

Finger Pop

**Abstract:**

The aim of this project was to identify and solve mutual exclusion and single point of failure problems while developing a multiplayer online game, using distributed algorithms. The team created a multiplayer Hangman-type word guessing game called “Finger Pop”, where each player competes with each other to score highest marks by guessing the letters of a given word. A total of four different implementations (client-server, peer-to-peer without lock, peer-to-peer with lock and peer-to-peer with token ring based algorithm) were incorporated to test if each implementation had the capability to achieve the desired result of solving both mutual exclusion and single point of failure problems. Based on the results, we found that peer-to-peer with ring based algorithm met both conditions successfully. The results were tabulated and provided to the readers for further research opportunities.

**Keywords**: Distributed algorithms, client-server, peer-to-peer, token ring, server lock, mutual exclusion.

**Introduction:**

Finger Pop is a hangman-style word guessing game where three or more players simultaneously compete against each other to guess the letters of a given word to achieve highest scores at the end of the game. The front-end of the game was developed using HTML, CSS and JavaScript while the core game was developed using Ruby. The whole game was version controlled in GitHub. The design decisions, background research, implementation and future improvements form the structure of this document.

**Gameplay:**

The game requires a minimum of 3 players to start a session. All 26 letters of English are displayed as clues on the screen and the players have to guess the word by pressing the keys on their keyboard. Every connected players in a session compete with each other. A correct guess, as decided by the game, earns the player 10 points, while a wrong guess earns him/her minus five points. As a bonus, if the player guessed the last letter of the word, he/she gets 20 points. The game does not end when there is a wrong letter press, but keeps going until the players decide to quit or the number of players drop below 3. The person who has the most number of points at the end of the game is announced the winner.

**Design Decisions:**

Four different types of implementations were carried out for this specific game in order to solve the mutual exclusion problem and single point of failure issue. The initial implementation was a client-server model which solved the mutual exclusion problem, but since the whole game relied on server, it was obviously a single point of failure. The next implementation was a peer-to-peer implementation, but this implementation suffered from mutual exclusion problem. Hence to solve that issue, the team implemented a “Server-Lock” algorithm which basically provides lock capability in the central server, which controls simultaneous access to critical section of the game. But implementing a lock in a central server again resulted in the issue of single point of failure, that is, when the server crashed, the game stopped working.

To overcome both issues, we finally decided to implement a token ring based algorithm in a peer-to-peer architecture, which effectively solved both mutual exclusion and single point of failure issues.

# Implementations:

# Implementation 1: Client-server implementation

## C:\Users\arun\Downloads\Initial Implementation - New Page.jpegMain Properties

**Figure 6.1**

A basic Client-Server model, where the client was 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

