Here's a redone version of system's final project description from someone who's done this project in the past – but on Word because I can't use LaTeX. Not everything on here is going to translate Franny-nese correctly but I'll try my best – **this means that you should always check the original description for anything I didn't translate clearly or ask on Piazza**. Some descriptions are longer because I like giving suggestions or being more verbose.

Please make sure to read the Methodology (at the bottom) before doing anything.
Side note: for those of you who are just starting to read the assignment description good luck, because you'll need it with the amount of time you have left.

0. Abstract

This project is just a more convoluted Git. If you don't know how Git works, I suggest learning the basics before starting.

The gist of this project is to do version control (essentially what Git does). What is version control? Something you should've been using this entire semester for your projects because this is a partner-based class (yes sharing code through email is the wrong way to do things, <insert name>). But the basics of it are – a server contains a collective version of your repository, and you and other programmers have your own version of this repository locally. In order to keep version of the code the same amongst the group, you upload (push), and download (pull) each other's versions of the code to and from the server (safely). That's a super simplified explanation but the rest of the project description should explain how to do this more clearly.

1. Intro

Vocab:

- Project: a directory that contains code, binaries, etc. for a, well… a single project.
- Repository: contains one or multiple projects. This includes metadata for projects and backups/history (I believe for each project).
- Manifest: the metadata which contains all paths to all files for a given project, the version of all files within the project, and the version of the project itself.
- History: a list of all changes to a project. Also serves as backups.
- Roll back: go to a backup version
- Commit: changes that have been made to something in the repository are claimed to be "finalized" (not really) on the client side. More info on this in the commit section.
- Push: uploading the changes that were committed.
- Checkout: akin to Git clone. More info in checkout section.
- Update: akin to Git fetch. More info in update section.

2. **Implementation**

.Manifest

Every project has this in its root directory, meaning the project directory itself.

The Manifest contains the following:

- Project version, this is incremented on every *successful* commit (in a .commit file - again more on this in commit section) and finalized on push (written to the .Manifest)
- The path to each file/name (ie. If the file name is A.txt, then projectA/A.txt is the full path)
- The version of each file, again incremented on every *successful* commit and finalized on push
- The hash of each file

How this is formatted is your choice. Do whatever is easiest to keep this part hassle free.

The Server

The server is responsible for many things.
First, it must keep a history – for the entire repository with previous versions of each project stored in this history. You could make a history directory for each project instead but just be careful when managing your .Manifest file since the history should **not** be in the manifest file.

Second, it needs to be reliable. This means multiple things:

- The server needs to be thread safe – it should lock the repository **or** the project being called on by a client so that changes from one client don't get overwritten by another by accident. Commands that are no longer valid from other client(s) should be cancelled by the server and let the client(s) know. **Hint: the answer to locking is not always the repository because locking the entire thing isn't efficient, but I'm going to be honest, you could just have one mutex over the entire repository because it's really hard to test for individual project mutexes with how fast the iLabs are. This isn't the correct way to do it, buuuuut… it's the least stressful way. Only do this if you are short on time/desperate. You didn't get this suggestion from me :quietpepe:**
- The server also needs to keep records of the most recent commits being made and update/store the newest version reliably (no files should be missing or corrupt). This obviously goes for old versions as well (history).
- It should also communicate properly with the client if anything goes wrong.

The Client

Your client should manage its own version of the repository. This means it should know what version it's on, changes that have been made, and what needs to be uploaded or updated. It needs to make sure that any updates made are safe, meaning it should not override new changes to the repository unless the user specifically wants that to happen.

To invoke the client the user must be able to do ./WTF *command* <required arguments> <…>

Messages Your Client Should Display

Your client is responsible for telling the user what went wrong for anything that goes wrong:

> Minimally, your code should produce the following messages:
> - Client announces completion of connection to server.
> - Server announces acceptance of connection from client.
> - Client disconnects (or is disconnected) from the server.
> - Server disconnects from a client.
> - Client displays error messages.
> - Client displays informational messages about the status of an operation (another operation is required, aborted and reason)
> - Client displays successful command completion messages.

Connecting to the Server

In the case that your client cannot connect to the server the first time, it should make an attempt to connect every 3 seconds until it is killed (via kill or kill -9 <pid> which can't be caught) or exited with Ctrl+C (SIGINT which can be caught).

Client Functions (and the effects they have on the server)

Note: for any function with file names, the full path to the file will be given (the assumption is the repository is the base directory but I would confirm this). I also suggest doing these functions in this order but that's just my opinion.

