MySVN revision control implementation using the concurrent TCP client-server model

Samson Ioan-Paul

University "Alexandru Ioan Cuza" Iasi, Faculty of Computer Science, Romania

Abstract. In this paper I documented my own implementation of a basic revision control application implemented using the concurrent TCP (Transmission Control Protocol) client-server communication, using the C/C++ programming language.

1 Introduction

I chose to implement this project in order to have a better understanding of what a revision control application is all about, because is it a tool that all developers use on a daily basis.

2 Technologies

In order for the communication to work, my client and server "systems" are connected with the appropiate networking hardware and softwate, including the TCP/IP protocol software (more specifically, for my presentation, the IPv4 internet protocol will be used). The "systems" communicate through socket binded to a port which is the same for both. That socket's communication semantics are of type SOCK_STREAM, which assures sequenced, two-way byte streams with a transmission mechanism for stream data.

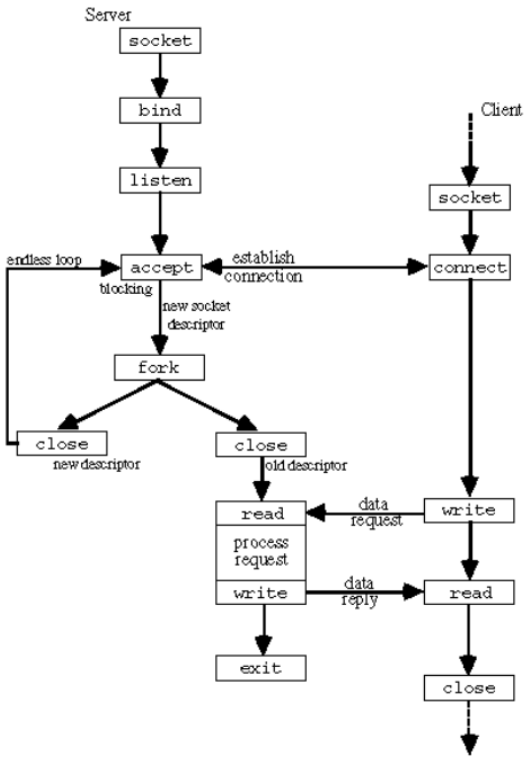

Fig. 1. Concurrent TCP general schema

For my implementation I used TCP because it offers a two-way connection between a server and a single client and provides reliable byte stream transmission of data with error checking and correction, and message acknowledgement, unlike the UDP (User Datagram Protocol) client-server communication, which is not connection based and, while providing fast and lightweight transmission of data through datagrams, is less reliable than TCP.

3 Must-know concepts to understand the project

3.1 Revision control

In software development process, **revision control** (also knows as **version control** or **source control**) is the managment of changes made over time. These changes can be made to the source code, project assets, or any information that goes into the product. It permits more people to work on the same part of a project without worrying that their changes will overwrite any others. The collection of revisions is called a **repository** (or **repo**, for short). The repository keep a step-by-stop chronological record of all changes made to help project managers revert to a previous state if necessary.

How revisions are made
Revision control systems are usually hosted on a networked server. After the repository is up, the developers can add new files, edit existing ones (by saving the files locally and committing the changes), merge commits that are in conflict (they affect the same parts (lines) of a file). If there are no conflicts, the new version is updated in the repository and it receives a revision number.

3.2 Repository commands

As mentioned before, the developers have different actions they can do in order to modify the repository.

Cloning the repository: it saves the latest version at the moment locally, on the developer's computer, in order for the developer to make changes and commit them when done.

Committing changes to the repository: it can mean either that the developer is adding new files to the repository, or changing existing ones. The files are added/changes locally and the changes are commited to the repository.

Reverting changes from the repository: if the developers sees that the new changes don't work, they can choose to revert the project to a previous working version. This can be done because of hashing; every state of the code has a hash, a unique stamp, thus ensuring that reverting to a previous version of the project is also unique.

Each developer has a set of permissions on the repository, developers that are missing certain permissions can not do certain commands. For example, only people that have read permissions can clone from the repository, only people that have write permissions can commit changes and only administrators of the repository can revert the project to a previous state

3.3 Hashing

Every commit effected on a repository is provided a hash (take for example a SHA1 checksum). Thus the hash number is unique for each commit, and reverting to a hash of a previous working state of the project is easy and safe.

