Finger Pop

Finger pop is a simple word-guessing game that can be player with 3 or more people at the same time.

During the development of this game, major changes have been made to the initial implementation, resulting in four differing implementations. The details of the implementation are given below in the Table 3.1. Each of the implementation has been explained in detail in this section as follows.

# C:\Users\arun\Downloads\Initial Implementation - New Page.jpegImplementation 1: Client-server implementation

## Main Properties

A basic Client-Server model, where the client is essentially a “thin” client. Important functionality such as the management of scores, players and words was kept on the server. The managing of players on the server side was done with a **Player Manager**. A **Score Manager** was responsible for handling scores, where correct guesses would earn 10 points, while a wrong guess would produce a penalty of -5 points. A **Word Manager** was responsible for managing the words that are displayed for every round.

## Gameplay

A minimum number of users are required to start the game. Once the required users are in, the game starts with a basic word of 3 letters. Every person gets a global timer and every correct word guess gets extra time (of 25 seconds). Once the timer reaches zero, the game is over.

Every correct letter guess gives a person 10 points while a wrong guess terminates the game immediately. The person with the highest score wins the game.

## Exemplified Program Flow

Let’s say there are 3 clients, C1, C2 and C3. Once the game is on, C1 presses a letter **“t”.** This enables the client to contact the server with the letter and acquire a **lock.**

Once the lock is acquired, the letter is then processed by word manager which checks if the letter is a part of the currently displayed word. If it is, then the score for the person is updated with the help of the score manager. A message is then sent to all the connected clients with following details:  **“letter”, [positions], PlayerID, Score.**  For example, if the PlayerID is 1, the sample message would be **“t”, [0,4], 1,10,** where t is the letter pressed, o and 4 are positions of the letter in the given word, the player who gets the score of 10. After the operation has completed, the acquired lock is released and the game continues.

Problems

The main reason for this project was to implement any algorithm in a distributed environment. Since this was very simple implementation, most of the issues like concurrency and mutual exclusion were not highly possible, especially due to the server lock mechanism. To make it a bit more complex, multi-threading was employed in the server front to artificially introduce deadlock and employ any algorithm that solves the issue but it did not prove enough of a challenge. Hence we decided to change the implementation slightly to make it a bit more complex and that is a bit more distributed than the previous version.

## Implementation Details

The implementation is split into 3 distinct parts:

* The client or GUI, which provides a game interface for the player.
* The server, which handles concurrent game events.
* The client-server interface, which handles the communication between the client and the server. This part will be discussed together with the client, as they heavily depend on each other for the distributed game.

As can be seen the implementation follows (a variation) of the MVC-pattern.

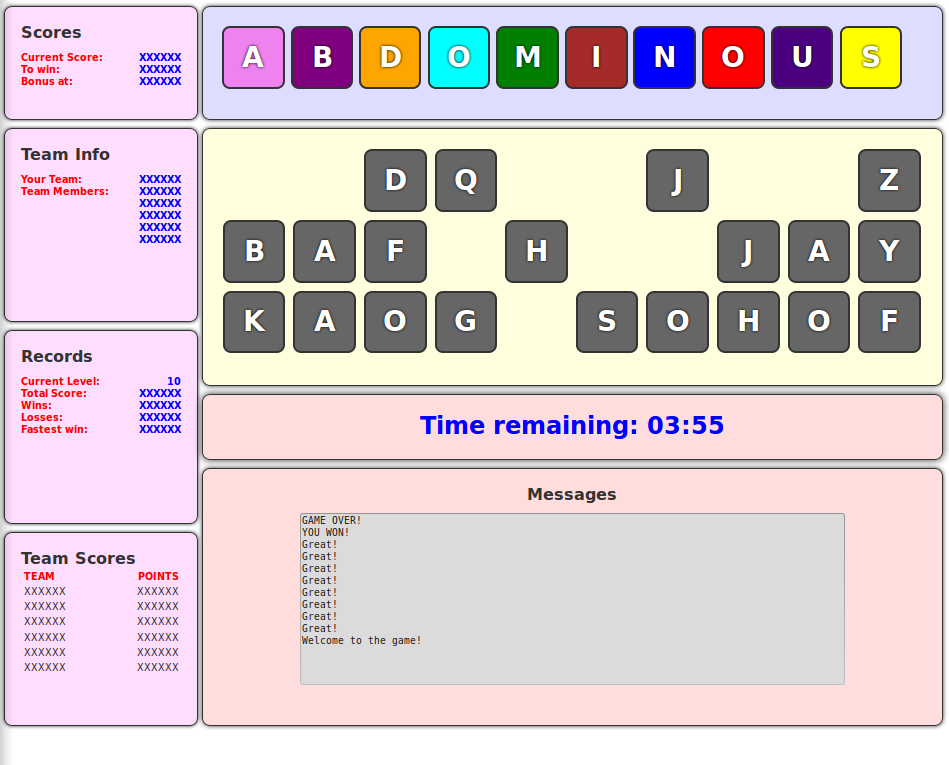
Above a screenshot of the first client prototype can be seen. Note that disabled/unimplemented functions are blanked out using filler data (XXXXX…).

