# Design and implementation of WTF and WTFServer

## How Things Are Formatted:

.Manifest Format:
ProjectVersionNumber\n
Mode FileVersion# FilePath HashNumber\n
Mode ….  HashNumber\n

.History .Commit, .Update, and commit storage center on server files Format:
Mode FileVersion# FilePath HashNumber\n
Mode ….  HashNumber\n

.Configure Format
IPAddress\n
Port#

**Mode Meanings:** A=Added, R=Removed, D=Deleted, M=Modified, C = Conflicted, Z=Unchanged

**File Path Format for Server:**
**Normal content:** ./<project name>/…<any subdirectories or files>
**Pending commits:** ./<project name>/.c/<all pending commit files>
**.files content:** ./<project name>/<.Manifest or .History>

**File Path Format for Client:**
**Normal content:** ./<project name>/…<any subdirectories or files>
**.files content:** ./<project name>/<.Manifest or .Commit or .Conflict or .Update>

**How Previous Versions are Stored:**
Newest version of the project holds the original name of the project. Old versions of projects are renamed with the original name but their version# appended to them. The old versions of the project remain in the same directory as the newest version.

Example: Original project name is “code”
Newest Version is called: “code”
First Version is called: “code1”
Second Version is called: “code2”
Etc…

## To configure the client:

Command Line:

./WTF configure  <IP Address> <port #>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 4), the client will print an error and swiftly return. The client also checks if the given port number is a number and return if it isn't. If a valid command-line argument is presented, a ".configure" file is created in the directory of the executable and the IP address is written into the .configure file followed by a new line and then the port number. The main function used is:

- int configureClient(char* ip, char* port)

## To checkout:

Command Line:

- ./WTF checkout  <project name>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 3), the client will print an error and swiftly return. If a valid command-line argument is presented, then the following function is called:

- int checkout(char * projectName){

This function first checks if the project exists on the client and if it does, it returns with an appropriate error message. Then, a string is created to send to the server that includes 3 letters to represent the function that is being called, a colon, the number of bytes of specific information that is being sent over, and then the information itself(which in this case is the name of the project). This protocol is consistent through all the functions that communicate with the server. This string is called as a parameter to the following function:

- int connectToServer(char * command, char * information, int size)

Within this function, the client creates a connection with the server and writes the string of information to the socket where the server reads it. On the server-side, the beginning string is read from the socket and the command is extracted from that string using the following function:

- void * commandExtract(void * fd2)

Within this method, the following function is called to complete the checkout with the socket descriptor as the parameter:

- int sendCheckout(int fileDes)

Within this function, the rest of the string that was sent from the client is read. The path to the project is created and then it checks if the project exists on the server. If it does not, then the server writes back with the first three letters of the protocol string being “fls” to represent that there was an error on the server-side and the client would print out an error message. Otherwise, the chosen project is tarred and the information is written back to the client in the same protocol that the client communicated with the server, now with the information being the contents of the tarred file. The content of the tarred file is obtained by using the following method:

- int returnContent(char * fileName, char ** charContent)

This method essentially inserts the contents of the named file into the named character array from the parameters and returns the size of the content in bytes. If the write to the client fails with the content of the array the “fls” protocol is written back to the client to prompt an error statement. Now back on the client-side, the connectToServer function reads the string from the server-side and checks for the command. If it is the same 3 letters used to represent the command, the program continues. Otherwise, if the command has changed, then the client outputs an error message and the program terminates. If the 3 letter command is correct, a file with the same name of the tarred project on the server-side is created and the contents of the tarred file from the server-side is written into the newly created file. Now this file on the client, is then untarred. After untarring, the client removes the remaining tarred file and deletes the .c commit directory and .History files within the newly checked out project.

