Department of Computer Science and Engineering
Universidad de Puerto Rico
Recinto Universitario de Mayagüez

Academic Research Paper
Firebase + Flutter Game Development

Prepared by:
Alexander Angueira Bretón
Kelvin González Cortés
Ian Ortiz Cuevas

Report Date: November 27, 2024

Table of Contents

Abstract

The question that this research aimed to answer was if Firebase can be used as a viable means for online game development. There previously seemed to be no proper infrastructure to use Firebase for this purpose, and so the team aimed to create a Flutter package that would facilitate the development of online games. Firebase allows devices to listen to the changes within a collection, which can be used for game development by live transmitting game events to other players. Accomplishments of this research include the creation of a game taxonomy to better analyze game constraints, a Flutter package that facilitates game development using Firebase, and a collection of proof-of-concept games to demonstrate the capabilities of the developed package. Many questions needed to be solved, such as the handling of simultaneous events while maintaining data synchronization, the reduction of requests to Firebase, game state propagation throughout the app, etc. Although using Firebase for game development may have its limitations, such as the 200 millisecond event transmission delay and Firebase requests count limits, the resulting accomplishments from our research show that Firebase can be used as a viable means for online game development.

Introduction

Many modern games feature multiplayer functionality, and most traditional games, such as card and board games, are played with at least two or more players. The best way to connect players together in the modern world is through the internet with the emergence of online multiplayer games. There is an issue however, online multiplayer game development requires a server or a peer-to-peer system to be established, and such a task requires background knowledge, time investment, and possibly funding to be completed. A free and simple way to communicate online to other users is using Firebase's real-time database functionality, providing users with live updates. However, there doesn't appear to be any proper infrastructure for using Firebase as a means of online game development as of now.

This research aimed to complete the following objectives:

- Research how Firebase can be used as a means of communication with other players for game development. By listening to updates from Firebase, we can create a platform that allows users to play games with each other from a distance.
- Develop a Dart package that provides an infrastructure that makes use of Firebase's and Flutter's functionalities to create online multiplayer games.
- Create a game hub that serves as a demonstration of the capabilities of the developed infrastructure and also as a place for users to play with their friends.
- Publish the Dart package to https://pub.dev/, the official package repository for Dart and Flutter, so that developers can use our package, allowing them to create whatever games they desire to make.

This report delves more into the problem at hand and discusses the existing alternatives, mentions the tools that are provided and how they were used, states the accomplishments resulting from investigating this field, and draws final conclusions.

Problem Statement

Online multiplayer game development requires the host to provide a server or a peer-to-peer system that establishes connections to users, a task that needs background knowledge and time investment to set up and maintain to be performed, and funding if the developer does not wish to run a server locally. This proves to be a barrier for many developers that would like to develop simple games and prototypes of online multiplayer games, but do not meet the requirements to do so using existing tools.

There are many existing services that can provide some or all of the functionality necessary for online multiplayer game development:

- Azure App Services – This is a platform as a service that can be used to develop the foundation for a real-time database system. Azure App Services however does not provide this functionality out of the box and must be implemented by the developer. Due to the need for background knowledge and time investment for this solution, it was not considered.
- Amazon Simple Notification Service – This service provides a means for users to subscribe to a topic and receive notifications submitted to the topic. It can create a connection between multiple users easily and is free. The drawback of this service is that it does not persist the data sent to it, so users that connect to the topic are unable to read the notifications that had been sent prior to them joining. This is a feature that is necessary for game development since players should be allowed to know what has happened prior to them joining a game, therefore this alternative was discarded.
- DynamoDB Streams – This service provides a real-time database with the ability to subscribe to a stream in order to obtain live updates. While this is the exact functionality that is needed, DynamoDB Streams is not a free service, requiring funding that developers may not want to contribute.
- Supabase – Supabase is marketed as the open source alternative to Firebase. It is a service that provides a free-to-use, real-time, relational database. The data model for online game development can be seen as a collection of individual rooms with each one having its own messages list independent from any other rooms. While Supabase meets the requirements to be used for online game development, the data model for game rooms is better suited for a NoSQL database, since the only relation would be a one-to-many relation between rooms and messages. Another point to note is that, since it is a start-up, the long-term reliability and stability may be uncertain, potentially posing risks for applications that heavily depend on it.
- Firebase – Firebase provides a free-to-use, real-time, NoSQL database. Firebase is a service provided by Google and has been available publicly since 2012. Although Supabase may also be used for online game development, because of these reasons the team proceeded using Firebase instead.