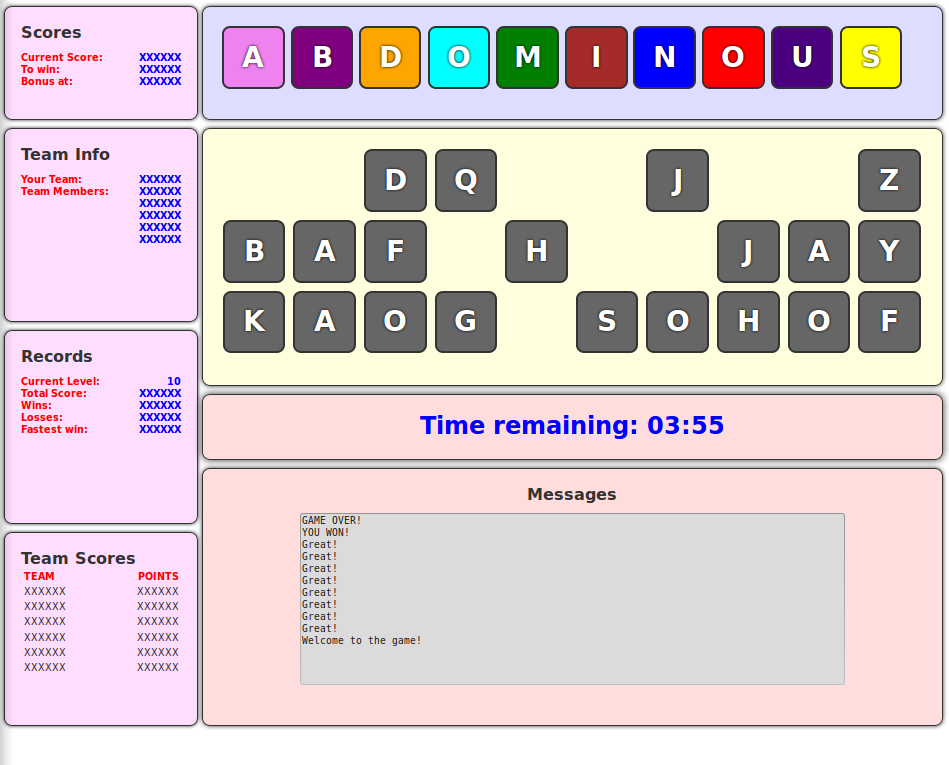
This initial implementation had a different gameplay compared to the other implementations. A minimum of three users were required to start the game. Once the required users were in, the game started with a basic word of 3 letters. Every person got a global timer and every correct word guess got extra time (of 25 seconds) for the player. Once the timer reached zero, the game was considered over. Every correct letter guess gave a person 10 points while a wrong guess terminated the game immediately. The person with the highest score was announce as winner of the game.

## Exemplified Program Flow

Assuming there were three clients namely C1, C2 and C3, once the game was on, C1 pressed a letter **“t”.** This enabled the client to contact the server with the letter and acquire a **lock.** Once the lock was acquired, the letter was then processed by word manager which checked if the letter was part of the currently displayed word. If it was, then the score for the person was updated with the help of the score manager. A message was sent to all the connected clients with following details:

**“letter”, [positions], PlayerID, Score.**

For example, if the PlayerID was 1, the sample message would be **“t”, [0,4], 1,10,** where t is the letter pressed, 0 and 4 being positions of the letter in the given word and the player who got the score of 10. After the operation had completed, the acquired lock was released and the game continued.