Configure (./WTF configure <IP/hostname> <port>)

Configure is responsible for connecting to the server at the given IP/hostname and port. Note this function **does not make an attempt to connect**. The IP/hostname and the port should be saved in a **.configure** file which should be in the root directory of the repository.

All other commands should fail if a .configure file does not exist. Configure never fails (*I think? He doesn't state otherwise).*

Create (./WTF create <project name>)

Create initializes (creates…) a project on both the server and client. What does this mean? The client should send a message to the server stating that a new project is being created locally with <project name> so the server should also initialize a new project with that name. The server is responsible for creating a .Manifest file and sending it over to the client. The client just needs to setup the project directory locally and store the .Manifest into it.

Create fails if the project already exists **on the server** or it can't connect.

The original description does not say this, but the server should lock the repository when creating a new project. It theoretically doesn't matter because the clients will have the same manifest anyway since it's a new project with the same name but why risk it?

Destroy (./WTF destroy <project name>)

Destroy removes an entire project from the server – and **only the server** *(client is mentioned nowhere in the original description and the client doesn't need a function to remove its own files. Just remove them manually…).*

Destroy fails if the project doesn't exist **on the server** and if the client can't connect.

The server is responsible for locking (thread safety) the **repository** and expiring any pending commits to the project.

Add (./WTF add <project name> <filename>

Add works closer to the actual add on Git. Since files on the client can be created or changed at any time, the .Manifest needs to know what files were created, which is the purpose of add. If the file did not exist in the manifest previously, an entry (see **commit** for what an entry is) should be added for the file with a version number (0) and hash code. If the file does exist in the manifest, then the entry should signify that it has been changed (check hint). **Do not** change the hash or version number. You will do that in **commit**.

This function **does not connect to the server. Hint:** it will be easier for you in **commit** if you use a code in the entry that signifies the addition or change of an entry (ie. 'A' for add, 'M' for modified). *Also mentioned in the original description.*

Add fails if the project does not exist client side.

Remove (./WTF remove <project name> <filename>)

Remove also works closer to the actual Git. Since files on the client can be removed just by doing *rm <file>* at any point, the .Manifest needs a way to know if a file was removed client side - and remove does exactly that. This function **does not connect with the server.** The client will signify in the manifest entry that the file has been removed…
… or not removed. Be careful with this. Remove does not require the file to not exist to be successful (since files are allowed to be left untracked), meaning as long as the manifest has an entry for that file, it will be successful, and when this eventually gets pushed, the server will just think the file doesn't exist in the client's copy. This is something the user should be wary of when calling remove (and not you) but on your end, make sure to actually signify the correct entry from the .Manifest and nothing else.

That being said, remove fails if the project does not exist locally. *No mentioning of this in the original description, but it should probably fail if the entry for the file doesn't exist in the .Manifest*
Hint: Same hint as add, you should make a code for this rather than removing the entry from the manifest file to make your life easier in commit (ie. 'R' for remove).

Currentversion (./WTF currentversion <project name>)

Requests the server for the ***server's*** version number of the project. This does not require the project to exist on the client side. The client should output a list of all of the files in the project along with the version numbers of each file.

Checkout (./WTF checkout <project name>)

Checkout is akin to Git clone. The server should send over the current version of the project to the client, and the client should make a *clone* of the project (.Manifest, subdirectories, etc.).

Checkout can fail for the following reasons:

- The project doesn't exist on the server.
- The client can't communicate.
- The project already exists locally (on the client).
- If configure was not already run.

Once again, it is not mentioned in the original, but make sure this is thread safe. You don't want the project to be changed while it is being read so that it can be sent over.

Commit (./WTF commit <project name>) *the descriptions get really long here to ensure clarity…*

Before pushing things to the server, you should always commit your changes. What does this mean? Any changes you make should be noted as "finalized and ready to be pushed." Otherwise the client won't know what it should be updating or pushing since everything would just be in a state of being changed, potentially. It's required to be able to tell what files are seen as final for merging or catching conflicts, otherwise the client won't know if the file should be overwritten or merged.

Let's define an **entry**:

- *A status (the code from the hint in add and remove)*
- File path
- File version number
- The hash code of the file

* The status should not be there for entries that do not need them. Only files that were added, changed, or removed should have entries with a status.

