Module Guide for SFWRENG 4G06: Dice Duels: Duel of the Eights

Team 9, dice_devs
John Popovici
Nigel Moses
Naishan Guo
Hemraj Bhatt
Isaac Giles

January 17, 2025

1 Revision History

Table 1: Revision History

Date	**Developer(s)**	**Change**
2025-01-08	Hemraj Bhatt	Added content to sections 4.1 & 4.2
2025-01-11	John Popovici	Formatted section 2; Added content to sections 4.1 & 4.2
2025-01-11	Hemraj Bhatt	Updated content on section 4
2025-01-14	Hemraj Bhatt	Updated content on section 3
2025-01-15	John Popovici	Added content to section 10
2025-01-15	Nigel Moses	Added content to sections 5 & 10
2025-01-16	Isaac Giles	Added content to sections 6 & 8
2025-01-16	Hemraj Bhatt	Added content to section 12

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
DAG	Directed Acyclic Graph
M	Module
MG	Module Guide
OS	Operating System
R	Requirement
SC	Scientific Computing
SRS	Software Requirements Specification
SFWRENG 4G06	Explanation of program name
UC	Unlikely Change
[etc. —SS]	[... —SS]

See SRS Documentation for any additional.

Contents

List of Tables

List of Figures

3 Introduction

Template introduction refactored. A few wording changes.

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (**?**). For the Dual of the Eights project, we advocate a decomposition based on the principle of information hiding (**?**). This principle supports design for change, as the "secrets" that each module hides represent likely future changes. This approach is particularly valuable in the context of Dual of the Eights, where modifications to game rules, mechanics, or visual elements are likely during development and playtesting phases.

Our design follows the guidelines laid out by **?**, as follows:

- System details that are likely to change independently, such as scoring algorithms, dice physics, or player health mechanics, should be the secrets of separate modules.
- Each data structure, such as those representing dice, players, or game states, is implemented in only one module.
- Any other module requiring information stored in a module's data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the Software Requirements Specification (SRS), the Module Guide (MG) is developed (**?**). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.
- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.
- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility, and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as follows. Section 4 lists the anticipated and unlikely changes of the software requirements. Section 5 summarizes the module decomposition that was constructed according to the likely changes. Section 6 specifies the connections between the software requirements and the modules. Section 7 gives a detailed description

of the modules. Section 8 includes two traceability matrices. One checks the completeness of the design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section 9 describes the use relation between modules.

4 Anticipated and Unlikely Changes

This section lists possible changes to the system. According to the likeliness of the change, the possible changes are classified into two categories. Anticipated changes are listed in Section 4.1, and unlikely changes are listed in Section 4.2.

4.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. The approach adapted here is called design for change.

AC1: Operating System: Godot makes porting the game to other operating systems simpler, but specific interfaces may have to change and testing would be required.

AC2: Hardware: The specific hardware on which the software runs is expected to evolve, particularly as the online multiplayer functionality will require a server to enable long-distance gameplay. This aspect of the project is subject to change based on cost, performance, and scalability requirements.

AC3: User Input: The format of the initial input data is expected to evolve to accommodate users in operating the game and setting game rules. These changes will be made to minimize user errors and facilitate a smoother gameplay experience.

AC4: File Type: The file structure for saving game states and related data is expected to evolve. A file structure that ensures efficient storage and facilitates quick retrieval of game states is expected to be used.

AC5: Scoring: The scoring calculations may be modified based on usability and player testing.

AC6: User Interface (UI): A rudimentary UI is currently used for development. The UI will need to be updated to accommodate the addition of animations and will have to be refreshed in order to look more appealing once the game's major features are complete.

AC7: Dice: Dice types can continually be added and modified as they use a simple interface and if added are simple to integrate within the larger system.

4.2 Unlikely Changes

The module design should be as general as possible. However, a general system is more complex. Sometimes this complexity is not necessary. Fixing some design decisions at the system architecture stage can simplify the software design. If these decision should later need to be changed, then many parts of the design will potentially need to be modified. Hence, it is not intended that these decisions will be changed.

UC1: Input Devices: The game is designed to play on a computer, i.e. it is designed to work with a keyboard and mouse. It is unlikely that the game will be playable through other input means such as controllers.

UC2: Output Devices: The game is designed to play on a computer, i.e. it is designed to display correctly on a computer screen. It is unlikely that the game will be playable through other devices who have vastly different screen sizes such as phones.