**Figure 6.2**

## 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.

**Client:**

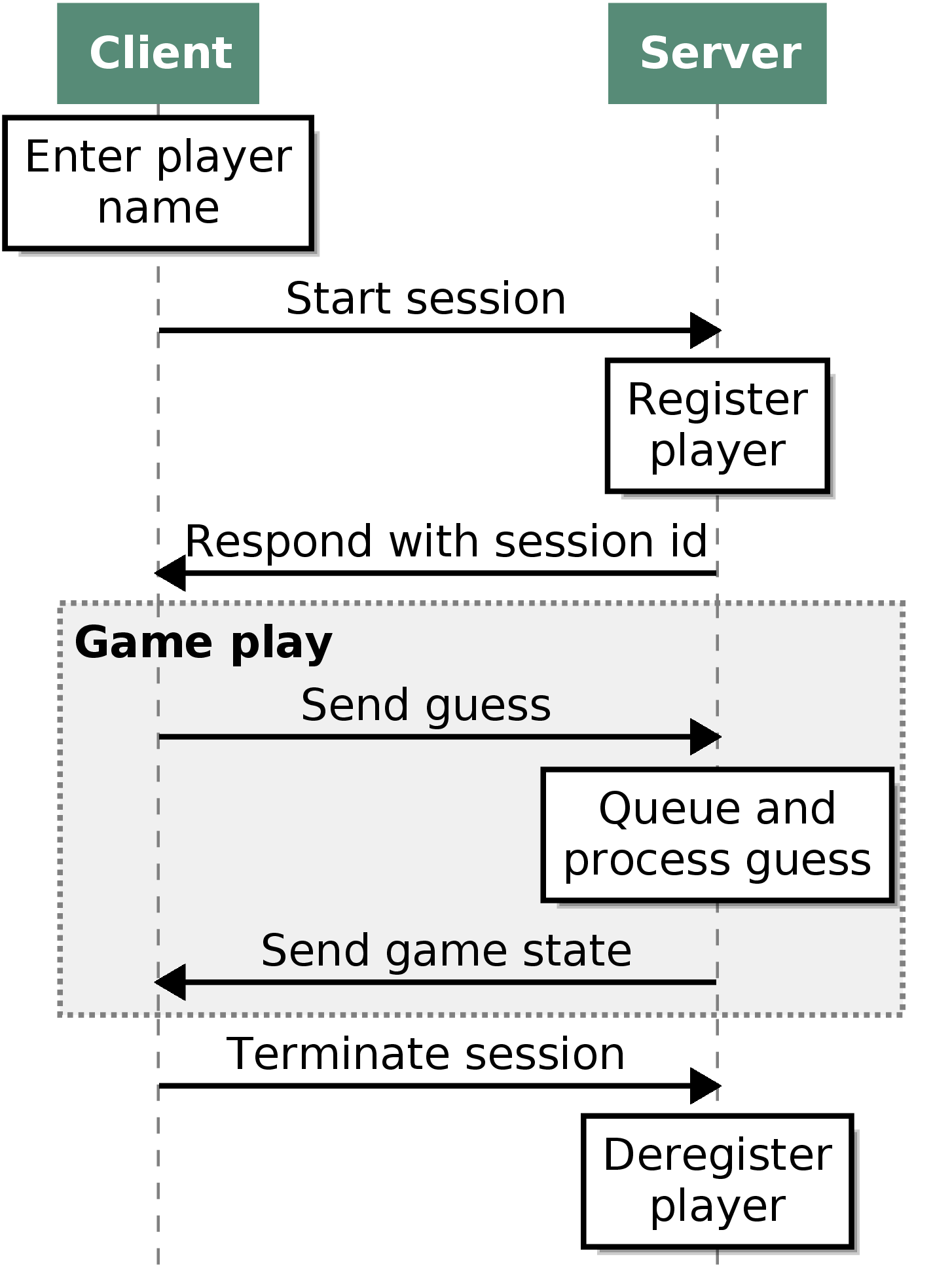
As can be seen the implementation followed (a variation) of the MVC-pattern. Figure 6.2 shows the client prototype for this implementation. Note that disabled/unimplemented functions were blanked out using filler data. The client functionality (interactive code) was completely coded in JavaScript. In 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. The team experimented with an adaptive layout before, but it turned out that it was time consuming and did not yield any better user experience .Before the game starts a modal window is used to allow the player to enter a name.

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

* Debugging the client became easier, as localized (dummy) functions can be used instead of client-to-server calls.
* Modularization was 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). From the Figure 6.2 it is evident that the client was split up into several visually distinguished modules such as

* 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 was manipulated via JQuery that targeted the contents HTML tags via their labels (id, class). The textual data itself was provided via the server interface. The timer module was implemented using JavaScript via a timed event. This timed event updated the clock and once the clock reached 00:00 a time-over event was triggered. This event can be defined by the programmer and typically was set to either change the game state to game-over or produce a penalty (score reduction). The message console module was implemented using an HTML textbox. Using JavaScript, this textbox was updated by appending strings to the end of it. The module was 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).

**Figure 6.3: Message Process Diagram**

The letters in the hint box could be initialized in multiple ways such as random initialization, strict alphabet. Both order and contents could be changed. Successful moves from the hint box to the letter box would 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 “stuck” to the dragged position and incorrect guesses “bounced” back. Keyboard events were caught using JavaScript. A pressed keyboard would 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 notified the server via the client-server interface, as given in Figure 6.3, via socket communication provided by the socket.io library. When running in distributed game mode, the client had a (socket) listener running in the background to intercept messages from the server. This way key press or drag-and-drop events (by other players were received and displayed.

