Liam and Nathan

Project Overview

For our project, we explore the topics of concurrency, threads, and client-server style distributed systems through the game Jeopardy!™, built to play from the command line with friends.

The game is based around a client-server relationship, where each player runs a client program that communicates with a single server. The server handles the game logic and communication between the clients, while the clients manage the UI and passing along input from the user.

The main aspects of the program are as follows. The server connects to all the clients and sets up individual threads for communicating with each. It then parses the critical game information from the JSON into a uthash map, and distributes it in a neat struct to the connected clients. The client received the current game state and displays the current scores of all the players and the up-to-date game board to the UI. The client whose turn it is lets their user select a question, which is then sent via the server to all the clients. At this point, the buzz-in and answer period begins. Each client has the opportunity to buzz-in and answer (if they buzzed in), but only the earliest client to have buzzed in and answered correctly will get the points for answering correctly. All the buzz-in time and answer data is sent to the server by each client, which determines if the answers are correct using fuzzy matching. The server sends the information on who won the round back to all the clients. Lastly, the server updates the game state and the cycle begins again when the server sends the updated game state to all the clients.

To evaluate our system, we tested the fairness of the timed buzz-in mechanism. We examine the correctness of the buzz-in times by evaluating against expected results and the real ones, and we also check that when players buzz in at approximately the same time, they are first about 50% of the time.

The results of our project are as follows: overall, we accomplished what we set out to do. We have a working Jeopardy game that does everything from parsing questions to handling remote connections through multithreading. While we took a few shortcuts (avoided N-curses, score penalties) in order to focus on the core of our program, we think the core Jeopardy is there and strong. As a testament to this, we spent the past three hours playing Jeopardy against each other rather than actually testing our program. In addition, this project allowed us to learn a lot about multithreading and socket connections in action, building large scale pieces of software and communication and version control. Therefore, we consider this project to be a success.

Design & Implementation

Our system is represented by a server-client relationship; so the two largest technical components are the server and client code. Therefore, there will be one computer (the

host/server) running the game, managing connections, keeping score, etc. There will also be a number of clients who act as a medium of communication between users and the server.

Server:

The server's responsability can be broken down into three parts: creating the Jeopardy game, handling client connections, and running the game. To create the Jeopardy game, the server parses a large JSON file and builds a set of Jeopardy categories, and then five are selected to build a board. Then, the server accepts socket connections, spins up threads, and sets up communication with clients until a game has enough players. Finally, once the game is set up and the players are connected, the server runs a game loop with all the users, reading in inputs and outputting updated game states, until the game is finished and everything is cleaned up.

Creating the Jeopardy game:

Prior to the server being created, a massive JSON file of Jeopardy questions was downloaded from Reddit. Because of the file size, we chose to only take the first 1000 lines, which still left us with ample questions without memory issues. When the server first starts up, it reads in the file into a buffer and parses it one JSON object at a time, as determined by a '}' (which is the terminating brace for a JSON object). At the same time as it reads the file into a buffer, the server creates a hashmap that maps the category username (a string) to a category object (the category). This workaround was necessary because the JSON file does not bound questions of the same category into the same object (and all of the category questions are interspersed). Each JSON object is read by the server and parsed, then the category in the object is looked up in the hashmap. If the category is already in the hashmap and not full, the question is added to the category. If the category is not in the hashmap, then a new category is created, the question is added, and the category is added to the hashmap. The hashmap is then culled to remove any category that doesn't have enough questions from the pool. When a board is created, a random index is selected, and the categories from the pool are selected accordingly (simply by iterating through the hashmap.) Other game variables are instantiated accordingly: num_players = 0, game_over = false, etc.

Handling client connections:

After creating the game board, the server opens up its port for connections and waits. For each client that connects, the server opens up file streams with that client and creates a new thread to handle that client. Once a client connects, the server adds the player to the game, including username and score. The first client to connect is allowed to pick the first question. The server only accepts as many connections as allowed by the game, and starts the game when that number of connections has been achieved. At the end of the game, the server joins the threads again.

Running the game:

Once the game has begun, each thread follows the same loop. First, the game state is sent to each client. If the thread corresponds to the player whose turn it is, the threads reads in the client's question choice. The server performs the corresponding operations: removing the question from the list of remaining questions, determining the correct answer and how many

points should be awarded, etc. Then, the question choice is sent to each other thread so every client can read the question. The server expects a response struct to be sent from each client, even those who didn't buzz in to respond (those structs are simply checked to see if the boolean did_answer is true or not.) This is to simplify server handling and not needing to determine if a message was sent or not, waiting, etc. Every time an answer struct is received, it is added to the thread-safe answer linked list. The answer linked list is a simple, thread safe way of recording all of the responses. The thread that chose the question is then tasked with looping through the list and determining the answer that is both correct and buzzed in the quickest. Whether or not an answer is correct is determined by the Levenshtein algorithm, where an answer is considered correct iff the output of the algorithm is less than half of the answer length (a general heuristic we found surprisingly accurate for types as well as differences in answers ("the neck" vs "neck", "Coriolanus" vs "Coreolanis"). The user that answered the quickest and correctly gets the corresponding points and to chose the next question. If no one responded correctly, then nobody gets points and the previous client gets to choose again. Once the game board is emptied, the game is over and the memory is cleaned up.