## To update:

- ./WTF update <project name>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 3), the client will print an error and swiftly return. If a valid command-line argument is presented, then the following function is called:

- Int update(char * projectName){

This function first checks if the project exists on the client and if it does not, it returns with an appropriate error message. Then, a string is created to send to the server that includes 3 letters to represent the function that is being called, a colon, the number of bytes of specific information that is being sent over, and then the information itself(which in this case is the name of the project) for example this would be “upd:<sizeofname>:<nameproject>”. This protocol is consistent through all the functions that communicate with the server. This string is called as a parameter to the following function:

- int connectToServer(char * command, char * information, int size)

Within this function, if the client had a .Conflict file but resolved all conflicts, the .Conflict file is deleted. the client creates a connection with the server and writes the string of information to the socket where the server reads it. On the server-side, the beginning string is read from the socket and the command is extracted from that string using the following function:

- void * commandExtract(void * fd2)

The function makes sure to attempt to find a corresponding project, if it does not it sends back an error message to the client which is standard across all commands, “fls:”. Upon locating the project, the manifest is sent back to the client in order to proceed with the update command from the client. Upon receiving the manifest, the client parses the manifest into a linked list and parses its own manifest into a linked list to do comparisons. The codes ‘A’ ‘D’ ‘M’ are appending according to the criteria listed in the AsstLast.pdf, where in short, A is for documents the client does not have, D is for documents the client has but the server does not, and M is for documents that have been changed, the client also appends these updates to a .Update file. If the client has a conflict where its live hash of the file does not match the stored hash or the server hash, this is outputted to a .Conflict file for the user to sort out. Once all “updates” and “conflicts” are detected, the findings are outputted to STDOUT and update is complete.

## To upgrade:

- ./WTF upgrade <project name>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 3), the client will print an error and swiftly return. If a valid command-line argument is presented, then the following function is called:

- Int upgrade(char * projectName){

This function first checks if the project exists on the client and if it does not, it returns with an appropriate error message. Then, a string is created to send to the server that includes 3 letters to represent the function that is being called, a colon, the number of bytes of specific information that is being sent over, and then the information itself(which in this case is the name of the project) “upg:<sizeofname>:<nameproject>”. This protocol is consistent through all the functions that communicate with the server. This string is called as a parameter to the following function:

- int connectToServer(char * command, char * information, int size)

Within this function, the client checks to see if it has a .Update file and a .Conflict file. If it has a .Conflict file the command fails, if it has a .Update file but it is empty, the project is considered Up To Date. If there is no .Update file then the user must first update before upgrade. If there is a .Update file that is not empty, the client creates a connection with the server and writes the string of information to the socket where the server reads it. On the server-side, the beginning string is read from the socket and the command is extracted from that string using the following function:

- void * commandExtract(void * fd2)

Upon receiving a command from the client, the server checks to see if it has the project stored and sends back the “fls:” error if it can not find it. On success, the server requests that the client send it’s .Update file for the required changes. The server cycles through the file and composes a tar file based on the updates the client needs from it. The server sends back a tar file to the client. Once the client receives the tar file, it untar’s it, where the tar file also has an updated manifest for the client and adds or updates changed files in the client’s version of the project according to the .Update file. Once the command is complete, the .Update file is deleted.

## To commit:

- ./WTF commit <project name>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 3), the client will print an error and swiftly return. If a valid command-line argument is presented, then the following function is called:

- int commit(char * projectName){

This function first checks if the project exists on the client and if it does not, it returns with an appropriate error message. Then, a string is created to send to the server that includes 3 letters to represent the function that is being called, a colon, the number of bytes of specific information that is being sent over, and then the information itself(which in this case is the name of the project) “cmt:<sizeofname>:<nameproject>”. This protocol is consistent through all the functions that communicate with the server. This string is called as a parameter to the following function:

- int connectToServer(char * command, char * information, int size)

Within this function, the client creates a connection with the server and writes the string of information to the socket where the server reads it. On the server-side, the beginning string is read from the socket and the command is extracted from that string using the following function:

- void * commandExtract(void * fd2)

The server first checks to see if the project exists, if it does not an fls: error is sent back. Then it sends it’s manifest to the client on success. The client compares the versions of the manifests and if it is the same a commit is possible. The client first checks to see if it has a previous commit. The way that commits are stored on the server are via the hash code of the .Commit. In the event that the client has already committed, the old .Commit is turned into a hash code, and a new hash code is computed from the following information. First the client will check through it’s own manifest and the server’s manifest to append information with an ‘A’ ‘D’ or ‘M’ flag. An ‘A’ flag requires that the client has a file which the server does not have, ‘D’ requires that the client does not have a file the server has, and ‘M’ requires that the has pending changes to a file that already exists. Once these changes are detected, they are outputted to a .Commit file. The old hash code (if it exists from the first commit) is appended to a message with the new commit hash code and the commit content. This string is sent over to the server where the server, upon receiving a message with an old hash code, replaces the old .Commit with the new one, or if there is no previous, it just stores the commit. Once this operation is complete, the server sends over a success message to the client and the command is complete.