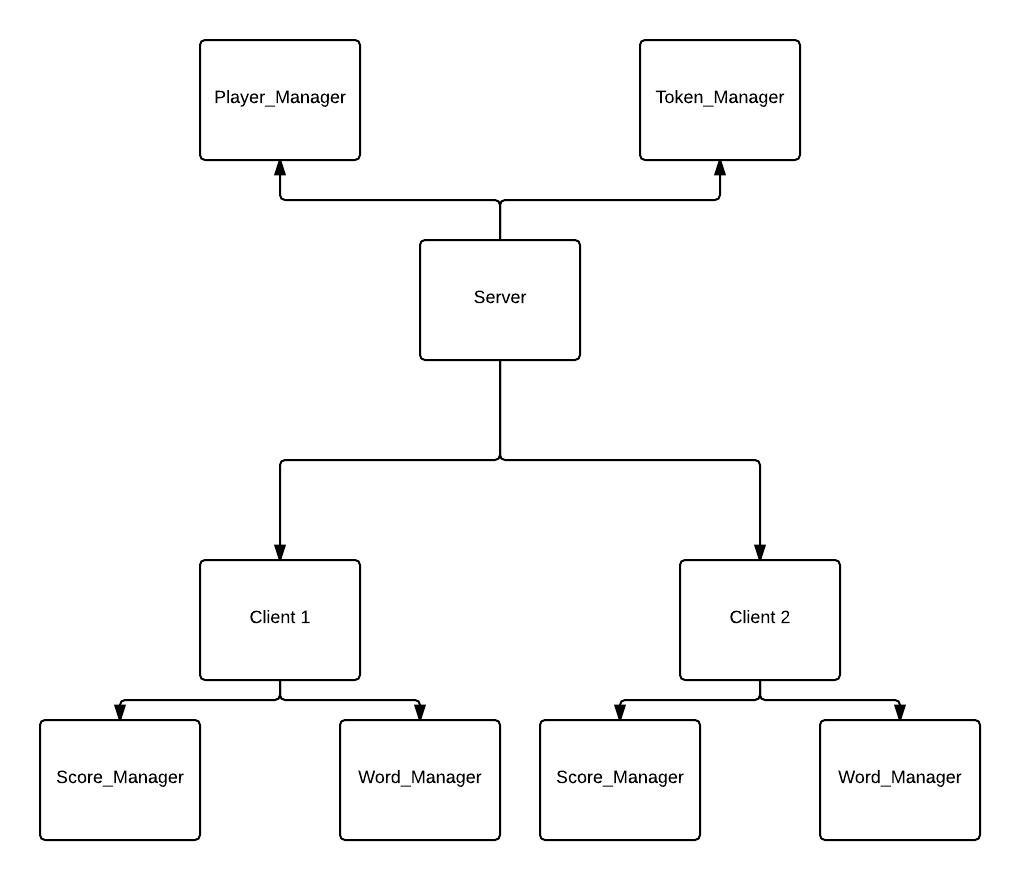
### Server

The server-code is written entirely in Ruby and is communicated via a socket interface. The server handled 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 were queued. Concurrent message handlers processed queued messages and executable operations (making a guess in this case) require locks. Legal moves were broadcast to all clients.The game’s logic was 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.

Implementation 2: Peer-to-peer without lock:

In this implementation, the clients were created to be “thick clients”, meaning most of the processing was moved to client side. The server managed only the players, rather than players, scores and words. The score manager and word manager was moved to the client-side code. Furthermore, the server also has something called a **Token Manager**, which managed the tokens for its clients.