UC3: Game Types: Despite being likely changes in the SRS stage of development, once formed, the game types designed would require massive module changes if they were to further be modified.

5 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

The modules are categorized into three types: Hardware-Hiding, Behavior-Hiding, and Software Decision modules. Below is the hierarchy:

5.1 Hardware-Hiding Modules

- **NetworkManager2P Module:** Manages connection and synchronization for two-player games.
- **GameUI Module:** Provides the interface for interacting with the game and displaying relevant data.

5.2 Behavior-Hiding Modules

- **MultiGameManager Module:** Manages the sequence of multiple Yahtzee games and customization phases.
- **PlayerManager Module:** Tracks player states, scores, and upgrades.
- **GameManager Module:** Handles a single game of Yahtzee.
- **GameSettings Module:** Loads and stores settings for this Yahtzee variant.
- **CustomizationMenu Module:** Implements dice and game customization between games.
- **DynamicScoreboard Module:** Tracks and displays scores dynamically.
- **CustomBaseDie Module:** Handles the 3D dice models, textures, and physics.
- **DynamicDiceContainer Module:** Manages dice interactions and rendering.

5.3 Software Decision Modules

- **ScoreCalculator Module:** Calculates scores for dice rolls based on Yahtzee rules and custom modifiers.

6 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in Section 7.

Requirement ID	**Requirement Description**	**Module(s) Involved**	**Design Decision**
R1	Support for online player vs player mode.	NetworkManager2P	Peer-to-peer communication and synchronization are handled by the NetworkManager2P module.
R2	Handle score calculations using Yahtzee rules.	ScoreCalculator	The ScoreCalculator module handles scoring algorithms for standard and non-standard Yahtzee rules.
R3	Simulate realistic physics for 3D dice rolls.	CustomBaseDie, DynamicDiceContainer	Physics for dice rolls are managed by the CustomBaseDie module, and the DynamicDiceContainer deals with dice rendering and states.
R4	Use the real outcome of a roll to get dice values.	CustomBaseDie	The CustomBaseDie module ensures that roll outcomes are derived from physics-based interactions and values are read directly.
R5	Support for dice with different numbers of sides.	CustomBaseDie	The CustomBaseDie module allows updates to dice textures and supports multiple dice variations/configurations.
R6	Implement simultaneous turn-based gameplay.	GameManager, NetworkManager2P	The GameManager controls game flow, while the NetworkManager2P ensures synchronization of simultaneous turns.

Table 2: Connection Between Requirements and Modules (Part 1)

Requirement ID	Requirement Description	Module(s) Involved	Design Decision
R7	Allow players to pick and omit dice for each roll.	GameManager, DynamicDiceContainer	The GameManager tracks player decisions, and the DynamicDiceContainer updates dice states accordingly.
R8	Display a user interface with scores, dice states, and player information.	GameUI, DynamicScoreboard	The GameUI presents game information, while the DynamicScoreboard provides real-time score updates.
R9	Provide controls for modifying game settings.	GameSettings, CustomizationMenu	The GameSettings module stores configurations, and the CustomizationMenu allows changes to these configurations.
R10	Provide presets for different game modes.	GameSettings	Presets are stored and retrieved through the GameSettings module.
R17	Always show the correct current state accurately.	GameUI, GameManager	State synchronization is managed by the GameManager, while GameUI reflects the current state.
NFR2	Implement a clear and easy-to-use interface.	GameUI	The GameUI module prioritizes usability through intuitive design.
NFR6	Ensure modular, easily extendable codebase.	All Modules	Modular decomposition of functionality ensures extensibility and maintainability.
NFR9	Maintain a consistent UI and 3D visual style.	GameUI, DynamicDiceContainer	Style guidelines and visual coherence are maintained across all UI and gameplay elements.

Table 3: Connection Between Requirements and Modules (Part 2)

7 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by **?**. The *Secrets* field in a module decomposition is a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it. For each module, a suggestion for the implementing software is given under the *Implemented By* title. If the entry is *OS*, this means that the module is provided by the operating system or by standard programming language libraries. *SFWRENG 4G06* means the module will be implemented by the SFWRENG 4G06 software.