Client:

Each client program has many important duties as the interface between user and server. The client must handle all UI output shown to the users, and manage sending relevant user input from stdin to the server via the POSIX socket API. Each user who wants to play the game must run the client.

In order for the client to begin its job, it must set up a connection with the single server. To get the required information, the client requires the host-name of the machine running the server, as well as the port number it is running on (and on a side note, the player's username for the game) to be input via the command line. Once a connection is established, the client sends their username to the server and receives their player ID in response.

The client will then launch the UI loop thread to handle the following IO sequences with the server. In order to keep all of the clients synced to the same point in the game, the client will then wait for the game state to be sent from the server, which will only be sent once all players (determined by a macro) have joined the game. This blocks the client from advancing until the server (and other clients) is ready. In the meantime, a message is shown on the UI that explains the program is stalled, waiting for all players to join the game. While the UI thread handles the important business, the main thread is stalled with an infinite loop on the condition that the game state is ongoing.

From there, the UI loop thread controls the repeated actions of the client program. The first thing that is executed in the loop is the reading of the game struct from the server. This struct is the most important data structure in the client; it holds all the information about the game and its current state, including whether the game is over, whose turn it currently is, names and current scores of all the players, the category titles, all the questions in each category, the values and

answers of those questions, and whether each of those questions has been answered. To make it easier to write all of this information, every string field of the struct has a maximum length (set by a macro), making the game struct a fixed size and thus writable in one POSIX socket call.

After reading the game struct from the server, the client checks the new game state to see if the game is over (all the questions have been answered). If it is, the UI thread calculates the winner of the game from the last known game state and displays a message to the user indicating the final scores and the winner. Finally, it changes the global state variable that holds the main thread in a loop, the main thread exits, and the program terminates. Otherwise, when the game is not over, the UI loop continues by printing the current game state for the user to see.

The client, on its UI thread, first prints the usernames and up-to-date scores of each player found in the game struct. Then, it prints the game board; making nice, evenly-shaped, rectangles that in the first row contain category names, and then the point values of the questions in each column. If a question has already been chosen in a previous round, its point-value is replaced with the string “XXXX” to indicate to users that that question is no longer a valid choice. Following this, the client determines if it should select the current question.

Since the game struct read from the server contains up-to-date information on whose turn it currently is, the client simply checks that information against its own player ID (previously read from the server) to determine if it is their turn to select the question. If it isn’t, then the client waits on the following blocking read to receive the coordinates of the question that was chosen. If it is the client’s turn, then the client sends a prompt on the UI to the user asking for input coordinates of the question they would like to answer (question row represented as A-E and category column as 1-5) repeatedly until the chosen coordinates are valid (coordinates are within range of the game board and the question has not already been answered). The client then sends those coordinates to the server to be distributed to all the clients (including the client who chose the coordinates, so all client actions can remain as in-sync as possible).

When the coordinates are received from the server, the client finds the corresponding question and displays it to the UI. This begins the answering period. For the sake of networking simplicity, we do away with the race condition of multiple clients trying to buzz in at once by making it possible for every client to buzz in and answer the question. But, only the client who buzzed in first will get the points for answering it in order to retain the competitive nature. The buzz-in time is determined by a fetching of the time in seconds since 1/1/1970 as soon as the client detects any input to stdin. Once any client buzzes in, they give the user the chance to answer the question. If no input to stdin is detected within a certain amount of time (performed using `select` with a timeout) during the buzz-in period, then the buzz-in time is filled as -1 to indicate a failure to buzz-in and the client does not give the user an opportunity to answer. The information (whether the user buzzed-in or not) is put into a struct and then sent to the server so it can determine from the correct answers, who buzzed in first and should get the points. We

acknowledge that this system inherently trusts the clients to have a correctly set system clock for this method to be fair.

The server determines the round winner and then sends all the clients the results in an answer struct that contains information on whether anyone answered the question correctly (or even at all), and if so, the user ID of who did answer correctly first. The client receives this information and displays it on the UI to inform its user of the correct answer to the question and who won the points of the now concluded round.

The UI loop then repeats, beginning again with reading an updated game state from the server.

Evaluation

For testing purposes, we chose to test using MathLan computers, both local and over network. Evaluation was done using synchronized commands between different users at different computers.

Testing fuzzy matching for answers:
Accepted “Samuel Coolridge” for “Samuel Taylor Coleridge”
Accepted “west wing” for “the West Wing”
Accepted “Nairobi” for “Nairobi, Kenya”
Accepted “Ivan the Horrible” for “Ivan the Terrible”
Rejected “sequoia” for “a sequoia”

Simultaneous buzzing in:
Two remote connections:
Client 1 3 : 7 Client 2 (after 10 trials)

One remote and one local:
Client 1 5 : 5 Client 2 (after 10 trials)

Games completed with everything functioning correctly (covers a lot of things):
4/4

Stress testing:
Worked great with 8, seg faulted with 10