## To push:

- ./WTF update <project name>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 3), the client will print an error and swiftly return. If a valid command-line argument is presented, then the following function is called:

- int push(char * projectName){

This function first checks if the project exists on the client and if it does not, it returns with an appropriate error message. Then, a string is created to send to the server that includes 3 letters to represent the function that is being called, a colon, the number of bytes of specific information that is being sent over, and then the information itself(which in this case is the name of the project) “psh:<sizeofname>:<nameproject>”. This protocol is consistent through all the functions that communicate with the server. This string is called as a parameter to the following function:

- int connectToServer(char * command, char * information, int size)

Within this function, the client first checks to see if it has a .Commit file, if it does not, it will prompt the user to commit first. The client creates a connection with the server and writes the string of information to the socket where the server reads it. On the server-side, the beginning string is read from the socket and the command is extracted from that string using the following function:

- void * commandExtract(void * fd2)

The server first checks to see if the project exists, if it does not an fls: error is sent back. Then it requests that the client send over it’s commit so that the server can verify it has the commit file. The client sends it’s commit over to the server, upon successful identification the project is locked and the server requests files listed in the .Commit file to be sent over. Once these are tarred and sent over, the server stores the previous version of the project as <projectname><projectVersion#> and leaves <projectname> to be the most up-to-date folder. The server untar’s the documents sent by the client and appends changes in the commit to the previous history files and stores that .History into the project. Once the files are successfully stored and the server’s .Manifest is updated by the .Commit, the server sends back its .Manifest to the client and the client updates its own manifest to the one the server sent. The push operation is now complete.

## To create:

Command Line:

./WTF create  <project name>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 3), the client will print an error and swiftly return. If a valid command-line argument is presented, then the following function is called:

- int create(char * name)

This function will create the protocol string with the project name and the 3 letters to represent the create command and use it as a parameter for the connectToServer method. The connectToServer method will create a connection with the server and then write the string to the socket where the server will read it. The extractCommand method on the server-side will extract the command from the string and call the following method:

- int creation(int fileDes)

In this method, the rest of the string sent from the client is read and the server tries to open the project to make sure the project does not already exist. If the project already exists, the server writes the error “fls” command back to the server to terminate the program and output an error. Otherwise, the project will be created and a .Manifest with the version number 1 will be setup. The server will write back to the client and the client creates the project and the .Manifest file on its side too.

## To destroy:

Command Line:

./WTF destroy  <project name>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 3), the client will print an error and swiftly return. The following function is called if the command line argument is valid with the project name as the parameter:

int destroy(char * name)

This function will create the protocol string with the project name and the 3 letters to represent the destroy command and use it as a parameter for the connectToServer method. The connectToServer method will create a connection with the server and then write the string to the socket where the server will read it. The extractCommand method

on the server-side will extract the command from the string and call the following method:

- int destruction(int fileDes)

The function takes the socket descriptor as a parameter and reads the rest of the protocol string from the client. This function identifies the project name, opens the project to make sure it exists(sends an error to the client if it does not), reads the version number of the manifest, and then deletes the project, locks the repository and expires any pending commits. Then the server uses the collected Manifest version number and uses it to delete the previous project versions as well. If the deletion of the project is successful, it sends a success message back to the client or sends a failure message to the client otherwise.

## To add:

Command Line:

./WTF add  <project name> <fileName>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 4), the client will print an error and swiftly return. The following function is called if the command line argument is valid with the project name and the filename as the parameters:

- int addFile(char * projectName, char * fileName)

The function first checks if the project even exists within the client. If it doesn't an error message is printed out and the program terminates. It also checks if the file exists within the project and it has the same error case. The manifest entry is created with the mode, the file version numbers, the file path, and the hash number.Then the following function is called within the addFile method with the path to the manifest, the manifest entry, and the path as parameters:

- int appendToEndAdd(char * manifest, char * entryToAppend, char * path)

In this function, the returnContent function is called to store the contents of the already present .Manifest within an array and then the following function is called to sort the manifest entries into the version number, paths, hashnumbers, and the mode using a struct and then links all the structs to create a linked-list:

struct manifestLL * getLL(int sizeOfFile, char * fileContent, char ** version);

