P2:Where's the File

Daniel Doan
Graham Duebner
Systems Programming, Spring 2020

Synopsis:

Where's The File contains files: (c/h indicates both a .c and a .h file)

- client.c/h
    - Holds the client portion of the project
- test.c
- server.c/h
    - Holds the server portion of the project
- gStructs.c/h
    - Holds linked list structure and associated functions
- fileManip.c/h
    - Library that holds file manipulation functions
- network.c/h
    - Library that holds networking functions
- Makefile
    - make all:Compiles network.c, fileManip.c, network.c into libraries and then compiles client.c and server.c into "WTF" and "WTFserver" respectively with the libraries created.
    - make clean:Removes all libraries and executables
    - make test: Runs the same as make all but also compiles test.c and testserver.c into "WTFtest" and "WTFtestserver" respectively and moves "WTFtestserver" into newly created directory "server"

Design/Methodology:

Where's the File works as a file repository similar to that of github with two executables:

"WTF":

WTF that will act as a link for the user's requests and the server. Note, not every command will necessarily connect to the server, e.g. *add or *remove.

Common Errors:

- If a client is not supplemented with a .configure file or an improperly formatted one, it will print out an error saying it cannot connect to the server.

- If a client connects to a server but supplies the command with a non-existent project, OR client tries to access local repository and cannot find it, it will print an error saying the project does not exist.
    - Likewise, if user wishes to add a file or remove a file, an error will printout saying that file cannot be found
- If the client recognizes the command user inputs an invalid number of arguments for it, it will print out an error saying that user inputted an invalid number of arguments.
- If user inputs and a command that client does not recognize, client will print out an error saying user has inputted an unknown command.
- If a user tries to access a repository while it is locked(due to another client accessing it at the time), client will print out an error asking them to try the command again later.

“WTFserver”

WTFserver is the server executable and takes only one argument to run, that being the port number. WTFserver will wait for incoming connections from clients and respond appropriately to the request. It is recommended to move WTFserver into an empty directory on the machine, as that directory will hold all projects.

Thread Synchronization:

For multiple clients, server creates a thread for each client connected. Afterwards for functions that read or write from a project including:

rollback, currentversion, destroy, commit, push, update, upgrade, history, and checkout

If a client wants to access any of these functions, it will first lock and check a global linked list where each node holds the project name and a value 0 or 1. If the node that the project client is trying to access holds a 0, that client will then change that value to a 1, indicating it is under use and unavailable, and unlock the global linked list, and proceed to perform one of the above functions. Once finished running one of the above, the client will then lock and access the global linked list and change the project’s node value to 0, indicating it is now available to use. If a client locks and accesses the global linked list and sees that the node of the project it wants to access value is 1, it will unlock the global linked list and return an error message to the client asking the user to try again as the project was under use.

Project Storing and Usage:

WTF will take in a variety of commands for accessing and modifying projects. <word> indicates user specified arguments for commands. NOTE: Commands are case sensitive

- configure <IP> <port>
    - Should be first command you want to enter if you have never used the executable in current working directory before

    - Upon being given the two arguments, client will create a new .configure file(replace the currently existing one) to store the IP and port given by user.
        - Format of the file is "<IP>\t<port>
        - Client will always access this file whenever trying to connect to server
- create <projectname>
    - Client side:
        - Client will attempt to connect to server
        - On successful connection, will message server with <projectname>. Client will also build a directory named <projectname> in the current working directory and create a .Manifest inside of it as well.
            - If the project already exists locally, an error will printout saying project already exists locally, but server will still go about business as usual and create a directory.
    - Server side:
        - On acceptance of client, server will attempt to create a directory within its current working directory with the name <projectname> and create a .Manifest file inside. It will also create a <projectname>log, which will act as a log for the project and holds all history of rollbacks and pushes, as well as changes made.
        - Also adds to a global linked list held by the server containing each project.
- destroy <projectname>
    - Client side:
        - Client will attempt to connect to server
        - On successful connection, will message server with <projectname>. Client will then remove the local <projectname> directory and all files inside.
    - Server side:
        - On acceptance of client, server will remove the <projectname> directory and all files inside as well as <projectname>log, <projectname>archive, and <projectnameCommit> if they exist.
        - Removes project from the global linked list of projects.
- add <projectname> <fileName>
    - Client side command only
        - Will look for <fileName> in the <projectname> directory and if found add it to the .Manifest file of the project.
        - If file cannot be found, client will print out an error saying file cannot be found in <projectname> directory.
        - If file is found BUT manifest already contains file, then an error will printout saying file is already added.
- remove <projectname> <fileName>
    - Client side command only
        - Will look for <fileName> in the <projectname> directory and if found remove it from the .Manifest file of the project.

