VirtualSOC

Socea Teodor

Universitatea ”Alexandru Ioan Cuza”, Iași, secretariat@info.uaic.ro
https://www.info.uaic.ro

Abstract. The following document contains details about the implementation and functionality of the VirtualSOC project, a client/server application aiming to simulate a social network. In the following sections can be found all possible commands, the message evaluation process and the technologies used.

Keywords: Social Network · Server · Client · TCP/IP

1 Introduction

VirtualSOC is an application aiming to simulate a social network. Users will be able to connect to the concurrent VirtualSOC server using the VirtualSOC client. Users will be able to view posts on the network, register for an account or log into an existing one. A registered user can post news and add friends. They can also change their profile to private or post news only a specific group of friends can view.
The server can be hosted on any Linux based machine and will run on port 2030 by default, but that can be changed in the code.
The client can run on any Linux based machine and provides a user friendly menu based console interface, no GUI is present.

2 Technologies Used

2.1 C++

The entirety of the application, both the client and the server are written in C++ and are aimed to be used on the Linux operation system exclusively. The reason for choosing to work with C++ over C was a simple one. It provides all the functionality a low level implementation would need but also has higher level features like OOP and templated types such as strings and vectors.

2.2 TCP/IP

The app only uses the TCP/IP protocol, because the sent messages from the client or the server need to be intact and in order when they reach. Therefor the correctness of the sent packets was prioritized for determining the memory to

be used and the time of execution.
In other words, UDP (acronym for **U**ser **D**atagram **P**rotocol) packet technology is not connection oriented meaning it does not perform checks for a correct transmission of packets and it can lead to incorrect messages sent or lost on the way.
On the other hand, the TCP (acronym for **T**ransfer **C**ontrol **P**rotocol) technology is connection oriented, bringing significant advantages to the application. The packets are sent integrally and in order and a check is always performed when receiving them. Despite the fact that these checks may have an impact on the performance, it is a trade off I was willing to make because having a complete packet is better than an incomplete one.

2.3 SQLite

The server side application uses the SQLite library for creating a database to store all the information in. SQLite databases are not "real databases", however they are easy to work with and are interfaced with the SQL language. More about the architecture of the database can be found further in the document.

2.4 OpenSSL

The client side application uses the OpenSSL library for hashing the user passwords. The aim of this was to hash the password as soon as possible and send it over the network already hashed, so anyone trying to capture information on the network will be unable to get the password. The SHA256 algorithm was chosen because it is fairly fast and incredibly difficult to brute force back to the original. On top of that, both the client and server side applications use a base64 encoding mechanism to encode and decode data sent over the network, so that no unexpected characters are present in parsing the package received.

3 Architecture

3.1 Application Diagram

The application contains 3 components that communicate with each other: Client - Server and Server - Database. For efficiency, a concurrent server was implemented, using the `fork()` function in order to create child processes which will handle each client request.

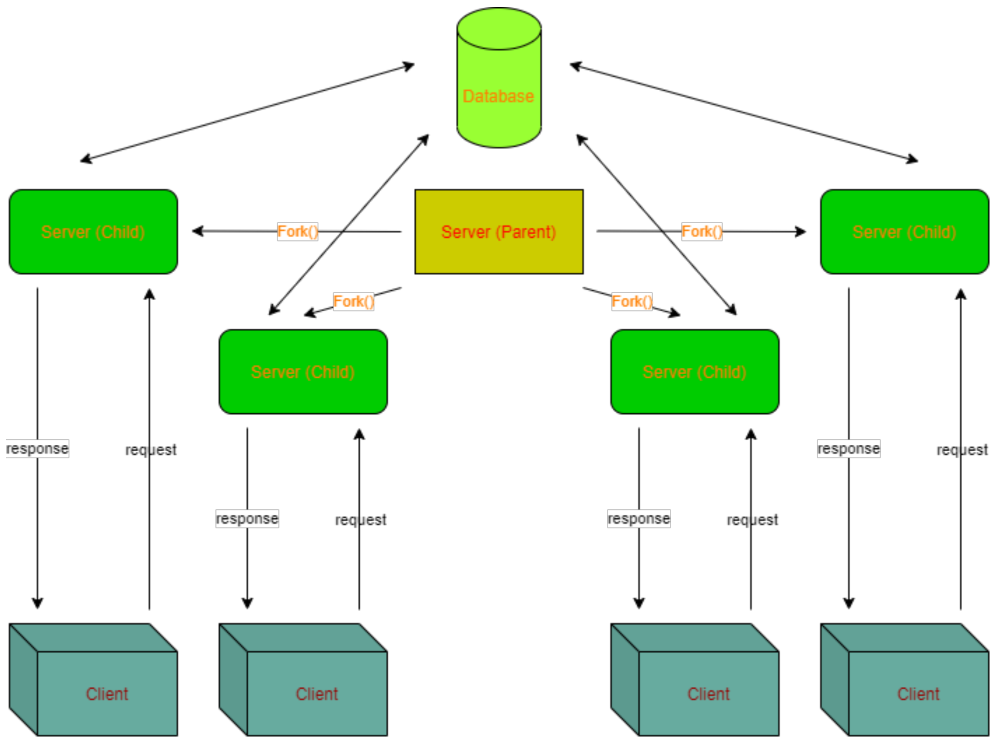


Fig. 1. Application Architecture

3.2 Data structures used

The application uses the following data structures in the form of classes:

– The *Command* class used for storing elements of a command to be sent to the server.

```
class Command {
    string id;
    vector<string> args;
}
```

– The *Response* class used for storing elements of a response to be sent to the client.

```
class Response {
    string status;
    vector<string> args;
}
```

- The *Post* class used for storing elements of a post. Certain elements of a post will be sent and received as arguments for *Commands* or *Responses*.

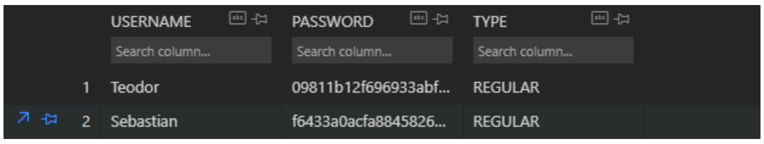
```
class Post {
    int id;
    string title;
    string author;
    string content;
    string date;
    string type;
}
```

- The *User* class used for consistency between the server and client. Users can be of type "REGULAR" or "ADMIN".

```
class User{
    int id;
    string username;
    string type;
}
```

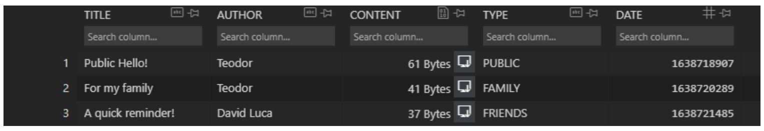
3.3 Database Architecture

The server side application uses a SQLite database, as previously stated, because of how simple and efficient it is to work with. So far the database has 3 tables: Users, Posts and Friends.
The Users table stores information about the users such as the Username, Password and Type. The password represents the hashed passwords the user has registered with. The type can be REGULAR or ADMIN.
The Posts table stores all the posts REGULAR users have created. Posts have a Title, Author, Content, Type and Date. The Date is the moment the post was created and is stored in unix time. The Type of a post can be PUBLIC, FRIENDS or FAMILY.
The Friends table stores the ID's of REGULAR users who are friends with each other and the Type of friends they are, like FRIENDS or FAMILY.
Message storage is handled by the database using tables whose names respect the following naming schema : "TBmin(id1, id2) max(id1, id2)". and store the sender, the message itself and the timestamp of when the message has reached the server.



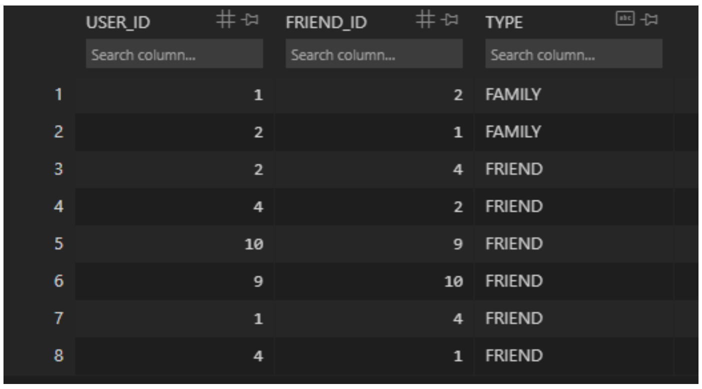
	USERNAME	PASSWORD	TYPE
1	Teodor	09811b12f696933abf...	REGULAR
2	Sebastian	f6433a0acfa8845826...	REGULAR

Fig. 2. Users Table



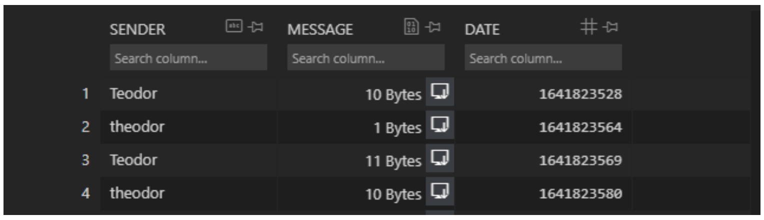
	TITLE	AUTHOR	CONTENT	TYPE	DATE
1	Public Hello!	Teodor	61 Bytes	PUBLIC	1638718907
2	For my family	Teodor	41 Bytes	FAMILY	1638720289
3	A quick reminder!	David Luca	37 Bytes	FRIENDS	1638721485