This function takes in the size of the file, the content of the file, and char** for the version and returns the head of the linked list of the structs and also sets the inputted version to the .Manifest version. From here, the inserted entry's path is compared with the paths of the entries that already exist in the Manifest. If the path's match, the function checks the mode of the entry and if the mode is 'A' or any other mode other 'R' which stands for added, the program returns an error message that says the file has already been added and terminates. If the mode is 'R' which signifies that the file has previously been removed and it changes the mode to 'A' , deletes the old .Manifest file and then writes the newly edited entries into a new .Manifest file. If there is no path that matches the new entry's path, it is then added to the end of the .Manifest file in a new line. If there was an error with the right, an error message displays and the program terminates.

## To remove:

Command Line:

./WTF remove  <project name> <fileName>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 4), the client will print an error and swiftly return. The following function is called if the command line argument is valid with the project name and the filename as the parameters:

- int removeFile(char * projectName, char * fileName)

This function first checks to see if the project exists and if it does not it prints an error message and returns. Otherwise, it opens the .Manifest within the project and sees if that exits. If it does not it prints an error message and returns. Otherwise, it then calls the returnContent method to store the contents of the .Manifest in an array and then puts that array into the getLL method to create a linked list of structs that breaks down the .Manifest entries. The function that traverses the linked list to see if the path of the desired file name to be removed matches with paths in the .Manifest and if there is a match, the mode is checked. If the mode is already 'R' then an error message is printed and the function returns. If it is not an 'R' then the function updates the mode to 'R'. The function then deletes the old .Manifest file, creates a new .Manifest file, and then writes the updated entries into this .Manifest file. Upon successful completion, the program returns without an error.

## To get the current version:

Command Line:

./WTF currentversion  <project name>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 3), the client will print an error and swiftly return. The following function is called if the command line argument is valid with the project name as the parameter:

- int currentVersion(char * projectName)

This function creates the protocol string with the correct three letter representation of the command, the size of the project name and the project name. Then, the connectToServer function is called with protocol string as the parameter. The string is then written to the server, where the server reads it, the commandExtract function deciphers the command, and then findCurrentVer function is called. This function reads the rest of the string, checks if project exists, and sends the contents of the .Manifest file back to the client attached to the protocol string. The client then sorts the Manifest into a linked list and uses that to print out the filepaths and their versions in STDOUT.

## To get the History:

Command Line:

./WTF history <project name>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 3), the client will print an error and swiftly return. The following function is called if the command line argument is valid with the project name as the parameter:

- int history(char * projectName)

This function creates the protocol string with the correct three letter representation of the command, the size of the project name, and the project name. Then, the connectToServer function is called with protocol string as the parameter. The string is then written to the server, where the server reads it, the commandExtract function deciphers the command, and then findHistory function is called. This function reads the rest of the string, checks if project exists, and sends the contents of the .History file back to the client attached to the protocol string. The client then sorts the .History file into a linked list and uses that to print out the filepaths, their mode, and their versions in STDOUT.

## To get the Rollback:

Command Line:

./WTF rollback <project name> <version #>

Design and Implementation:

When the command line does not have a sufficient number of arguments(in this case, 4), the client will print an error and swiftly return. The following function is called if the command line argument is valid with the project name and verision # as the parameter:

- int rollback(char * projectName, char * version)

This function appends the version number to the end of the project name and then creates the protocol string with the correct three letter representation of the command, the size of the project name, and the project name. Then, the connectToServer function is called with a protocol string as the parameter. The string is then written to the server, where the server reads it, the commandExtract function deciphers the command, and then executeRollback function is called. This function reads the rest of the string, checks if project exists, checks if the .Manifest exists, and parses the .Manifest to get its version number. It then deletes the newest version of the project and then deletes the rest of the versions ahead of the chosen version by incrementing the version# and then appending it to end of the project's name. It then removes the version number from the end of the chosen projects name and renames it so that it is recognized as the current project version. It then writes back to the client to tell it whether the rollback was a success or not and the client relays this to the user.

## Multithreading and Mutexes:

Our server accommodates multiple clients using the server at the same time by creating threads from the server's main thread for each client's request. The client's request is passed through the thread with its process ID where the client's request gets fulfilled by the server which writes back to the client. This allows for multiple client requests to be processed by the server simultaneously. Additionally, there are mutexes implemented in order to prevent two clients from making changes to the same project at the same time.