Network Trivial Game

Project Overview:

Our project is a trivia game where users connect to a central server and answer trivia questions. In our project, we will take trivia questions off of a trivia file that come from this website (https://opentdb.com/api_config.php) and display them to the users along with potential answers. The users must answer the question in a given amount of time. If a user doesn't answer the question within the time period or the user got the wrong answer, they will not get any points. At the end of each questioning round, the leaderboard will be displayed with the current leaders. Once the questions are exhausted, the game will end and the final leaderboard will be shown.

Design & Implementation:

So as to make the system work we created a server file that handled server side operations and a client file that we distribute to the clientele.

General overview of Server:

In the server.c file, we read off the trivia questions stored in our questions (json) file using Jansson (Version 2.12). We open up connections to the clients and prompt the person using the server to press "s" when they want to start the game. Note that clients cannot (and should not) connect to the server while a game is ongoing. Once the game is started, the server uses a header file that contains metadata for the first question to send all the required information to the clients. We then start the question timer on the server and accept any answers that come from the clients. We compare our client answer to our real answer and calculate the score depending on the amount of time it took for the client to answer (and correctness). Once our question timer is up, we assign score values to the clients, sort the clients based on score (thus creating the leaderboard), and send out our leaderboard to our clients. After a short delay for the clients to check their score, we will start the cycle again by sending out our next questions' metadata. Once we have exhausted our questions, we will send the leaderboard information out one last time for the users to see how well they did.

Play_game: This function takes care of the bulk of the game, it does the following:

- Checks the current questions' json objects for errors
- Randomizes the question order to obscure the correct answer (since the json reads them out with the correct answer as the first one)
- Send out the question to all the clients
- Waits for clients for the allotted time
- Sends out leaderboard information to all the clients using report_standing()

Listen_for_new_connections: Adds any clients that connect into the client hashmap using new_user
New_user: Adds the client into the hashmap
Listen_msg: Listens for messages from the clients that answer the current question. We do the following:

- Check for the answer correctness (we take in capitals and lowercase)
- Assign score values by calculating the score using score()

Score: Calculates the score depending on how long it took for the user to answer
Report_standing: Sends out leaderboard information to all of the clients, this sorts the clients by score

General overview of Client:
In order to run the Client.c file, a user will need to provide a username, the server computer's name and the server's port number (ex. ./client ryan rasiowa 38087). After connecting to the server, the client will have to wait for the server to send its first question out. Once the server sends out the question, the client will assume that the first question starts the game. Upon receiving a question, the client will display the question and possible answers while providing warnings to the user about time remaining to answer. In order to answer the question, the user will send a/b/c/d through the command line and the client file will send the answer to the server afterwards. If the client does not answer the question in time, the client file will inform the user and prevent any input from the user from reaching the server.

Client:
Listen_Thread: This function is called with a pthread. It waits until a header_t is sent, then it reads off the message type and decides on what action to do in accordance to the message_type flag.

- Q = question
    - This tells the function to start reading in questions and answers and displaying them
    - Once everything is displayed, we call Wait_for_input()
- L = display leaderboard, calls the Display_Leaderboard() function
- F = Ends the game

Wait_for_input: Waits for a specified amount of time, warns the user when there are WARNING_TIME seconds left. It also reads off input the user submits and calls Send_Message in order to send the message to the server.

Send_Message: Sends a message back to the server containing the time answered and the submitted answer

Display_Leaderboard: This function reads in leaderboard information from the server and displays it to the user using print statements. Whenever this function is called, it open ups connections to the server and attempts to read in leaderboard information.

Evaluation:

Hardware:

- For the evaluation the hardware we will use includes desktop PCs that are connected on MathLan. The number of PCs depends on the number of potential players of the Trivia Game.

Software/Libraries:

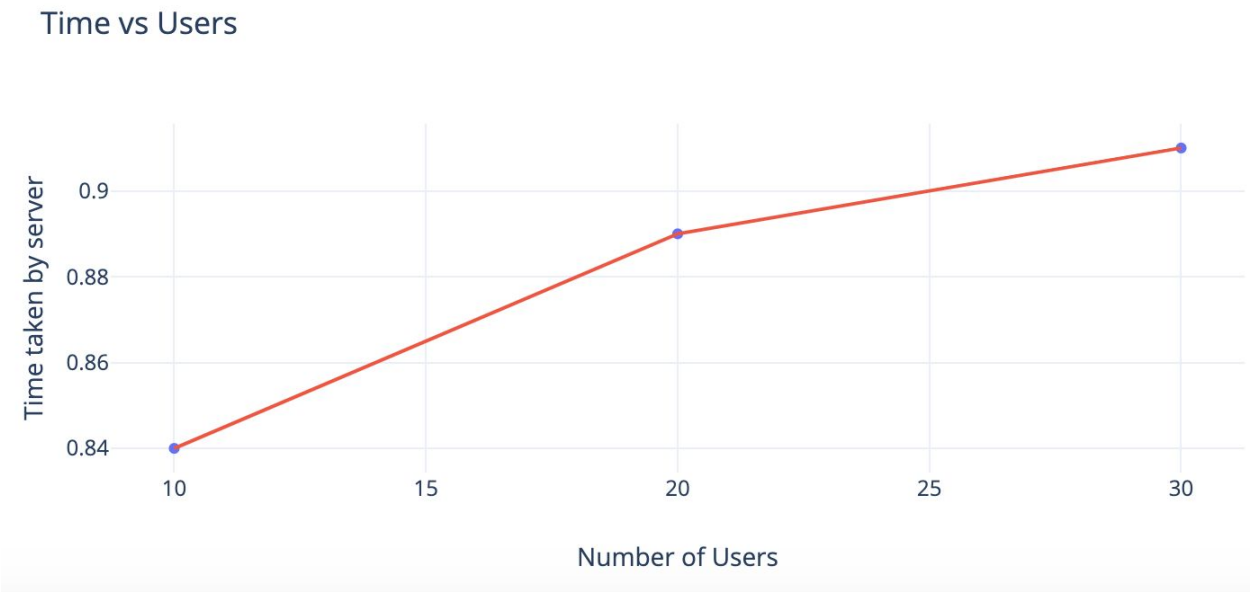
- Among the utilized libraries is the Jansson C library. Jansson is a C library for encoding, decoding and manipulating JSON data. It features: Simple and intuitive API and data model, comprehensive documentation, no dependencies on other libraries, full unicode support (UTF-8), and an extensive test suite.
- So as to keep track of time we use the *"time.h"* library. Time is utilized so in the code in order to count down the number of seconds left to answer a question. Each player has a total of 20 seconds to answer a question. A warning is also issued when the player/user has 5 seconds left
- The time.h library is also utilized in order to time how long the server runs for.

The aspect of our system behavior that we are measuring is its efficiency given an increase in the number of connected users.

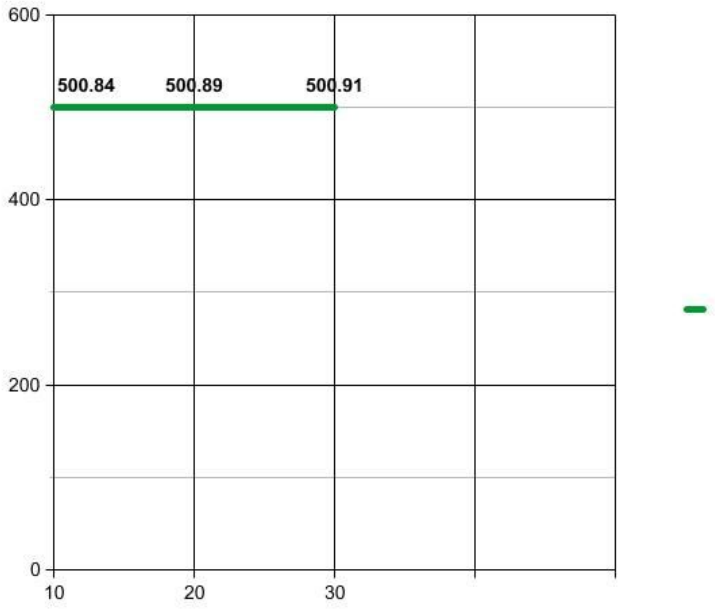
Evaluation Process:
We would run the server with a timer. As soon as we do so we would beguin timing the server. Given that the pool of questions we are drawing from contains an x number of questions and given the fact that a total of 20 seconds is allotted to answer each question there is a total of (20*x) seconds that is relegated to giving users time to play the game. Any time the server spends beyond the base (20*x) seconds while running, is "extra" time. We will see how this "extra" time varies given the number of connected users. These findings will then be depicted on a graph, with the x-axis representing number of connected users and the y-axis representing the amount of "extra" time taken by the server to finish running.

Results:

Graph Depicting Users vs Time



As can be seen in the graph above, there is a weak positive correlation between users and time. As the number of users increases, the time taken by the server also increases in turn. This graph has 500 seconds subtracted from each time.



