When "push" locks list_lock it will always unlock list_lock without calling any other project mutex. Even if any thread fails to run a command it will always unlock the project mutex it was given or unlock list_lock and the project mutex in the case of "destroy" and "push". Hopefully this algorithm for selecting a mutex made sense and wasn't too confusing because it took a while trying to come up with a safe method to avoid deadlocks and choose appropriate mutexes for given project names.

Now that we have discussed how our server is multi-threaded and how thread synchronization is acquired, lets move on to explain how we implemented every command. Now that handle_connection has received an appropriate mutex for the given project name it will lock the mutex given and receive a command from the client, indicating which command to run on the server. It uses that command string to call the appropriate functions to perform their operations. Once the operations are complemented and control is returned to handle_connections, it will unlock the mutex, close the socket, and exit the thread. Now let's move on to show each command separately.

Checkout:
This command first checks if the project name exists on the server, if it cannot find the project it will notify the client by writing a "fail" message to indicate the error, and returns to handle_connection. The client also checks if it contains the project, if it does, it will notify the server that the project already exists and the server will unlock any mutex and exit the thread. The server then checks if there is a .Manifest file inside that project, if not it notifies the client and both end their connection. From now on whenever we say either the server or client failed to do something, it is assumed that it will notify the other side that it failed and both will terminate their connection gracefully, and we won't mention this process again to avoid this readme from being too long. If there are no errors, the server will prepare to send a stream of formatted data similar to the format specified in the project description. The server will open its manifest for that project and parse the file. Every file listed in the manifest will send it to the client to copy. It will first send through the socket the number of files the client will need to create, then the server sends the filepath of all these files. The client will receive the file path given and create all the directories and subdirectories it needs to create the file. Once all the file paths are sent and the client creates the empty files, the server will send the bytes associated to each of these files, and the client will take all these bytes and fill them in their respective files.The server then writes "done" to the socket and returns command to handle_connection to unlock the mutex, close the socket, and exit the thread. Once the client receives the "done" command it will do the same except unlock any mutex because it didn't create any.

Create:

Almost every command follows the same error checking principals when invoked. Create first checks if the project exists on the server. For the command to work, the project cannot exist on the server repo, if it does it will send a “fail” message to the client and both will end their execution. The client also checks if the project exists in it’s repository. If it does it will fail. Once all the error checking has been done, the server will create a directory with the project name given in its repository and insert an empty .Manifest file inside it. It then writes “1\n” to the manifest to initialize its initial version to 1. It also creates an empty .history file. This .history file will store every successful operation performed to the project during a “push” command. This file will be used later in the history command to print out the history of the project. Once these two files have been created, the server and client both call the checkout method. It does this to reuse functionality already implemented. Since the checkout command takes an existing project on a repository and sends it to the client, this function can be used to send the newly created project to the client. The implementation of checkout has been described above. Once the project has been sent over, both the client and server will return control to handle_connection and end their connection gracefully.

Commit and Push:
Now let's talk about one of the main commands that make git so powerful, commit and push. First let's talk about how commit works and then we can move on to discuss how push is implemented. The client first checks if the project is in its repository, if not it tells the server to disconnect. If it is in the client repository it tells the server to commit. The server then does the same thing and checks if the project AND the if there is a .Manifest inside that project. If at least one of them doesn’t exist it notifies the client and both fail. The client also checks if it has a .Manifest inside the project, if not it tells the server to fail. The client does further error checking to see if there is a non-empty .Update file or if a .Conflict file exists in its project. If either is true it fails. Once all the error checking has been done, the client sends a “success” message to the server, and the server moves on to send it’s .Manifest file to the client. The client receives the manifest as a stream of bytes. The main way the client compares the client and server manifest is by creating two linked lists containing the entries of the server manifest in one and the entries of the client manifest in the other. Each node contains the version, project name, filepath, and hash all given by reading a single line in the manifest file. Once the contents of both the client and server manifest have been copied into separate linked lists, the client will start to compare the entries. The last node in the linked list contains the version of the manifest so the client first traverses to the end of both linked lists and checks the versions. If the versions do not match, the client will print out an error indicating the manifest versions do not match and fail. If the versions do match, it first saves the current version of the manifest in a string in case the client fails for some reason later and needs to restore the manifest to the version it was on before. It then creates a .Commit file and appends the current version of the manifest but increments it by 1. The push command command will read this to update its manifest to the new incremented version. The client then compares each node in the client list to each node in the server list. If it finds two nodes that match with the same file path and project name, it checks if the hashes match, if those match then it checks the live hash of the file in the client. If the live hash is different that means the file has been edited and the change needs to be committed, so the M

