QuizzGame

David-Andrei Viziteu
viziteu.david@gmail.com
Group E1, 2^{nd} year

Alexandru Ioan Cuza University of Iași

1 Introduction

QuizzGame is a client/server app that distributes the clients into sessions. All clients in a certain session will get multiple questions (one at a time) and must answer each one in a given interval so they can achieve points. At the end of the session, the clients are ranked by their total score. If a client leaves a session (willingly or unwillingly), he will not be able to re-join that session.

The administrator of the server will prepare the questions, set the number of clients required for a session to start and set the time for all questions. Then, he will be able to instruct the server to start the sessions or to no-longer start new sessions. After one session finishes, the results can be exported to a .txt file.

2 Used technologies

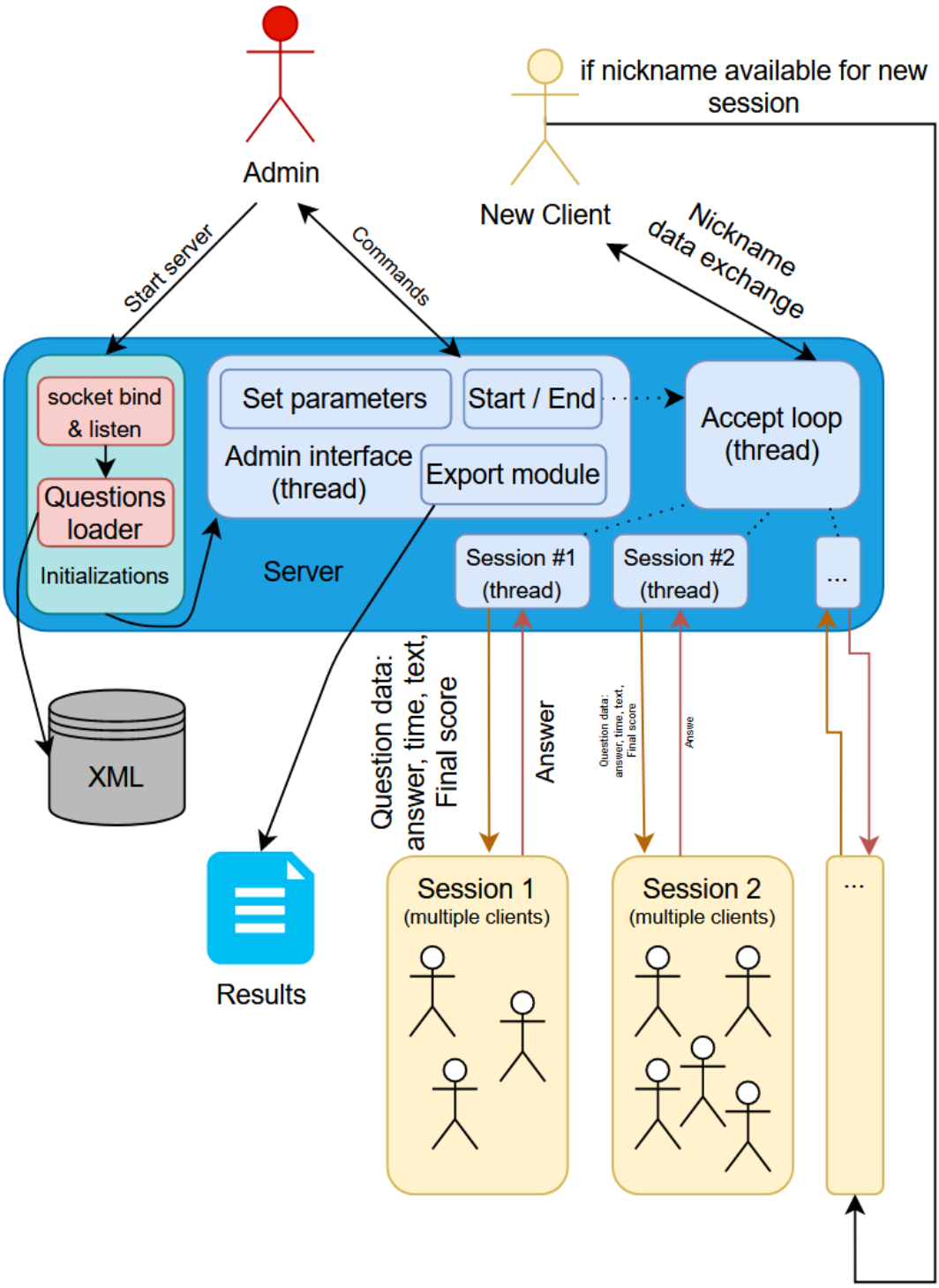
For the communication between the clients and the server I choose the TCP protocol over UDP, as this application does need a message reception confirmation on both client and server side. Furthermore, TCP allows applications to send a continuous stream of data for transmission. Applications don't need to worry about making this into chunks for transmission - TCP does it.