        - If manifest does not contain the <fileName> entry, then client will print an error saying entry cannot be found to be removed.
- checkout <projectname>
    - Client side:
        - Client will attempt to connect to the server
        - On successful connection, client will send <projectname> to the server, and will receive the current version of <projectname> from the server, creating a <projectname> directory and storing all files obtained from the server in there
            - At the end, client should have a mirrored version of the server's current copy of <projectname>.
    - Server side:
        - On acceptance of client, server will look for the <projectname> directory and send the client all the files found inside.
- currentversion <projectname>
    - Client side:
        - Client will attempt to connect to the server
        - On successful connection client will receive the .Manifest of the current version of <projectname> from the server and print out it's contents.
        - Does not need to have the project locally to send the request
    - Server side:
        - On acceptance of client, server will send over the <projectname> .Manifest to client.
- rollback <projectname> <versionNumber>
    - Client side:
        - Client will attempt to connect to the server
        - On successful connection client will send a message to server containing the <projectname> and <versionNumber> for <projectname> to be rolled back to.
            - If the version number is invalid(i.e. Higher than or equal to current version), client will printout an error saying invalid version to rollback to
        - Does not need to have the project locally to send the request
    - Server side:
        - On acceptance of client, server will remove current <projectname> directory as well as any of the more recent than <versionNumber> compressed versions of the project in <projectname>archive.
        - Server will then decompress the <versionNumber> compression of <projectname> and move it into the current working directory, and remove that <versionNumber> compression.
        - Server writes to the <projectname>log indicating a rollback to <versionNumber>
- history <projectname>

    - Client side:
        - Client will attempt to connect to the server
        - On successful connection client will send a message to server containing <projectname> and recieve <projectname>log and print out its contents.
        - Does not need to have the project locally to send the request
    - Server side:
        - On acceptance of client, server will send the <projectname>log to the client
- update <projectname>
    - Client side:
        - Client will attempt to connect to the server
        - On successful connection client will ask server for their .Manifest of <projectname> and upon receiving it will compare it to the local .Manifest. Depending on the differences between updates, client will create a .Update file to represent them.
        - Four cases:
            - If a file is found on the server's manifest that is not on the local
                - Add "A <filename> <serverFileHash>" to .Update
            - If a file is found on the local manifest but not the server's
                - Add "D <filename> <serverFileHash>" to .Update
            - If a file is found on both, AND the hashes are different on the .Manifests
                - Add "M <filename> <serverFileHash>" to .Update
            - If a file is found on both, AND both hashes are different on the .Manifests, BUT the live hash of the file is different from both hashes on the manifest
                - Add "C <filename><liveFileHash>" to .Conflict(creates it if needed to)
            - For each case, client will print out the additions made without the hashes.
        - If the .Manifests match, client will blank out the .Update and remove .Conflict and print out a message saying local is already up to date
    - Server side:
        - On acceptance of client, server will send <projectname>'s .Manifest.
- upgrade <projectname>
    - Client side:
        - Client will check for if a .Conflict exists. If so, client will exit and print out an error saying conflicts need to be addressed before upgrading.
        - Client will check if a .Update exists, if not client will will exit and print out an error saying to update first before upgrading.
        - Client will attempt to connect to the server.

        - On successful connection, client will send over it's .Update file for the server and will receive and delete files as requested by the .Update. .Manifest will be updated to server's copy as well.
        - Files not listed on the .Update are not affected
        - Client will delete its .Update once done
    - Server side:
        - On acceptance of client, server will receive .Update and look at it and appropriately send files marked with "M" or "A" in the .Update, as well as the server's .Manifest.
- commit <projectname>
    - Client side:
        - Client will check if a .Conflict or a nonempty .Update file exists, if so, client will print an error asking to synchronize with the repository before committing. Blank .Update files are ok.
        - Client will attempt to connect to the server.
        - On successful connection, server will send over its .Manifest for client to check project versions. If both are equivalent, then Client will create a .Commit file and add entries to it based on the following three cases.
            - If a file is found on the server's manifest that is not on the local
                - "D <filename><updatedVersionNum> <serverFileHash>" to .Commit is added
            - If a file is found on the local manifest but not the server's.
                - "A <filename> <updatedVersionNum><serverFileHash>" to .Commit is added
            - If a file is found on both,and the hashes are the same on the .Manifests, but on the live hash of the file on the client is different from both(Indicating changes to the files have been made by user)
                - "M <filename> <updatedVersionNum><serverFileHash>" to .Commit is added
            - Each entry added to .Commit is also printed out by Client without the version number and file hash.
        - If both are not equivalent, the client will print out an error asking to synchronize with the repository before committing.
        - On either success or fail, client will delete its own copy of .Commit
    - Server side:
        - On acceptance of client, server will send <projectname>'s .Manifest file, and receive the Client's .commit file.
            - To distinguish different commits, the client's IP address is appended onto the .Commit and stored in a separate directory in the server's current working directory of the name <projectname>Commit.
            - New commits made by the same client will overwrite previous ones.