The client functionality (interactive code) was completely coded in Javascript. In order to save implementation time, advanced functionality was coded with the help of popular Javascript libraries based on JQuery.

The styling was done via CSS and HTML. The HTML and CSS components use a fixed sized layout. We experimented with an adaptive layout before, but it turned out that it was too hard (or better yet time- consuming) to produce.

Before the game starts a modal window is used to allow the player to enter a name.

### Client



The client was coded in such a way that it can function with or without the server module. This has several benefits:

* Debugging the client becomes easier, as localized (dummy) functions can be used instead of client-to-server calls.
* Modularization is improved, as server and client can be easily swapped out if required.

Data is interchanged via the client-server interface, where necessary (e.g., making guesses).

As can be seen, the client is split up into several visually distinguished modules:

* The score module, which displays various (local) score information.
* The team info module, which shows information of the team.
* The record module, showing record information.
* The team score module, which shows scores by team.
* The message console module, which can print messages for the user.
* The timer module, which displays how much time is left for the user to make a move.
* The hint module, which contains possible letters that the user can manipulate.
* The word module, which shows the (guessing) progress of the user.

All textual data within the record, team, score and team info module is manipulated via JQuery that targets the contents HTML tags via their labels (id, class). The textual data itself is provided via the server interface, a dummy function or it is disabled.

The timer module was implemented using Javascript via a timed event. This timed event updates the clock and once the clock has reached 00:00 a time-over event is triggered. This event can be defined by the programmer and typically is set to either change the game state to game-over or produce a penalty (score reduction).

The message console module has been implemented using an HTML textbox. Using Javascript this textbox is updated by appending strings to the end of it. The module is useful for informing the user and can also be used for debugging purposes.

The hint and word module work together. The word module shows the status of the current word being guessed by the player(s). The user can in this version make guesses with either a keyboard event or the by dragging and dropping a letter from the hint box/module into a slot in the word box module. When a correct guess is made a programmer-defined event is triggered. Typically this event would (positively) update the score and timer. Incorrect guesses would trigger punishment event (game over or score reduction).

The letters in the hint box can be initialized in multiple ways: random initialization, strict alphabet. Both order and contents can be changed. Successful moves from the hint box to the letter box result in the target letter box to change color, giving the player some visual feedback. The drag-and-drop functionality was realized using JQuery libraries. Correct guesses “stick” to the dragged position and incorrect guesses “bounce” back.

Keyboard events are caught using Javascript. A pressed keyboard will cause a drag-and-drop event to trigger for the appropriate key with the same results, as if the item was dragged there. Mapping the key events to the drag-and-drop functionality had the positive effects of reducing the code complexity and length.

When running in distributed game mode, the client notifies the server via the client-server interface (see above) via socket communication provided by the socket.io library.

When running in distributed game mode, the client has a (socket) listener running in the background to intercept messages from the server (see above). This way keypress or drag-and-drop events (by other players) can be received and displayed.

### Server

The server-code is written entirely in Ruby and communicates via a socket interface. The server handles the game logic as well as the communication between the game participants.