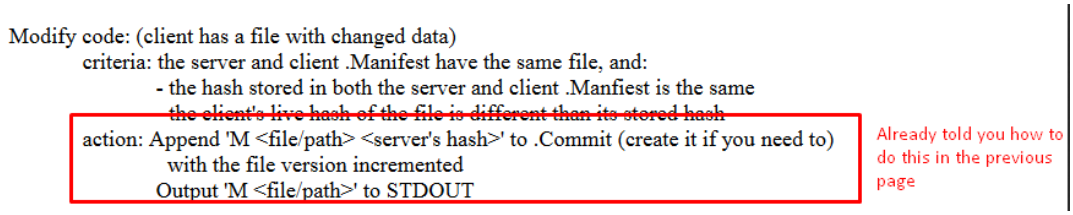
Commit should start off by fetching the server's .Manifest. The **project version numbers** (not files) should match in the server manifest and the client manifest. **Do not** increment the project version in the manifest in this function or any function above!!! Otherwise you will never have the same project version numbers. If the versions match, the client should create a .Commit file, scan its own manifest for entries that need to be updated (remember those codes mentioned earlier, this is where they make things easier. Look for those codes that signify added files, changed files and removed files rather than scanning through the entire entry).

- Any newly added entries should be written to the .Commit. You must output to STDOUT "M <file path>"
- Any changed entries should be written to the .Commit with an incremented **file version number** and a new hash code. **Do not** make increments or change hashes in the manifest file – only in the .Commit. You must output to STDOUT "A <file path>"
- Any entries that have removal signified should be written to the .Commit. You must output to STDOUT "D <file path> "

For all of these I would put the entire entry in the .Commit (version number, file path, etc.)
Hint: when writing to the .Commit, you might want to keep the codes you used so that it's easier for the server to know exactly what to do with each entry.

Things to consider:

In the original this section signifies successful commits. They contain an "action" bullet for how you should write entries into the .Commit, but I already told you how you should do that above in the hints. The results are the same so don't worry about it. Example screenshot below:



Changed entry:

- The server and client manifests should have the same hash code and file version number for that entry.
- The new hash code generated should be different from the one in the manifest.

Added entry:

- The server manifest should **not** have this entry, but the client manifest does.

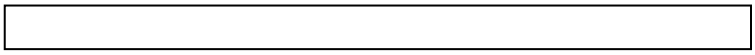
Delete entry:

- The server manifest **should** have this entry, but the client does not.

After all of this, you need to send the .Commit to the server and the server should save it and report success.

Commit fails for the following reasons:

- The project doesn't exist on the server.
- The client can't connect.
- The client has a .Update file that is not empty (not having a .Update is fine and should not make commit fail).
- The client has a .Conflict file.
- The project versions of the server and client manifests do not match.
- The file versions are not synced. I'm having trouble parsing this one (it's 5am) so here's a screenshot of the original:



If however there are any files in the server's .Manifest that have a different hash than the client's whose version number are not lower than the client's, then the commit fails with a message that the client must synch with the repository before committing changes. If the client's commit fails, it should delete its own .Commit file.

Here's some extra stuff to make this even longer but it's important and not mentioned.

The user can call commit multiple times before pushing. Instead of doing multiple .Commits, just check if it already exists and update it for one big commit file. The server should do the same on its end. Just make sure to manage the status codes correctly (ie. Don't accidentally add multiple status codes for one entry).

The server should have a list of .Commits from each client until a push is called. Let's take this scenario. If A, B, and C called commit (for simplicity they called it once each), there should be a .Commit for each person for that project. Let's say B called push, then the server should first expire the other commits and no longer store them (so when A and C call push, it fails), and then move forward with making the changes from B's .Commit file.

It should be known by now but this needs to be thread safe. The list of .Commits may need a lock so that it doesn't accidentally lose a .Commit from two threads linking the same node pointer to different new nodes or segfault when trying to remove .Commits (assuming you use a list. Should still lock the structure you're using). **Honestly just make your life easier and lock the project.**

The long descriptions don't stop here…

Push (./WTF push <project name>)

The push command is where all of you changes finally get made to the server.

The client should send its .Commit to the server. If the server has the .Commit file for this client, and they are the same, then the server should request the files that need to be changed (unless they're being removed). If there are other .Commit files pending, expire them so that push calls from other clients fail. Once the files are received from the client, the server should update it's own copy of the project with those files and the manifest file. On success, the manifest for both the client and server should increment the **project and file versions,** update hashes as well as remove the status codes on the entries that previously had them. The server should expire the .Commit after successfully updating the project and the client should erase the .Commit regardless of success or failure. Again thread safety, so lock the project.