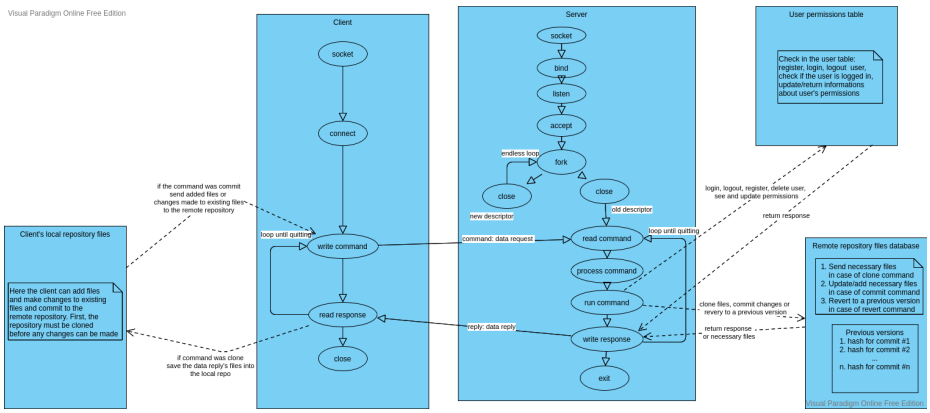

Fig. 2. General Project Schema

4 Implementation

The project if implemented using a C style concurrent TCP client-server communication. I am also using some C++ (c++17) functionalities (like std::string, std::vector).
Communication is realized like in the above shown schema. The client will attempt to connect to the server, send commands and receive success or error responses. The server will bind the socket to it's port and IP address and then listen for clients who are trying to connect. Then the server will accept said clients, create a child process that will take care of their "assigned" clients and then it will keep listening for more clients. The child processes will receive the commands that the clients input, process them (as well as do all the checks necessary to see if they are logged in or have the needed permissions), send back a response message and quitting when the client closes their session. The command and responses from the client and server respectively are read/written to the socket as strings, prefixed by the string length. This approach assures that we can send strings of variable sizes and it removes the unused space that a fixed length string sending approach would have.

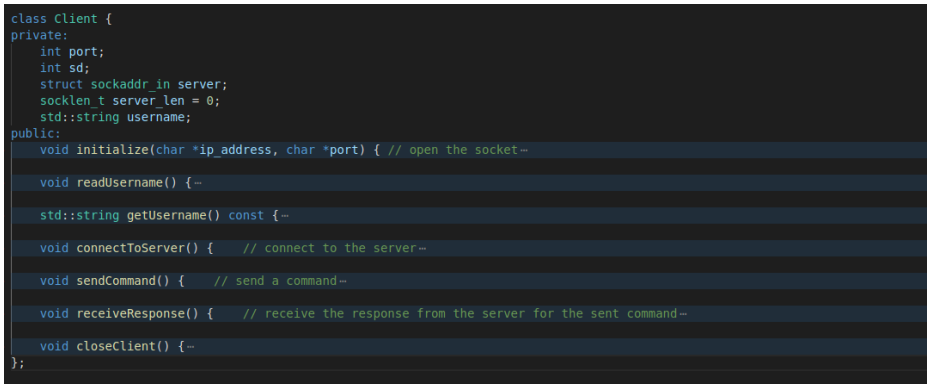

Fig. 3. Client Implementation

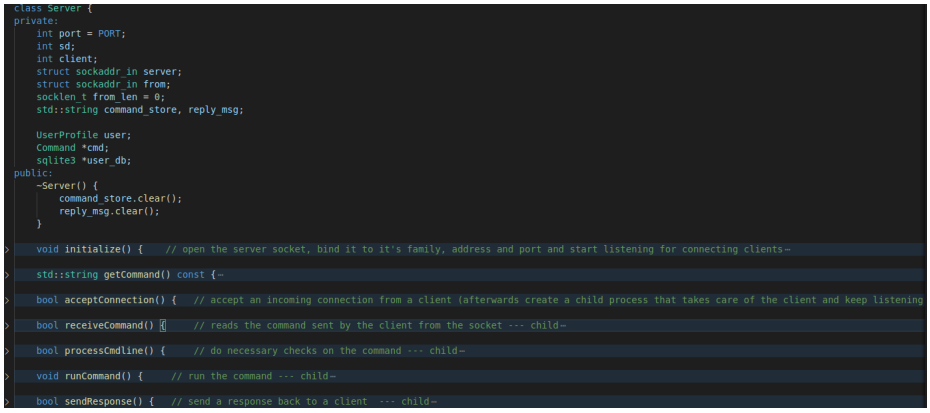

Fig. 4. Server Implementation

The users and they permissions on the repository are stored in a database. The repository files and folders are also stored in a separate database. For the implementation of the database system I used the C/C++ library SQLite3, which implements small, high-reliability SQL relational database engine.