- push <projectname>
    - Client side:
        - Client will attempt to find <projectname> .Commit. If it cannot find it, client will print out an error asking user to commit before pushing.
        - Client will attempt to connect to server.
        - On successful connection, client will send its .Commit file over to the server. If server's .Commit matches with it, client will begin to send over files in entries labeled with "M" or "A".
        - Client will then delete its own .Commit and replace its .Manifest with one from the server.
    - Server side:
        - On acceptance of client, server will attempt to find .Commit that corresponds to the client. If found, server will see if .Commits will match.
            - If it is not found, client will print out an error asking user to commit.
            - If they do match, client will print out an error asking user to recommit.
        - If .Commits do match, server will compress it's current <projectname> directory and name it in the format <projectname><currentVersionNum> and move the compressed file into <projectname>archive. Server will then receive files in entries labeled with "M" or "A" and update the repository with those. Files in entries labeled with "D" will be deleted from the repository.
            - Each file will be updated accordingly in the .Manifest.
        - Server will then open <projectname>log and write in "Push <currentVersionNum+1>\n", and copy over the contents of .Commit into the log at the end.
        - Server will then send over it's updated copy of the .Manifest over to the client and remove the currently used .Commit and pending commits by deleting <projectname>Commit directory.
        - If any part of the process fails, server will delete it's current <projectname> directory and decompress <projectname><currentVersionNum> in <projectname>archive into server's current working directory

Helper Functions/Structures:

Structures(s):

This struct acted as the networking protocal for the project.

```
typedef struct _message{
  char *cmd;
  int numargs;
  char **args;
  int numfiles;
  char* dirs;
  char** filepaths;
  int *filelens
}message;
```

Helper Functions(s):

- printManifest(char* **manifest**)
    - Given a string that holds all the contents of .manifest(presumably sent by server and is properly formatted), it'll print out the current project version number, files and their current versions numbers. Ignores hashes.
- compressProject(char* **project**)
    - Given the project name, it will compress the directory and its contents and move the tar file into a new directory of the format <project>archive. It will also append the version number gained from the .manifest file onto the file itself, so the compressed file will be <project><version number>.
        - On first compression(detects by reading 1 in .manifest file), creates an archive folder in the format <projectname>archive
        - Does NOT remove the current version of the project after compression
- decompressProject(char* **project**,char* **version**)
    - Given the project name and version number it wants to decompress, the function will go into the <project>archive folder and will untar the compressed project specified by the version number into the current directory.
        - Before calling, please remove current version of project in directory to avoid any potential overlaps/errors
- copyFile(int **ffd**, int **ifd**)
    - Copies contents of **ifd** from where ever file descriptor offset happens to be to the end of the file into **ffd**
- copyNFile(int **ffd**, int **ifd**)
    - Copies contents of **ifd** from where ever file descriptor offset up to N number of bytes into **ffd**

- sendMessage(int **fd**, message **msg**)
    - Fill message struct with a command (**cmd**) and either short messages (**args[ ]**) and/or file names (**filepaths[ ]**) with **dirs[ ]** (1 for directory, 0 for file) and file lengths (**filelens[ ]**)
    - Will fill metadata into **fd**, followed by each file's bytes, in order entered in **filepaths[ ]**
    - Message not freed in this function
- recieveMessage(int **fd**, message **fd**)
    - Takes fd, filled with the message, and enters the metadata into msg (see above)
    - Use copyNFile() to copy bytes of sent files from fd into a file descriptor you must create, do so in the order of **filepaths[ ]**
- freeMSG(message ***msg**)
    - This function will free a fully malloced message pointer
    - ONLY USE IF EVERY ARRAY WAS MALLOCED AND NOT A HARD CODED STRING
    - Can be used on msg created by recieveMessage
- hashFile(char* **fileName**, char* **myhash**)
    - Function will create an fd that will read the entire file specified by **fileName** and put into string before hashing using SHA8 conventions and storing the hash into **myhash** and return it.
- readBytesNum(int **fd**)
    - Will read in bytes until a char ":","\n", or " " is read in, at which point it will convert the read characters into an integer and return it.
    - Used with sockets to find file length to be read, if message struct is not used
- itoa(char* **snum**,int **num**)
    - Converts **num** into a string, stores it in **snum** and returns it.
- strfile(char* **file**,char* **str**)
    - Will open a file descriptor using **file** as it's path and attempt to find **str** inside of it.
    - If **str** is found, it will return its offset as an integer in the file.
    - If **str** is not found, it will return -1