Additional In-depth Explanations:

Server:

- The server contains a locally created **“connections_threads”** array that stores the process IDs of all the connected users.
- Structs are created so as to store user information, header information, and passed arguments.
    - Our header struct for storing responses (**“response_header”**) contains the following fields:
        - char response_type
        - int username_len
    - Our user struct for storing the different clients (**“user”**) contains the following fields:
        - char username
        - int points
        - int fd
    - Our user struct for storing further user data such as a users array and number of users (**“users_map”**) contains the following fields:
        - user_t users_arr
        - int num_users
        - int map_size
    - Our args struct for storing arguments later passed as parameters for thread functions (**“args”**) contains the following fields:
        - int id
        - char message
    - Our header struct (**“header”**) contains the following fields:
        - char message_type
        - int num_users
        - int message_len
        - int username_len
        - int ans_lens[4]
- Due to the use of threads two locks are preemptively declared

- Functions within the server:
    - **“report_standing”**
        - Going through all the users, compiling their points with their corresponding usernames and sorting them only to then sending this information to the client thus allowing them to acquire the information needed to display a leaderboard.
    - **“listen_msg”**
        - This is a thread function.
        - This function is in charge of listening for a write from the client. In which case, the information written to the server will be read and stored locally in the aforementioned structs.
    - **“new_user”**
        - Takes some client id as a parameter. The function then adds the new user to the user array, an array declared within a *“users_map”* struct.
    - **“listen_for_new_connections”**
        - This function takes some p as a parameter and casts it to a “args” struct, a struct containing the following fields; id, and message.
        - A for loop that waits and accepts connections by adding them to the global *“conneciton_threads”* array is initialized within the function. So as to properly add these connections, locks are utilized.
    - **“play_game”**
        - This function takes a parameter of type *“jason_t.”*
        - The passed parameter contains the questions read in through the use of the Jansson C library for encoding, decoding and manipulating JSON data.
        - With the aid of the Jansson C library, questions, as well as other relevant information (*“question_text”, “difficulty_text”, “category_text”, “correct_answer_len”,* etc) are read in, locally stored, and written to each connected user as found in the *“users_arr.”*
        - As well as writing the previous information to the clients, the incorrect answers are also sent to the connected users. After writing all the relevant information, the function waits 20 seconds, giving the user time to enter their answer before moving on. After the time has elapsed, users are shown the current leaderboard through an invocation of the *“report_standing”* function.
    - **“main”**
        - Relevant variables are declared and assigned.
        - A server socket is declared and initialized. The socket begins to listen.
        - A new connections thread is created and initialized. The thread runs the “listen_for_new_connections” thread function.

        - Jansson is set up so as to begin reading in questions from a specified file.
        - The “play_game” function is invoked.

When sending the leaderboard info, we first sort the users array to get the updated leaderboard, then we send each user a leaderboard with the following info:

1. A header that contains the total number of users so we know how many to read
    a. Length of the username on the leaderboard
    b. The username itself
    c. The score associated with the name

Client:

- Globals
    - WARNING_TIME: The amount of time we give the user before warning them about the timer
    - Questions_log[]: Contains the question history that’s been sent to this client
    - Cur_question_num: The current question we’re displaying
- Structs:
    - Header_t:
        - Message_type: Contains the type of the message we’re sending/receiving
            - _____ (insert stuff here)
        - Num_users: Total number of users (this is used in display_leaderboard)
        - Message_len: Length of the message
        - Username_len: Length of the username
    - Question_t: This is our question log. It is only used for storing questions for history. This is NOT for communicating with the server
        - Question_num: The question’s number
        - Question_text: The question’s text
        - Question_answer[4]: An array holding the possible answers to the question
        - Category_text: The category of the question
        - Difficulty_text: The difficulty of the question
    - Response_header: Contains the metadata for a client’s response to a question
        - Response_type: _____
        - Username_len: Length of the username
    - User_t: Contains the information that represents a user
        - Username: Contains the username
        - Points: Contains the points associated with the user
        - Fd: The connection between the user and server
- Functions

- Display_Leaderboard: Reads in leaderboard information and displays it to the user. This should be called each time after the question is answered
  - Param: Num_users
  - We construct a user_t array to store the user information that will be displayed
  - For Loop 1: For every user, we will read in the following and add it into our user_t array. This will represent all of our users in the game and their respective scores
    - Username length (so we can read in the username)
    - The username itself
    - The points associated with the user that we're reading in
  - For Loop 2: For every user, we display the user and their respective points
- Send_message:
  - Params:
    - Message: a string with the message in it
  - Code:
    - We check whether the text is a valid message, otherwise we yell at the user
    - If our text is valid, we display it and send it to the server
    - We send the message to the server using 3 writes
      - 1. We send a response header
      - 2. We send the username
      - 3. We send the message (this will be 1 char)
- Listen_thread: Listens to messages from the server, no params
  - We keep an infinite loop open since we're running this on a thread
  - In each loop iteration, we check whether we read in a header file or not. If we don't, we continue looping around
  - Once we read in a header file, we check the type of the header, the types follow:
    - Q: This is a question the server is sending to the client
      - We read in the question number, category text, difficulty, question text (along with the lengths for each) and store them into our global questions_log[] array.
      - Once we have all this information, we display the question
      - We then read in the questions from the server (question length, then question text) and store them in our question log
      - We then display the answers and wait for user input

- Main
  - Checks for whether the user provided the necessary input for creating a user on the server
  - If the input is proper, the client will connect to the server and send a header_t file with the response type of “!”. This will tell the server to create a new user in the server’s user array that will represent the client.