7.1 Hardware-Hiding Modules

7.1.1 NetworkManager2P Module

Secrets: The underlying implementation of peer-to-peer communication and synchronization between two clients.
Services: Handles connection setup, data synchronization, and disconnection handling for a 2-player game.
Implemented By: SFWRENG 4G06.

7.1.2 GameUI Module

Secrets: The logic for UI components and interactions.
Services: Displays information to players, including scores and customization options, and provides action buttons.
Implemented By: SFWRENG 4G06.

7.2 Behavior-Hiding Modules

7.2.1 MultiGameManager Module

Secrets: The logic for managing multiple games, including customization and upgrades between games.
Services: Tracks progress and integrates player upgrades into gameplay.
Implemented By: SFWRENG 4G06.

7.2.2 PlayerManager Module

Secrets: The data structure for tracking player-specific details, including scores, dice, and upgrades.
Services: Tracks and updates player states, including consumables and modifiers.
Implemented By: SFWRENG 4G06.

7.2.3 GameManager Module

Secrets: The logic for managing the flow of the game.
Services: Tracks game progress and controls flow and state of the game.
Implemented By: SFWRENG 4G06.

7.2.4 GameSettings Module

Secrets: Configuration storage and retrieval.
Services: Stores and loads game settings, including the number of games and customization options.
Implemented By: SFWRENG 4G06.

7.2.5 CustomizationMenu Module

Secrets: The logic for presenting and applying customization options between games.
Services: Displays upgrade options, enforces selection rules, and updates player dice and modifiers.
Implemented By: SFWRENG 4G06.

7.2.6 DynamicScoreboard Module

Secrets: The algorithms for dynamically generating score displays.
Services: Updates and displays scores in real-time during gameplay.
Implemented By: SFWRENG 4G06.

7.2.7 CustomBaseDie Module

Secrets: The interaction logic and rendering of custom dice, including dynamic face changes.
Services: Manages dice texture changes based on player customization and handles physics and collision for rolling.
Implemented By: SFWRENG 4G06.

7.2.8 DynamicDiceContainer Module

Secrets: The algorithms for managing a collection of dice and their rendering.
Services: Dynamically creates and organizes dice based on game settings and updates dice states during gameplay.
Implemented By: SFWRENG 4G06.

7.3 Software Decision Modules

7.3.1 ScoreCalculator Module

Secrets: The scoring algorithms based on Yahtzee rules and custom modifiers.
Services: Calculates scores for dice rolls and applies passive modifiers.
Implemented By: SFWRENG 4G06.

8 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements and between the modules and the anticipated changes.

Req.	Modules
R1	NetworkManager2P, GameManager
R2	ScoreCalculator
R3	CustomBaseDie, DynamicDiceContainer
R4	CustomBaseDie
R5	CustomBaseDie
R6	GameManager, NetworkManager2P
R7	GameManager, DynamicDiceContainer
R8	GameUI, DynamicScoreboard
R9	GameSettings, CustomizationMenu
R10	GameSettings
R17	GameManager, GameUI

Table 4: Trace Between Requirements and Modules

AC	Modules
AC1	GameUI, NetworkManager2P
AC2	NetworkManager2P
AC3	GameUI, GameSettings
AC4	GameManager, GameSettings
AC5	ScoreCalculator
AC6	GameUI
AC7	CustomBaseDie, DynamicDiceContainer

Table 5: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. **?** said of two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. Figure 1 illustrates the use relation between the modules. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels.

[The uses relation is not a data flow diagram. In the code there will often be an import statement in module A when it directly uses module B. Module B provides the services that module A needs. The code for module A needs to be able to see these services (hence the import statement). Since the uses relation is transitive, there is a use relation without an import, but the arrows in the diagram typically correspond to the presence of import statement. —SS]

[If module A uses module B, the arrow is directed from A to B. —SS]