Firebase already has some uses in game development; there even are development kits for C++ and Unity. The problem is that there seems to be no proper infrastructure for using Firebase as the central platform for executing online multiplayer games, where Firebase

effectively manages the entirety of the communications and state of the game. These existing kits are also built to be used with existing game engine software which can present a steep learning curve for developers that are more familiar with mobile or web development software.

Methods

- This research project used Firebase's functionalities to connect players to other players, synchronizing data on each of their devices so that they are all playing the same game at the same time. Firebase provides many individual features that have been used in this research project:
    - Documents – A document is a unit of data that represents a single item within a collection in Firebase. It is similar to a record in a database table. In this project, documents were used to represent both individual rooms and the aggregation of a subset of events within a room.
        - Format – The key-value nature of documents in Firebase allows us to aggregate "pseudo-documents" into a singular document where the key is the id and the value is the object itself, instead of storing them as their own individual document to reduce the number of Firebase reads. This technique was used for event transmission in games, where the event is stored inside of a "concatenated event" rather than as its own document.
        - References – Dart's Firebase framework allows users to save the reference for any document that they have created. We made use of this by keeping track of the room that the player is currently in and what is the current concatenated event used for event transmission.
    - Collections – A collection is a group of related documents, similar to a database table. Collections were used in this project to contain the list of rooms per game and the list of events per room.
    - Subscriptions – A Firebase subscription is when a device opts in to listen to live changes for either a document or a collection. We used this feature by making user's devices subscribe to the room's events collection to receive live updates for when new events have been added to the collection. When new events have been received, each user's device can process it and update their room and game state accordingly, ensuring synchronization between different devices.
    - Transactions – Transactions are used to ensure that the document-to-be-updated can only be modified by one device at a time, applying any other device's update request immediately once the in-progress modification has completed. This feature was used to preserve the event processing order between devices when multiple devices send an event at the same time.
    - FieldValue Feature – Dart's Firebase package provides a FieldValue feature that allows users to increment a number in a document, to update a document's list, and to store the timestamp that Firebase processes the update request. The number incrementer was used to keep track of the room's player count. The timestamp feature was used to store the time when an event was received by

Firebase so that players who just joined the game can read the full event log ordered by the event's timestamp.

- Either is a library for error handling and railway oriented programming. It gives functions the ability to return either an error or a value. This feature was used in our codebase by returning an error whenever a user's request fails a defined game condition, and returning a value otherwise.

Results

Game Classification

There are two ways that timekeeping is handled in games: real time and turn-based.

Real Time – The flow of the game progresses continuously, as opposed to taking turns. This means that all players are able to perform actions simultaneously, and all players are made aware of the actions' consequences simultaneously. Most sports are examples of real time games, as both players (or teams) must take actions and make decisions at the same time.

Turn-based – The flow of the game is broken down into defined parts, called "turns" or "moves", that are played sequentially. There are two main approaches in how these states are defined:

- Player-alternated or sequential – This is the most common type of turn-based game. This format follows players taking their turns one after the other. During a player's turn, only that player can choose an action. This category can be further subdivided depending on how a game handles the order of play.
    - Fixed order – In this category, the order of play remains the same throughout; after all players have taken their turn, the same order is repeated (e.g. in a four player game: P1→P2→P3→P4→P1→…). Most board games are examples of this; in chess, two players alternate moving one piece at a time
    - Variable order – In this category, the order of play changes in the middle of a game.
        - In-game actions can modify the order of play. In the card game Uno, playing certain cards will reverse the turn order, or skip the next player in the sequence.
        - The order of play can be determined by the current game state. In the sport of golf, the player whose ball is furthest from the hole goes next.
- Simultaneously-executed – In this type of turn-based game, players plan and make their choices at the same time without knowledge of the other players' choices. Then, their actions are revealed and processed at the same time. The game sequence can be modeled as choosing state → action state → choosing state → action state → …. Rock-Paper-Scissors is one such example, where two players make a choice at the same time, then reveal their answer simultaneously. Another example is the Pokémon video games: instead of two players taking turns attacking each other, both players make their move in secret, and the resulting actions are developed at the same time.

The range of games in existence is so broad that while these definitions cover most games, others might need additional elements defined. Some games even attempt to bridge the gap between real time and turn-based by incorporating elements of both into different parts of the game.