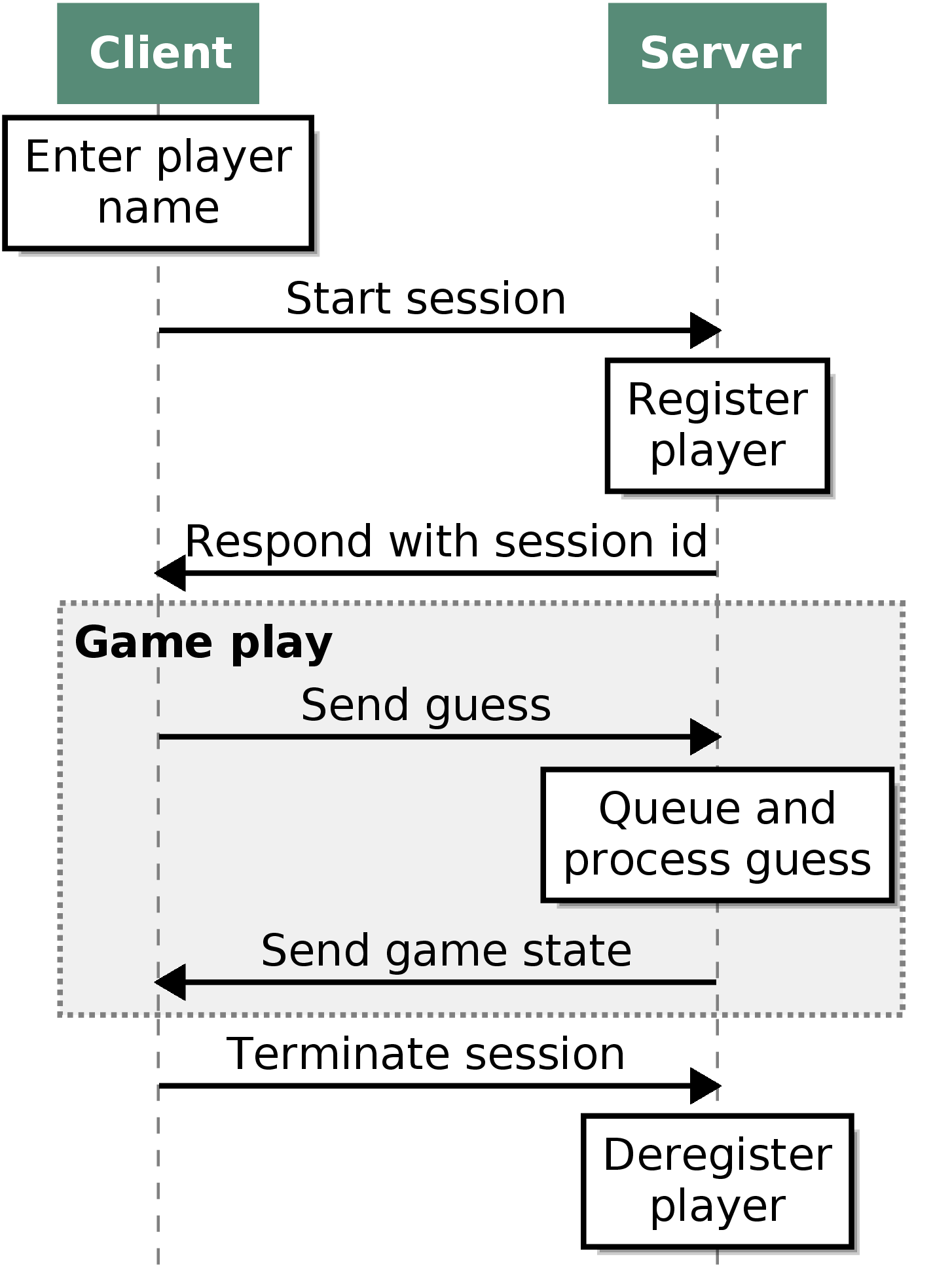
Before a game can be played a new client-server session must be started. Closing a game can be done by signaling that the client-server session should be closed.

Received messages are queued. Concurrent message handlers process queued messages and executable operations (making a guess in this case) require locks. Legal moves are broadcast to all clients.

The game logic is handled via three managers:

* The word manager defines what word to guess via predefined lists. These lists are (per default) randomized. The word manager can be queried with guesses (via a programmatic interface).
* The score manager is an in-memory data store for the scores of all available players.
* The player manager is an in-memory data store for the available players.

### Message Protocol Diagram



# C:\Users\arun\Downloads\Current Implementation - New Page.jpegCurrent/Final Implementation

## Properties

Still a Client-Server model, but the clients are “thick” clients this time. Now the server manages only the players; rather than players, scores and words. The score manager and word manager have been moved to the client-side code. Furthermore, the server also has something called a **Token Manager**, which manages the tokens for its clients.