Figure 1: Use hierarchy among modules

10 User Interfaces

Figure 2: Main Menu User Interface Sketch

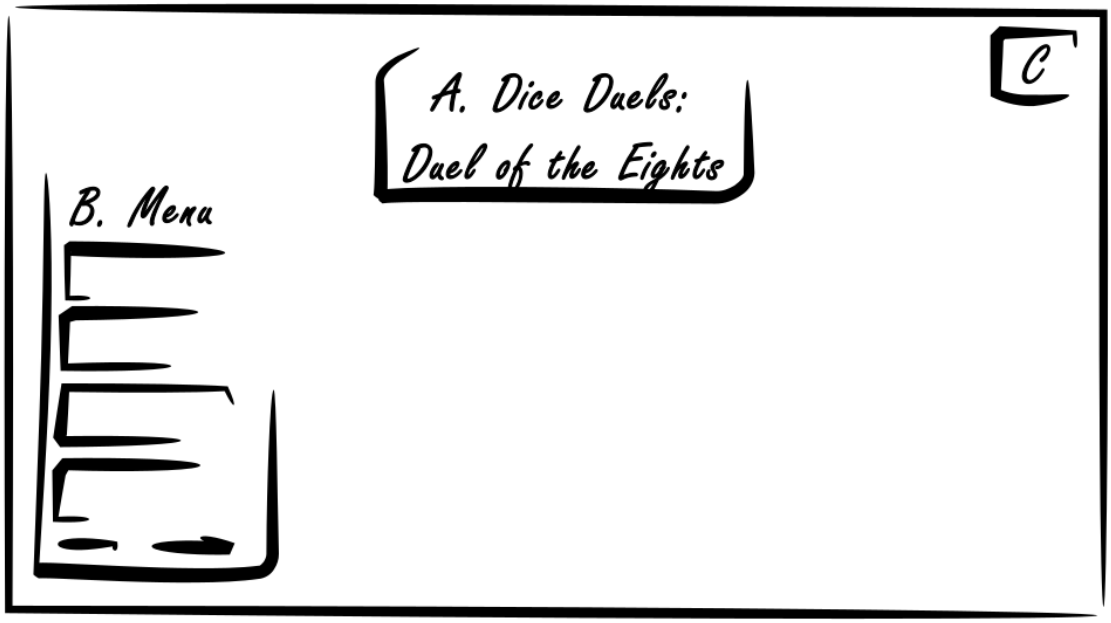


Figure 3: Settings Menu User Interface Sketch