**Exemplified Program Flow**

**Figure 6.4: Architecture Diagram**

Assuming C1, C2 and C3 to be the three clients who joined the game, they were assigned player IDs of 1, 2 and 3 respectively. If C1 pressed the letter “t” for the first word that is displayed on the screen, it basically asked for a token request to the server. The server would then check the token manager which has the following attributes:

* + 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 would then communicate the token to the client. The client then would respond back to the server with a message, which is similar to

**“token”:xxxxx**

**“message”: { “t”, [0,2], 1, 10}**

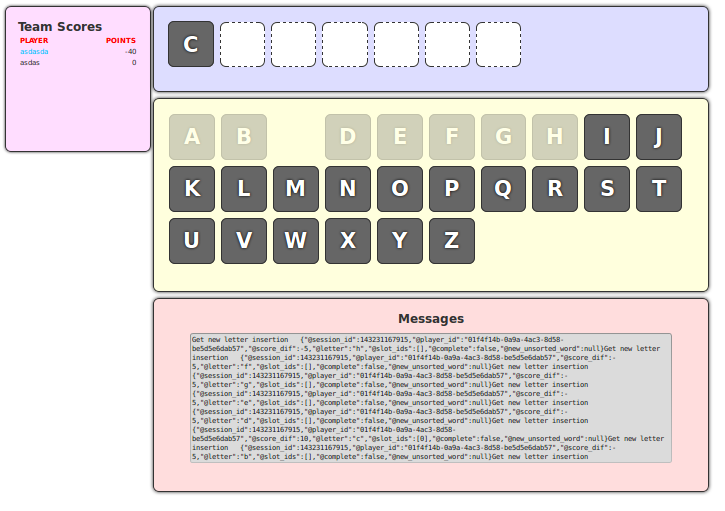
Where ‘1’ is player ID and 10 is score. The server then would broadcast the message to all clients, at which point the clients were expected to update the state by themselves, meaning the words and scores were updated at each client once they received this broadcast message. If another client (C2) pressed the same letter at same time, and it reached after the first client’s (C1) token request, the request would be put in the queue. If the previous client had successfully updated the letter, and this new client received 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 reached the token first, but due to connection issues, it hadn’t released its token yet, the token manager would revoke it and give access to C2’s request, which would then go one to update the word, thereby getting 10 points.