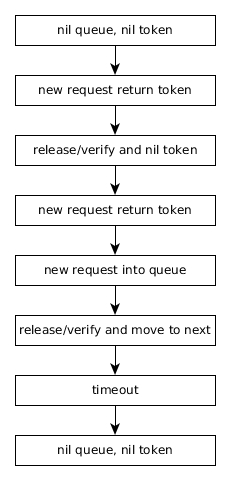
## Gameplay

Still requires a minimum of 3 players to start a game and the timer has been removed. All 26 letters of English are displayed and the players have to guess the word by pressing the keyboard keys only or clicking a letter.

As before, for every wrong suggestion the player gets -5 points, while for a correct guess he earns 10 points. The game does not end when there is a wrong letter press, but keeps going until they decide to quit. The person who has the most number of points at the end of the game wins.

## Exemplified Program Flow

Let us consider C1, C2 and C3 to be the three clients who join the game. Their player IDs are 1, 2 and 3 respectively. C1 presses the letter “t” for the first word that is displayed on the screen. The following will happen:

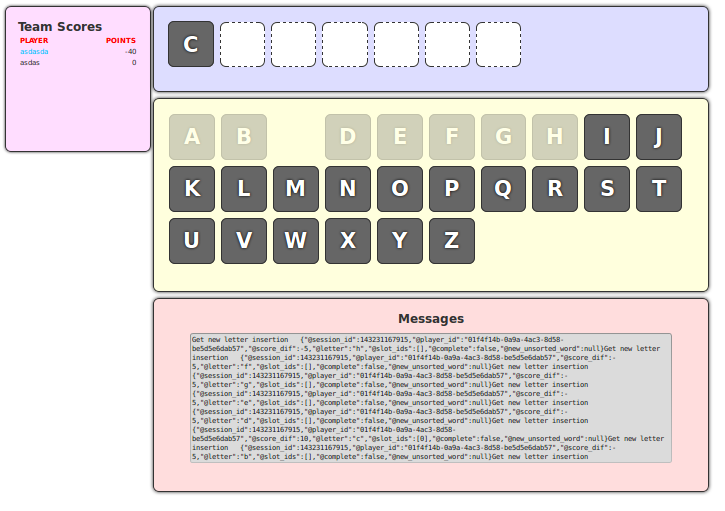
* C1 asks for a token request to the server.
* The server checks the token manager:
  + The token manager has a queue system, where all the token requests are queued.
  + If there is already a Client who requested a token, this request is put below that and once the previous client has released a token, this request is satisfied.
  + Else, the token manager assigns a token for this client.
  + This token manager also employs a time limit, where if a client has not released a given token within 500ms, the token manager shall revoke the token by itself and provide the next client with new token.
* The server communicates the token to the client.
* The client then responds back to the server with a message, which is similar to
  + “token”:xxxxx
  + “message”: { “t”, [0,2], 1, 10} where 1 is player ID and 10 is score.
* The server then broadcasts the message to all clients, at which point the clients update the state by themselves, meaning the words and scores are updated at each client once they receive this broadcast message
* When another client (C2) presses the same letter at same time, and it reaches after the first client’s (C1) token request, the request is put in the queue. If the previous client had successfully updated the letter, and this new client receives the token now, it checks against the word slots in its latest state and if the letter is already in place, the client releases the token without further action.  
    
  On the other hand, even though C1’s request reaches the token first, but due to connection issues, it hasn’t released its token yet, the token manager revokes it and gives access to C2’s request, which will then go one to update the word, thereby getting 10 points.

Notes

* The various actions possible by the token manager can be found by going over the unit tests folder.
* The server still has a thread, but it is used to manage tokens now
* Token management is the utilized algorithm in this game.

## Implementation Details

### Client



The client is based on the previously discussed prototype. Most implementation details are thus the same as previously discussed. Parts of the original GUI have been stripped out. These were removed for one due to lack of time to implement the necessary features. The drag-and-drop mechanic was replaced by a click-mechanic.

The client now takes over the word and score manager functionalities, previously provided by the server.

The messaging system now follows the token management algorithm (see below).

### Server

The new server is similar to the previous server implementation with the exception that the word and score manager functionality was moved to the client.

The locking mechanism was replaced with a token management mechanism (see the following picture).

### Message Protocol Diagram