I chose C++ to implement the project. The questions are stored either in an XML file.

3 Application architecture

The project consists of two parts: the client app and the server app. The server is being used to synchronise the clients:

- clients will be distributed into sessions of N participants (server-controlled number)
- each question has the same number of points
- the answer to every question must be submitted by the client in n seconds (server-controlled number) to be taken into consideration
- if a client disconnects unexpectedly, he is disqualified and will not be able to rejoin the session
- at the end of the game, the server establishes the ranking of the participants of a session by their total score



4 Implementation details

Both the client and the server app have been implemented to be IP version-agnostic [1].

```
unsigned session_size = 2;           // clients
unsigned q_pause_time = 3;           // seconds
unsigned questions_time = 3;         // seconds
unsigned questions_per_session = 3;  // questions
unsigned queue_time = 10;            // seconds
auto sock_flags = MSG_CONFIRM | MSG_NOSIGNAL | MSG_DONTWAIT;
struct client {
    char ip[INET6_ADDRSTRLEN];  // client's ip address in human readable form
    int fd, id;                 // socket file descriptor, client id
    string answers;             // client's answers
    unsigned int score = 0;     // current/final score
    bool is_connected = true;   // flag
    char nickname[100];
    string *correct_answers;
};
vector<vector<client>> sessions;
vector<pair<string, char>> questions;  // pair of question and answer
bool allow_new_clients, last_session;  // some sort of flags
int total_sessions, current_client_id = 1;
mutex stdout_mutex;                 // for threads
vector<int> completed_sessions;  // for exporting functions
int sock_fd;                        // listening socket
```

An unique ID will be assigned to every client as they connect.
Server commands:

- start sessions
- set questions time *number* (command available only before the start of the sessions)
- set pause time *number* (command available only before the start of the sessions)
- set questions number *number* (command available only before the start of the sessions)
- set queue size *number* (command available only before the start of the sessions)
- set session size *number* (command available only before the start of the sessions)
- print results results $finishedSessionNumber$/all
- export results $sessionNumber$
- last session (do not accept new clients)
- exit

A separate thread will handle new connections. When the administrator uses the "start sessions" command, this thread will start answering to the pending connections and start a session as soon as there are a certain number of clients waiting.

For each session, another thread will be made to iterate through a vector of clients, sending a question and receiving an answer. If a client does not answer in the given interval, the thread will add a special character to the answers array. If a client disconnects, the thread will set the corresponding flag of the client structure accordingly.

The export results command will be available to use only after the game has finished. It will export the ranking of the players of a session in a .txt file.

The clients will be able to only send (almost) anything to the server (letters/numbers), but the server will only take into consideration only one char (1 byte) for each given question (after the text has been sent to the client). When the time expires, the next question will be displayed.

4.1 Communication protocol

After the administrator uses the "start sessions" command, the server will accept incoming requests from the clients. The server assigns and sends to each client an unique id (4 bytes, integer), and the session players' nicknames (pre-formatted string). After receiving the id and the string of opponents,

the clients will wait for tuples of: a char on 1 byte (question answer), an integer on 4 bytes (question time), a text of variable length (question text, pre-formatted with answer choices by the server). After all the questions, the server will send the client it's total points (an integer, on 4 bytes) and the ranking of the players in that session. Please notice that a question time can only be a positive, not null integer (you cannot answer a question in 0 seconds or in -1 seconds). To inform the clients that the session has finished, the server sends to them their score, multiplied by -1. The score (question time) will be either 0 or a negative number, and when it's received, the client app will reverse the multiplication, display the score for the user/participant, will wait for a pre-formatted text containing the rankings of the players and will stop listening for new questions, ending it's execution.

When the client connects to the server, it firstly sends a string - the nickname chosen by the client user - and then waits for a confirmation of availability (4 bytes, integer) for that nickname from the server. If that nickname was taken, both the server and the client closes the connection, the user being asked for another nickname. Otherwise, the server places the client in a session.

Once the sessions starts (the clients receives his id and the string with players), the client app will be split into two threads: one that will handle the user input and send it to the server, and the other that will handle the incoming data from server.

5 Conclusions

The project is very complex and it can be used in schools or companies to evaluate the participants or just for pure fun.

Furthermore, it can be improved by adding a GUI to both the client and/or the server thus providing a better user experience.

References

1. https://beej.us/guide/bgnet/html//index.html
2. https://profs.info.uaic.ro/~computernetworks/
3. http://man7.org/linux/
4. http://en.wikipedia.org/wiki/Transmission_Control_Protocol
5. https://nanxiao.me/en/use-epoll-in-multiple-thread-programming/
6. http://www.cplusplus.com/reference/thread/thread/
7. https://stackoverflow.com/questions/108183/how-to-prevent-sigpipes-or-handle-them-properly
8. https://stackoverflow.com/questions/2593236/how-to-know-if-the-client-has-terminated-in-sockets
9. https://app.diagrams.net/