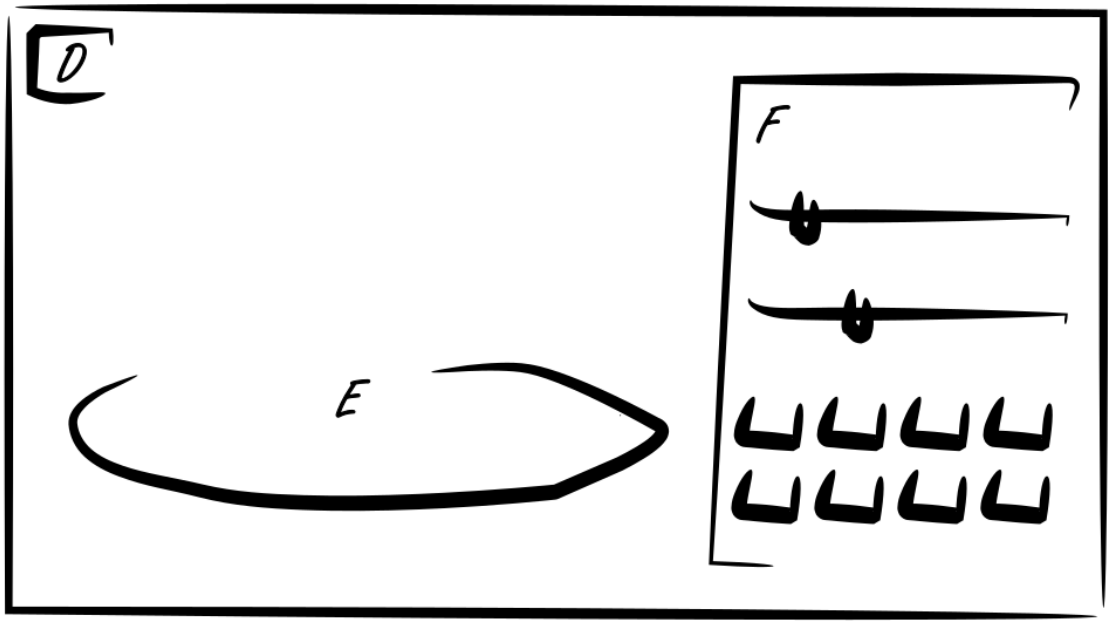


Figure 4: Game User Interface Sketch

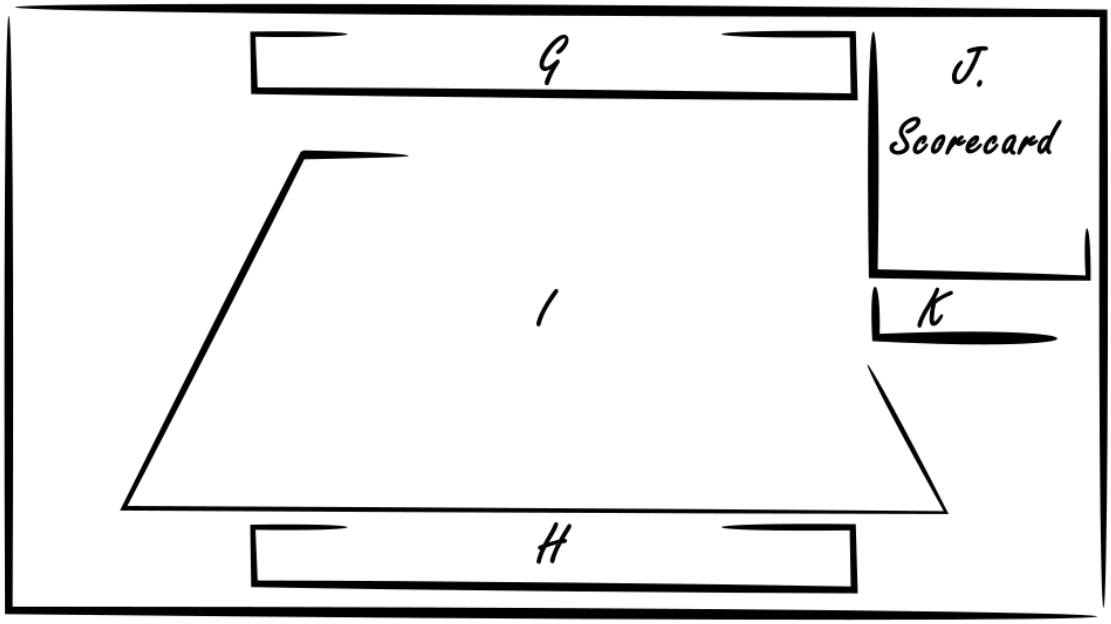
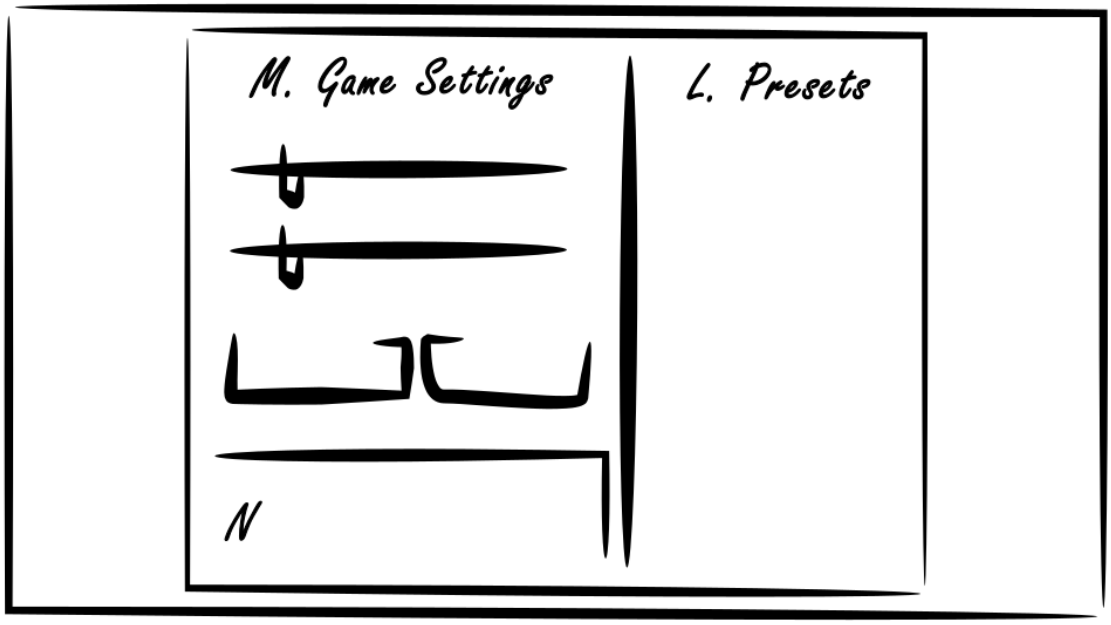