Push fails for the following reasons:

- The project doesn't exist
- The client can't connect
- The .Commit does not exist on the server or the .Commits are not the same in which case, the user must call commit again to sync the .Commit files.

Update (./WTF update <project name>)

Update is similar to Git fetch. The client should get the server's Manifest and compare the entries in the server and client manifests to see if there are any changes from the server's copy. If there are, the client should make a .Update file which is similar to .Commit except this is changes from server to client instead of client to server. Similar to commit, for any changes noticed, you should write into .Update added, changed, or removed entries (with codes to make your life easier on what action to take for that entry). You must print to STDOUT in a similar fashion to commit for entries that need to be updated. Just like commit, I would put entire entries into the .Update file instead of just what the original description says.

In the case that there is an update to an entry but the client's manifest also signifies that the file has been changed (meaning no push), then create a .Conflict file and delete the .Update file. If there are no changes noticed, blank the .Update, and delete the .Conflict if there is one. Mutex locking isn't really necessary here since it's only the server manifest being sent.

I think these are explained pretty well above but sometimes it's nice to have a treeview of things rather than a paragraph *(pg. 5 of original since the screenshot is kind of small)*:

```
Full success case: (client won't have to download anything)
        Update code: (server has no updates for client and client may or may not have updates for the server)
                criteria: the server and client .Manifests are the same version ... can stop immediately!
                action: Write a blank .Update file because everthing is awesome!
                          (*dum*dum*dum*dum*dum*... everything is great when you're part of a team! ...)
                        Delete .Conflict if it exists
                        Output 'Up To Date' to STDOUT


Partial success cases: (client will have to download some things)
        Modify code: (server has modifications for the client)
                criteria: the server and client .Manifest are different versions, and the client's .Manifest:
                        - has files whose version and stored hash are different than the server's,
                          and the live hash of those files match the hash in the client's .Manifest
                action: Append 'M <file/path> <server's hash>' to .Update (create it if you need to)
                        Output 'M <file/path>' to STDOUT

        Add code: (server has files that were added to the project)
                criteria: the server and client .Manifest are different versions, and the client's .Manifest:
                        - does not have a file(s) that appear in the server's
                action: Append 'A <file/path> <server's hash>' to .Update (create it if you need to)
                        'A <file/path>' to STDOUT

        Delete code: (server has removed files from the project)
                criteria: the server and client .Manifest are different versions, and the client's .Manifest:
                        - does have a file(s) that does not appear in the server's
                action: Append 'D <file/path> <server's hash>' to .Update (create it if you need to)
                        'D <file/path>' to STDOUT


Failure case: (need to download some things, but can't because the user has made changes to the same files)
        Conflict code: (server has updated data for the client, but the user has changed that file locally!)
                criteria: ther server and client .Manifest are different versions, and the client's .Manifest:
                        - has a file whose stored hash is different than BOTH the server's .Manifest and
                          a live hash of the file
                action: Append  'C <file/path> <live hash>' to .Conflict (create it if you need to)
                        'C <file/path>' to STDOUT
                note: Don't stop if you hit a conflict! Keep going and find all updates and conflicts.
                      After scanning all of the server's .Manifest, be sure to output to STDOUT
                        that conflicts were found and must be resolved before the project can be updated.
```

Update will fail if the project doesn't exist, if the client can't connect, or one of the failure cases above.

An Aside: This part of the project use to have a ‘U’ for upload and the acronym spelled UMAD to which Franny made a joke about [literally had (U MAD, bro?) written in the description]. This and the fact that this project was released a day or two before April Fool’s made all of us think this was a joke because when we saw the assignment (that semester was the first time the assignment had been given) we immediately thought that the project was insane – and people were pretty upset about it when they found out it was real. The reason for the removal of UMAD is unknown other than maybe the fact that it caused a riot.

Upgrade (./WTF upgrade <project name>)

Upgrade is like Git merge (FYI, Git pull is just fetch and merge). The client will apply the changes in the .Update file created by **update**. It should request for files from the server if it needs them (which it does for modified and added files).

- Any entries marked for deletion should have their entries removed from the client’s manifest and deleted from the repository.
- Any entries marked ‘M’ should have the files overwritten by the one from the server and have its hash and version updated in the client manifest.
- Any entries marked ‘A’ should be added to the clients manifest and the file should be written to the project directory.

If the .Update is empty, then there isn’t anything to do so just say “up to date” and tell the server that as well. The server should lock the project so that nothing gets changed while the client is requesting files.