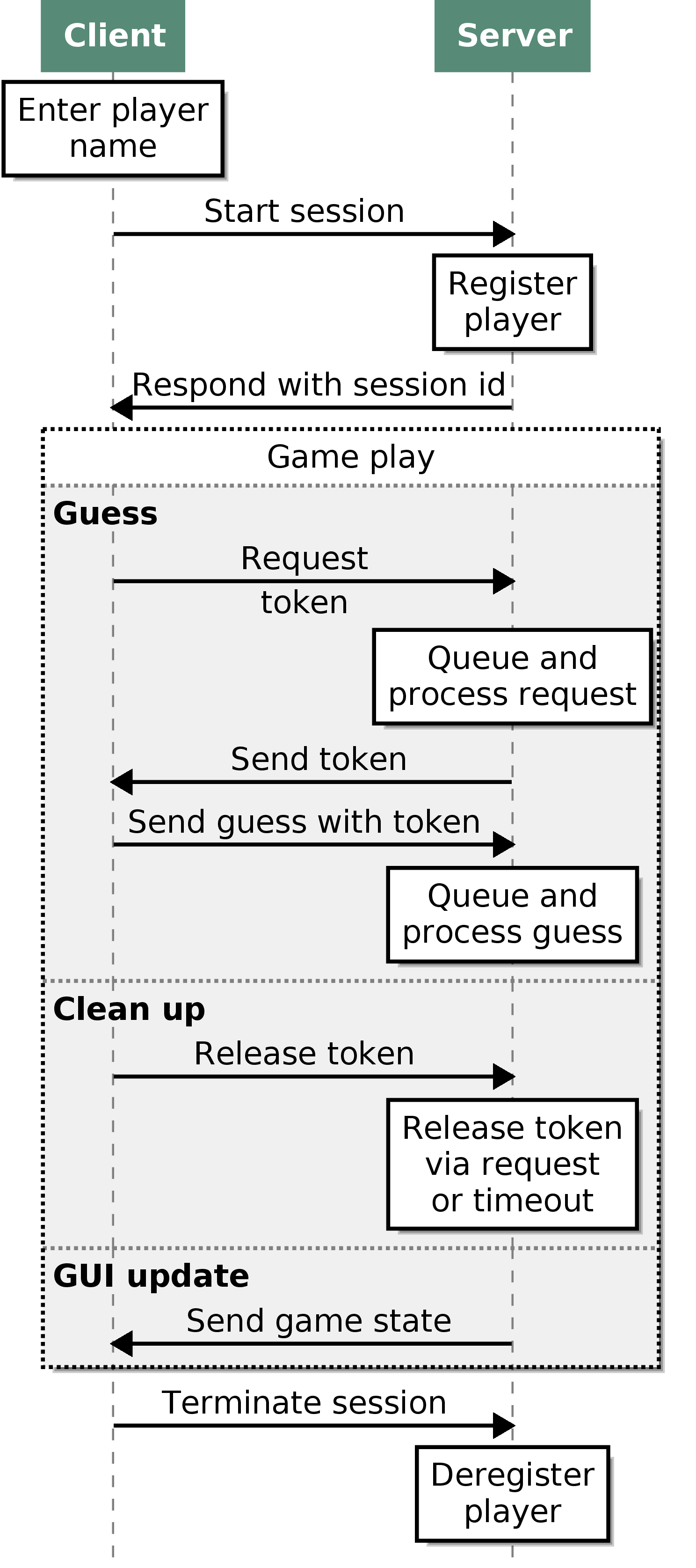
**Implementation Details**

### Client

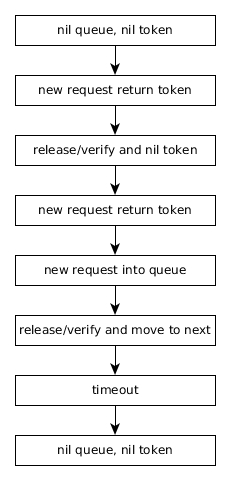
The client was based on implementation one. Most implementation details were thus the same as previously discussed. Parts of the original GUI had been stripped out. These were removed for one due to lack of time to implement the necessary features. The drag-and-drop mechanic was replaced by a click-mechanic. The client was designed to take over the word and score manager functionalities, previously provided by the server. The messaging system now followed the token management algorithm as explained in Figure 6.4.



**Figure 6.5:Front-end of Implementation 2**



**Figure 6.6: Message process of Implementation 2**

****

**Figure 6.4: Program flow**

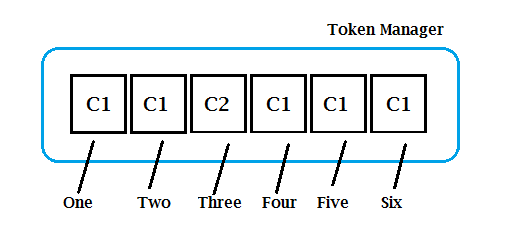
**Server:**

The new server was made similar to the implementation one with the exception that the word and score manager functionality were moved to the client. The locking mechanism was replaced with a token management mechanism, as described in the Figure 6.6.

**Discussions and future work:**

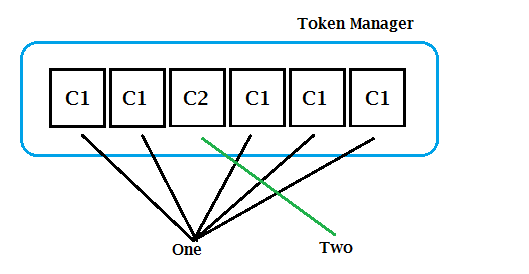
The central server token based algorithm could be improved by implementing few modifications in implementation level, which would in turn improve the speed at which the tokens are processed and served to the clients.