operation is written to the .Commit file. If the live hash is the same, the file is ignored. If a project name and file path match is found but the hash does not match it compares the file version. If the file in the server manifest list is on the same OR higher version, then the commit command fails, the .Commit file is deleted, the old version of the manifest is restored, and the client fails. If a node in the client list is not found in the server list, that means the server manifest needs to add this new file to its project, so a “A” command is inserted into the commit file. Once all the files from the client list are compared, one more search is performed through the server list. For all the files that didn’t have a match with something from the client list we write a “D” to the commit file because the client removed that file from its manifest and we need to reflect that change in the server manifest. All these rules are described in the project description. Once everything has been compared, if the .Commit file is empty (except the first line which holds the new version), then the file is deleted and the old version of the manifest is restored. After that the client sends this .Commit file to the server and the IP address of itself. The server takes this IP address and creates a file with the same project name but appends the host name to it like such, <projectname>_<host>. Once this file is created, the contents of the .Commit file are copied into it. This file is how the server saves active commits. It uniquely identifies which client that active commit belongs to, this is why we need to make sure that two clients run on two seperate machines to allow for two clients to have two different active commits stored in the server repository. Once that is done both clients are done with their operations and control is returned to handle_connection and both terminate gracefully.

Once commit has run it is time to push those changes to the project on the server. The client does some error checking first, by checking if the project exists in its repository, if not the client fails. It also checks if it has a .Commit file, fails if it doesn’t. The server then locks the repository by calling the global mutex list_lock which prevents any other thread from trying to select a mutex from the mutex list to run their command. Calling list_lock prevents another other client from trying to access the repository but it doesn’t prevent the commands that are already running. That is why the thread then goes to sleep for 1 second, which allows any other thread to execute their command and unlock their project mutex before push influences the repository. The server then proceeds to check if it has the project in its repository, if not it fails. Then it proceeds to check if it has an active commit for that client in its repository. It does this by allowing the client to send its host information IP address so that the server can compare it to see if there is an active commit that matches the project name and the host. If so it checks if the commits are equal by requesting the .Commit file from the client and comparing it to the active commit it has stored. If the commits are different the command fails. If they are the same, a function is called to perform the push. This function applyCommit first reads the first line from the commit file, which holds the new version of the manifest. It updates its own manifes’s version to that version and then starts reading the next line in the commit file. But before updating its own manifest it first duplicates its project to a folder with the same name but the old version of the manifest appended to it like such, <projectname>_<oldversion>. This folder saves the state of the project before doing any pushes so that if a client calls rollback later on it has a saved version of the project. Once the directory is duplicated, it opens the .history file for that project and starts reading lines from the commit file. For every operation labeled M or A, it requests the file from the client, and creates any directories it needs to create the file, then overwrites the file with the bytes from the file on the client. If the operation was “A” it says the add function (same function as in the client but copied over to the server source code) to add its manifest. Once added it updates its version to the version stored in the .Commit file. The new versions of every file is stored as the last token in any given line in the .Commit file. If the command was a “D” it deletes the entry from the server manifest. Once all the operations from the .Commit file are performed the file is deleted from the server and client and control is returned to handle_connection to end the connection gracefully.

Update and Upgrade:
To keep this readme relatively short, we will not make the description of update and upgrade too detailed. Both the client and server both do some error checking before during its operations. The client first checks if it has the project on its server. If it does it checks if there is a .Manifest file inside it. The server also checks the same things and if either one doesn’t exist the command fails. Once all the errors have been checked, the server sends its own manifest to the client similar to the commit command. The client takes this .Manifest and creates the same manifest linked list as it did in commit. It uses this linked list to do all its comparisons as well. Once both linked lists have been created, the client traverses to the end of the linked list which stores the file version. If the file version is the same, it prints that the manifest is on the same version. If the manifests are on different versions, it creates an empty .Update file and a .Conflict file. It compares each node from the client linked list and the server linked list and using the rules from the project description it decides if a file should be “A”, “M”, “D”, “C” or ignored. If there are any C or conflicts, the file is inserted into the .Conflict file. After comparing every file if there are any conflicts, it deletes the .Update file and both the server and client end their connection gracefully. If there are no conflicts then the .Conflict file is removed and then both client and server end their connection.