The suggestion here is to do deletions first because you don’t have to fetch files for that, then do add, and then modified.

Update will fail for the following:

- The project doesn’t exist
- The client can’t connect
- There is a .Conflict – the client should tell the user to resolve the conflicts first and update again
- There is no .Update - the client should tell the user to call Update first
- If for some reason the .Update becomes expired because the server was pushed to in-between update and upgrade calls, this *might* fail. *Not mentioned in the description so I wouldn’t worry too much about it.*

We're now officially more pages than the original

History (./WTF history <project name>)

The explanation to this is pretty good in the original so here's a screenshot:

The **history** command will fail if the project doesn't exist on the server or the client can not communicate with it. This command does not require that the client has a copy of the project locally. The server will send over a file containing the history of all operations performed on all successful pushes since the project's creation. The output should be similar to the update output, but with a version number and newline separating each push's log of changes.

Rollback (./WTF rollback <project name> <version>)

Make sure you the way you store backups works perfectly before doing this. The server will revert its current version of the project back to the previous version number and delete any versions that are more recent than that version number.

Your backups will likely be in tar files. Delete the current version of the project, untar the backup version into the repository (you should have a backups directory containing the backups), and delete all other tars that are more recent for that project.

Lock the repository for thread safety.

3. Methodology

The original has a section for this that I won't be covering. These are just things that I thought were important considerations.

Mutexes

1) I suggest doing mutexes **at the end**. Trying to integrate them at the same time will only cause headaches.
2) To avoid weird locks, you can encompass the entire function with the lock rather than small sections of the code. This isn't efficient but it's much safer and more hassle free. This is also what you should do initially.
3) Use Trylock. I suggest using this over straight up just mutex_lock. Check the man pages for more info.
4) Look into semaphores if you have time. They might be easier to use for projects (the repository should still be a mutex).

Deadlocks

1) Be absolutely careful where you place your lock calls. If you deadlock, it's over.
2) Be careful of when you lock repositories. You can cause deadlocks if other threads are locking projects and all of a sudden the entire repository gets locked.

Messages

1) Always … ALWAYS send messages for every success and failure from server to client (and vice versa if needed). ALWAYS print from the client successes and failures and why if you can. Try not to print them from the server if you can avoid it since printf isn't thread safe (although you can always use write() to STDOUT).

Dividing Work

1) One person should do client, the other do server, not both. There's no reason for this to happen since the code for the client never calls anything from the server and vice versa. You and your partner can work on things separately if you divide it by server and client and pace the implementation of each function at the same rate (instead of one doing update and the other doing commit and then the guy doing update needing commit to be done to even do anything). The only thing you should work out before hand is how the client will be sending messages to the server for each command and what success/fail messages will be sent and when. This part is crucial otherwise you'll either end up waiting forever for a message that's never coming or reading old messages from the buffer.

Design

1) Take the time to design your program and pseudocode it. No, really, spend a lot of time carefully designing this out. It'll cut down on coding time.
2) Compartmentalize your code. Don't put everything in one client file and one server file. Please. Just don't. Make a client driver file that just checks the incoming client commands from terminal and then calls functions from other files. Same thing with the server. The driver file should just thread things out and initialize things and then call functions from other files for other actions. If a piece of code can be called more than once, turn it into a function, throw it into some file and call it from there. This will keep each file relatively small and easy to read.
3) Keep a clean code base. Don't put everything in one directory. Have a separate directory for server code and client code. Also put your binary files in another directory as well. You're gonna have a lot of files by the end of this.
4) Don't keep dead files around. Again, clean code base.

System()

1) system() is your best friend for tarring and removing/creating files/directories. Use it.

Makefile

With your code in different directories it'll be annoying to go into them and call make separately. Of course you can also use one over-arching makefile outside of those directories but then you would have to type make server or make client (oh the horror of typing more than just make because too lazy), or potentially turn your one makefile into a giant mess. Use a recursive makefile that calls other makefiles to keep each one small. This isn't that great to do in practice because it's kinda slow for projects with many subdirectories and makefiles but it's good enough for this project.

Extra Credit

I suggest doing to the 20 point extra credit at least because system() makes it easier to do by a lot.

<Enter Conclusion Title Here>

This ended up being a lot longer than I expected, sorry about that. But I hope that it explains this project much better than the original so that deciphering it isn't just another thing to stress about. Good luck to you all, I was hoping this project wouldn't get assigned ever again, but here it is (so is the FileCompressor actually but I quite liked that project).