Fig. 3. Posts Table



	USER_ID	FRIEND_ID	TYPE
1	1	2	FAMILY
2	2	1	FAMILY
3	2	4	FRIEND
4	4	2	FRIEND
5	10	9	FRIEND
6	9	10	FRIEND
7	1	4	FRIEND
8	4	1	FRIEND

Fig. 4. Friends Table



	SENDER	MESSAGE	DATE
1	Teodor	10 Bytes	1641823528
2	theodor	1 Bytes	1641823564
3	Teodor	11 Bytes	1641823569
4	theodor	10 Bytes	1641823580

Fig. 5. Messages Table Example

4 Implementation Details

In the following sections can be found a description of the protocol used in the application and how the messages are being sent between the client and the server. Possible situations:

- A client has connected to the server as guest.
  The guest has access to the following commands:
  - `register` → after the registration process, a message respecting the folling format will be created: `REGISTER:USERNAME:PASSWORD`, where the password is already hashed. These are the following responses the server will send:
    - `SUCCESS` → registration completed successfully
    - `FAILED:USERNAME ALREADY EXISTS` → registration did not complete successfully because the username already exists
  - `login` → after the login process the following message will be constructed: `LOGIN:USERNAME:PASSWORD`. These are the responses the server will send:
    - `SUCCESS` → loged in successfully
    - `FAILED:WRONG PASSWORD` → username or password wrong
    - `FAILED:USERNAME DOES NOT EXIST` → the username does not exist in the database
  - `View Posts` → is the method guest users can see the public posts on the network. A message like `LOOKUP:PUBLIC` will be sent, which will receive the following response:
    - `SUCCESS:POSTCOUNT` → the server will send the client POSTCOUNT posts in forms of strings.
    - `NOPOSTS` → no more posts are available to view
  - `Close Application` → this will terminate the connectiont to the server.

- The client has connected as **REGULAR** to the server.
  These are the available commands:
  - `View Posts` → the REGULAR user will send messages representing which type of posts they would like to see like GETPOSTS:PUBLIC, GETPOSTS:FRIENDS, GETPOSTS:FAMILY and will receive the same response as a guest user with filtered posts.
  - `Add Friend` using this command users can add friends to their friends list. LOOKUP:FRIENDUSERNAME will be sent to the server. The server will send back a list of users that match the FRIENDUSERNAME sent. The authenticated user can select a username and the type and send to the server ADDFRIEND:USERNAME:TYPE.
  - `Create Post` → Using this command the user can create a post, give it a title and select what group the post is adressed to. After this process is completed, the message will be created:
    CREATEPOST:TITLE:CONTENT:TYPE.
    The server will send a SUCCESS response regardless of if it errored or not.

  - `View Friends` → Using this command the user can view their friends list. The message sent to the server is LOOKUP:FRIENDS. The server will return the message SUCCESS:FRIENDSCOUNT, followed by FRIEND:USERNAME - TYPE, or NOFRIENDS when no friends are available. If a friend from the list is selected the client will send a CHATWITH:FRIENDNAME message to the server and will begin chatting with that person. The server will send SUCCESS:MESSAGECOUNT and the latest 10 messages in the table as MESSAGE:MESSAGECONTENT. The user can press ”ESC” to send DONE to the server and return to the main menu.
  - `Log out` → Using this command the user will log out and become a guest user. Message is : LOGOUT. Server will send back SUCCESS.
  - `Close Application` → The client is going to send QUITSESSION to the server and close. The server will exit out of the child.

– The client has connected to the server using an **ADMIN** account.
  An administrator only has access to 3 commands:
  - `Delete Posts` → Using the same protocol for getting posts previously mentioned, the user will receive a list of all posts, public or private, and will be able to select one and delete it. The delete command is DELETE:POSTID. The server will respond with SUCCESS regardless.
  - `Log out` Is the same like a REGULAR user.
  - `Close Application` Is the same like a REGULAR user.

5 Conclusions

At the moment the application is working as intended, the commands and messages are parsed accordingly, and the user interface is intuitive and easy to use, however there are a few improvements that can be made.
First of all, message security was not prioritized so they are sent as base64 encoded text over the network, adding a layer of inconvenience to an attacker.
The application does support a messaging feature however there are no automatic refreshes. This could be improved and turned into a live chat. Lastly, further ADMIN functionalities could be implemented such as a shut down remotely.

References

1. Retele de Calculatoare
   `https://profs.info.uaic.ro/~computernetworks/`
2. StackOverflow
   `https://stackoverflow.com`
3. Tex StackExchange
   `https://tex.stackexchange.com/`
4. Diagram Software and Flowchart Maker
   `https://www.diagrams.net/`
5. TutorialsPoint
   `https://www.tutorialspoint.com/index.html`