Upgrade is the next command. Error checking is almost the same for every command. The client checks if it has the project on its server, a .Manifest inside that project, an non-empty .Update file, and no .Conflict file. If any of those are not true the client fails and the connection is terminated. If there is a .Update file but it is empty, the client just prints “up to date” and terminates. The server also checks if it has the project and the manifest for that project. Once all the error checking is done, the client opens the .update file and reads the first line. The first line in the .Update contains the new version of the manifest, so the client updates its manifest version first. After that it starts reading line by line and using the same rules as described in push the client creates files and edits or deletes from the manifest depending on the operation described in the .Update file. Once all the operations are done, the client and server manifest should look the same.

Destroy:
These next few commands are pretty simple and don’t need much explaining because their implementation is pretty linear and logical. For this the client does not check if it has the project because it doesn’t need to. The server does check if it has the project and if it doesn’t it will fail.

If everything is good, the client sends its host name to the server so that the server can expire any pending commits by deleting those files. Once those files have been destroyed, the server calls a recursively destroy function which accepts a directory path to empty out the directory using rmdir and remove commands. Once the project has been emptied the server deletes the project name and the server ends the connection since the project has been destroyed.

Currentversion:
This command is also very simple and can be described in a couple lines. The server checks to see if it has the project, if not it fails. It then checks to see if it has a .Manifest file, if it doesn't it fails. If both exist, the server sends the .Manifest file over line by line. The client receives this data and prints it out to stdout excluding the manifest version and hash for each line. Once the manifest has been printed out both the server and client terminate successfully.

History:
Goes through the EXACT same process as currentversion but instead of the manifest file it changes it to the .history file. The client simplify prints out this file.

Rollback:
This is the last command we have to describe. The client does not need to have the project in its repo. The server takes the project name and version to roll back to as arguments and calls the perform_rollback function. This function opens the server repository and searches to find a back up folder with the same name as specified and the same version number. Remember that a backup of a project has the naming scheme <projectname>_<version>. Once it find the folder, if it exists, it calls another function delete_all_higher_rollbacks whose implementation looks pretty much the same as perform_rollback but the difference is that if it finds a backup folder with the same name but a higher version, it uses the function in destroy to destroy that folder. Then perform_rollback will destroy the current version of the project and rename the backup folder it just found to the project name (aka without the _version) appended. Control is then returned to handle_connection for both the client and the server and the connection is terminated gracefully.

Files included:

- Asst3.tgz
    - Asst3
        - readme.pdf
        - testcases.txt
        - All source code
            - WTFserver.c
            - WTF.c
            - WTFtest.c
        - testplan.txt
        - Makefile

Compiling the program:

To compile the program you need to use your terminal commands where you initially create the server and client. The port numbers need to match in order for the server and client to be simulated at the same time. Running make will create the WTF and WTFserver executable. If you want to run our test cases for demonstrative purposes, a WTFtest executable is also created.

The server only takes in a single command line argument, which is a port number to listen on.

- The server is created by running **./server <port number>**

When creating the client, we need to provide the hostname or IP address of the machine on which the server is located along with the port number

- The client is created by running **./client configure <IP address> <port number>**

There are 13 different commands that can be used by the client:

1. Configure**: ./client configure <IP/hostname> <port>**
2. Checkout**: ./client checkout <project name>**
3. Update**: ./client update <project name>**
4. Upgrade**: ./client upgrade <project name>**
5. Commit**: ./client commit <project name>**
6. Push**: ./client push <project name>**
7. Create**: ./client create <project name>**
8. Destroy**: ./client destroy <project name>**
9. Add**: ./client add <project name> <filename>**
10. Remove**: ./client remove <project name> <filename>**
11. Current Version**: ./client currentversion <project name>**
12. History**: ./client history <project name>**
13. Roll Back**: ./client rollback <project name> <version>**

In the cases where we have **<file name>**, we need to make sure to put in the path name in place of it. The command needs to know exactly where it is adding the file. For example, if the client wants to use the add command, he/she needs to make sure to enter the command as follows: **./client add project1 ./project/file1**

THE END