4.1 Users

The clients must first register to the server, after which they will have only the permission to read (clone) from the repository. There are only administrators of the repository, which can at any time add or remove any permissions for certain users. The username of the clients logged in the database acts as a primary key, thus there can not be two users with the same username.

```
class UserProfile
{
private:
    std::string username, permissions;
    bool is_logged = false;
public:
    void loginUser(const char* username) {…
    void setUsername(const char* username) {…
    void setPermissions(std::string permissions) {…
    std::string getUsername() const {…
    std::string getPermissions() const {…
    void logoutUser() {…
    bool isLogged() {…
    UserProfile& operator= (UserProfile user2) {…
    void userPrint() {…
};
```

Fig. 5. User profile class that memorizes if the users are logged in and their usernames and permissions

After the account is registered and the administrators set the necessary permissions, the users must log in the the server with their username. As explained before, the users can do different actions on the repository depending on what permissions they have. The users can log out at any time, or simply quit their current session.

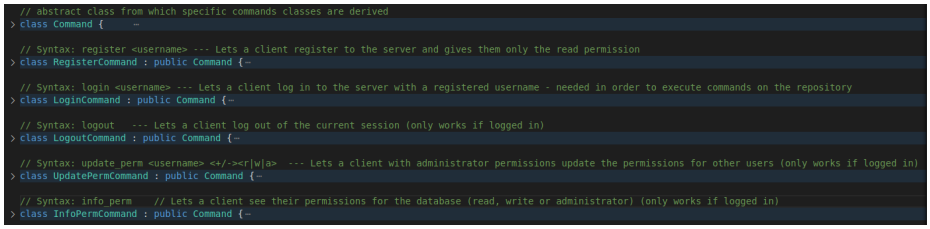

Fig. 6. Implementation of users logging and permission commands

4.2 Working on the repository

After they are logged in, the clients, depending on their permissions, can clone, commit changed or revert commits on the repository. As mentioned before, having read permission ('r–') means that a user can clone from the repository, having writing permission ('-w-') means that a user can commit files and changes to the repository, and having administrator permissions ('–a') means that the user can also revert the project to a previous state if necessary. Note that having administrator permissions implies that you have all permissions ('rwa').

4.3 Commands

register (username) — Lets a client register to the server and gives them only the read permission
login (username) — Lets a client log in to the server with a registered username - needed in order to execute commands on the repository
logout — Lets a client log out of the current session (only works if logged in)
update_perm — (username) (+/-)(r—w—a) — Lets a client with administrator permissions update the permissions for other users (only works if logged in)
info_perm — // Lets a client see their permissions for the database (read, write or administrator) (only works if logged in)
remove (username) — // Lets a client with administrator permissions remove other users from the users database
init — initialize the repository
clone (repo_version) — // Clones the repo's (repo_version) to the client's local storage
commit — // Lets a client commit their local changes to to repository to the remore repository
revert (version) — // Lets a client (with admin permissions) revert the remote repository to a previous version
verhashlog — // Lets a client see all commit hashes along with their commit messages

4.4 Utilization scenarios

We start by registering a user, logging in to an existing user, logging out and back in, checking what permissions we have. We can also start a new client without waiting (since the implementation is concurrent), log into a user with administrator permissions, and see if we can update other users' permissions. Then, depending what permissions we have, we try to clone the repository, commit changes or files to it, or revert it to a previous state. When we're done using we run the command "quit_session" in order to close the current client (quitting also automatically logs you out of the server first).

5 Conclusions

The concurrent TCP client-server communication is a very safe was to realize such a communication because of its reliable byte stream transmission of data with error checking and correction, and message acknowledgement.

Revision control is a very good way for multiple developers to cooperate on the same project. Reverting to previous working versions of the same project is also a very good way to make sure that even if last moment bugs happen and there is no time to fix them, the developers still have the opportunity to go back to a previous working state, while also keeping the code and project assets organized.

One way to improve the project could be keeping the repository files in a document oriented database (like MongoDB), because it would have faster access time to the project code and assets. Also, implementing a branching system, so not all commits get pushed right into the main branch can fix the issues of overwriting that could appear and it would make it easier to see and review the changes for each commit, when the merges of the different branches to the main branch happen.

References

TCP/IP protocol documentation:
1. https://en.wikipedia.org/wiki/Internet_protocol_suite
2. https://docs.intersystems.com/irislatest/csp/docbook/DocBook.UI.Page.cls?KEY=GIOD_TCP
3. https://profs.info.uaic.ro/ gcalancea/laboratories.html
4. https://profs.info.uaic.ro/ computernetworks
Revision control documentation:
5. https://ro.wikipedia.org/wiki/Apache_Subversion
6. https://subversion.apache.org/docs/
7. https://www.computerhope.com/jargon/r/revision-control.htm
8. https://liquidsoftware.com/blog/the-7-deadly-sins-of-versioning-part-2/
SQLite3 relational database documentation:
9. https://www.sqlite.org/cintro.html
C/C++ function documentation:
10. https://www.cplusplus.com/
Schema was made with:
11. https://online.visual-paradigm.com/app/diagrams/