The current configuration of token access by the clients are given in the Figure 7.1 below. Here, the server process each request separately and grants access to them based on the order of the request received. The advantage of this implementation is that, the properties of mutual exclusion are persevered (safety, liveliness and ordering) but the drawback is that it can take a longer time to process each request and the lag becomes noticeable when there is a heavy load. In the Figure 7.1, the terms one, two, three etc. are the order in which the server processes each token request and grants permission to modification of word.



**Figure 7.1: Current implementation**

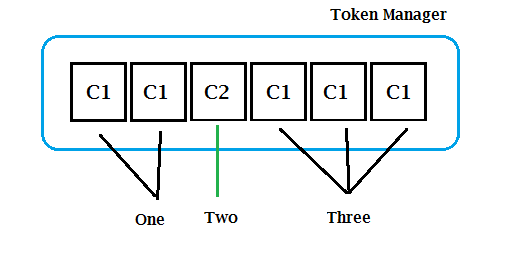
This basic algorithm could be modified in two ways to overcome this drawback. The first method is by grouping all the requests received by a client and then granting access to it, letting the client to make all the changes it requires, before moving on to next client in queue in the token manager. If C1 and C2 are two clients, and the order in which the requests come to token manager are: “***C1, C1, C2, C1, C1, C1***”, then the server shall group all C1’s requests together and grant it access first, before letting C2 make any change as described in the Figure 7.2.



**Figure 7.2: Method One**

This will significantly reduce the load on the server by cutting the number of requests to be processed to dramatically low count, hence increasing the updating speed. While this may be a good option to increase the speed, the drawback is, this does not comply with the basic property of mutual exclusion, which is ordering. It is very essential to preserve ordering of the requests in the queue to make sure every client gets a fair share of processing from the server.

To overcome this drawback, a slight modification can be done in the grouping stage of the messages, thus making the algorithm comply with mutual exclusion properties, while increasing the update speed and reducing the processing load on server. The requests from each client are still processed together as previous algorithm, but only until a new client’s request is found in the queue, as given in the Figure 7.3. If C1 and C2 are two clients, and C1 makes two requests, while third request is of C2, and the rest of the requests are from C1, the server process the first two requests from C1 as a single request, grants it permission to make changes. Once the token is released from C1, the server allocates C2 a token and once it is released, the server then treats the remaining requests from C1 as a single request and grants it permission to make changes as a whole. This considerably reduces the overall processing time, hence making the game faster.



**Figure 7.3: Method 2**

Another area for improvement is the token request-grant method from the central server. In current implementation, as given in Figure 7.4, the central server manages all the token request, release and token grant, which puts significant load on the server and a certain loss of time during heavy load. As an alternative, another optimized version of centralized token algorithm using a central coordinator [1] could be used, where the token requests are sent to the server, which then sends a notification with the details of the node requesting access (N2), to the node which currently has the token (N1). The current node then releases the token to the new requested node (N2) directly, instead of going through the server, hence reducing the time and network usage significantly, as given in Figure 7.5.

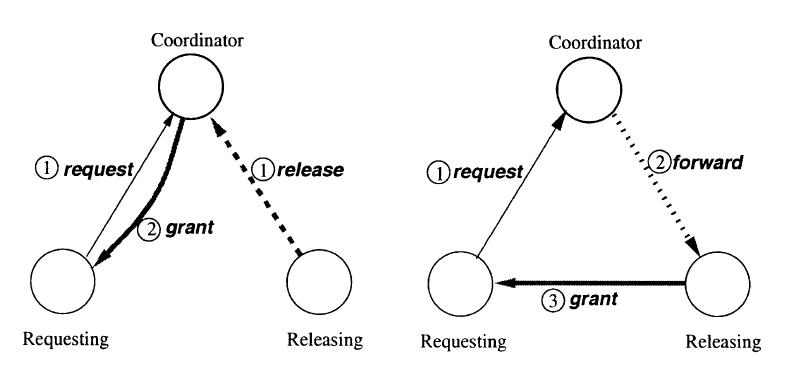


Figure: 7.4: Current Implementation Figure 7.5: Alternative Implementation