Figure 5: Game Creation User Interface Sketch



Elements of the main menu user interface:

- A: Title and game name
- B: Menu and game options
- C: Options and settings

Elements of the settings menu user interface:

- D: Back to main menu
- E: Dice example display
- F: Customization options

Elements of the game user interface:

- G: Opponent display dice information
- H: Player display dice information
- I: Game board display
- J: Scorecard and score
- K: Game turn controls

Elements of the game creation user interface:

- L: Game presets controls
- M: Customizable game settings
- N: Game connection and start information

11 Design of Communication Protocols

The communication protocols for this project leverage Godot's built-in multiplayer networking libraries to facilitate data synchronization and interactions between players. The design supports two key modes of communication: local area network (LAN) and online multiplayer through a server-based hosting system.

11.1 Use of Godot Multiplayer Networking Libraries

Godot's multiplayer networking system provides the foundation for real-time communication between players. The networking stack includes:

- **ENet Protocol:** A reliable UDP-based protocol for efficient and low-latency data transmission.
- **@rpc and @rpc("any_peer") Functions:** Remote Procedure Calls (RPCs) are used to synchronize state and transmit data between connected peers. The `@rpc` and `@rpc("any_peer")` functions allow for targeted and efficient communication.

The networking manager ('networkManager2P') is designed to handle peer-to-peer communication for two players over a LAN or via an online server. The system uses a host-client architecture, where the host player is responsible for maintaining the game state and transmitting updates to connected clients.

11.2 Server-Based Hosting for Online Multiplayer

For online multiplayer, a server will be implemented to host game lobbies for players who are not on the same network. The server's responsibilities include:

- Managing multiple lobbies and connections simultaneously.
- Authenticating and assigning unique identifiers to each connected player.
- Relaying data between the host and clients, ensuring seamless communication.

This server-based approach allows players from different locations to connect and ensures a consistent experience by maintaining a central point of synchronization for game state.

11.3 Types of Data Passed Between Players

The communication protocol is designed to handle various types of data to synchronize game state and ensure fair gameplay. Based on the 'networkManager2P' module, the following types of data are transmitted:

- **Game State Updates:** Includes information about the current round, dice rolls, and scores. These updates ensure both players are in sync at all times.

- **Player Actions:** Data related to specific player interactions, such as dice selections, re-rolls, and scoring decisions.
- **Customization Choices:** Information about selected customizations, such as dice face changes, passive modifiers, and consumables.
- **Connection Management:** Includes data for handling player connections and disconnections. This ensures smooth transitions when players join or leave a game.

11.4 Functions for Data Communication

The following functions from the 'networkManager2P' module facilitate communication:

- `broadcast_disconnect(state: String, data: Dictionary)`: Calls the rpc receive_disconnect on peers to notify.
- `receive_disconnect()`: The rpc function that can be calls to receive a disconnect notification.
- `broadcast_game_state(state: String, data: Dictionary)`: Calls the rpc receive_game_state on peer to notify each other of game their state.
- `receive_game_state(state: String, data: Dictionary)`: The rpc function that is called to receive the other peer's game state.
- `ping()`: Calls itself on all connected peers to track connection status.
- `send_game_settings(_game_settings: Dictionary,_hand_settings: Dictionary)`: Calls the rpc receive_game_settings on peer to send the game settings to the client device.
- `receive_game_settings(_game_settings: Dictionary, _hand_settings: Dictionary)`: The rpc function that recieves the game settings from the host device.

11.5 Design Considerations

The communication protocols are designed with the following considerations:

- **Reliability:** The ENet protocol ensures that all critical game state data is reliably delivered to connected peers.
- **Low Latency:** The use of UDP minimizes delays, ensuring real-time synchronization during gameplay.
- **Scalability:** While the current implementation supports two-player games, the architecture can be extended to handle larger multiplayer games if required.

This protocol design ensures a robust and flexible communication system for the Yahtzee game, providing a seamless multiplayer experience across both LAN and online environments.

12 Timeline

A Gantt chart was created to effectively assign tasks to team members. The chart can be accessed using the link below:

Access Gantt chart