- One aspect of real-time games not usually present in turn-based games is the pressure of having a limited amount of time to think about your next action. Games can incorporate turn timers, where players are penalized for not completing their turn or turns within a time limit. For example, chess tournaments have stop clocks to avoid players taking an inconsiderate amount of time strategizing a move.

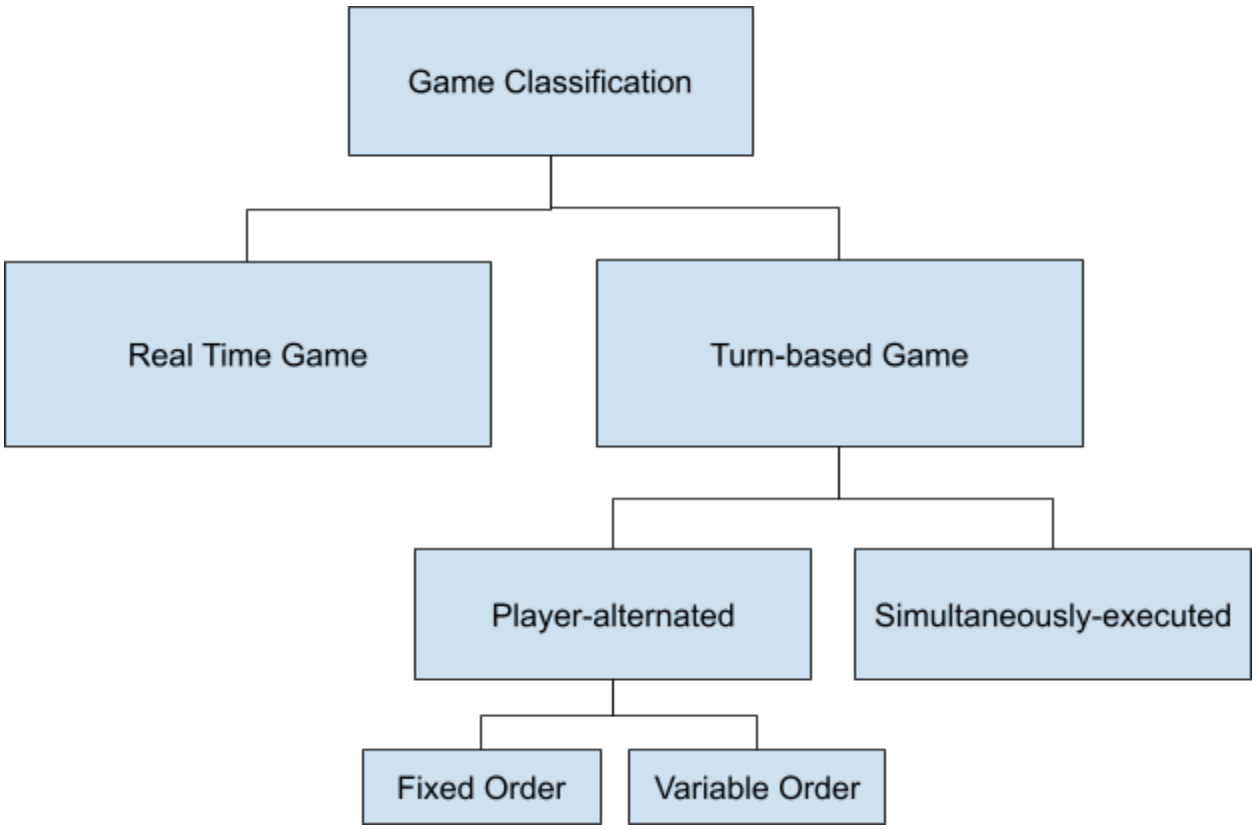

Figure 1. Game classification diagram

These categories cover most of the games that the members of the research group could think of, but we do not claim or aim for a coverage of all games. Some games may not always strictly fall under one single category, but instead are composed of multiple elements that each fall under a different category. The purpose of classifying timekeeping in games in this way is to explore the types of games that are of interest in this research and determine their compatibility with this package.

We had discussed ways to divide the real time category into smaller subcategories like we did with the turn-based category, but no satisfactory way to do so was found. Criteria like "fast-paced" is broad and open to interpretation, "frequency of actions" is too technical, and

"minimum reaction time required" depends heavily on when that action is required. The scope of real time games is a wide spectrum that cannot easily fit into categories with strict definitions. As stated before, the purpose of this classification is to better explain the games that this research will evaluate, and since no division would help us with that goal, we decided that having subcategories under real time was not necessary.

Design

Layers

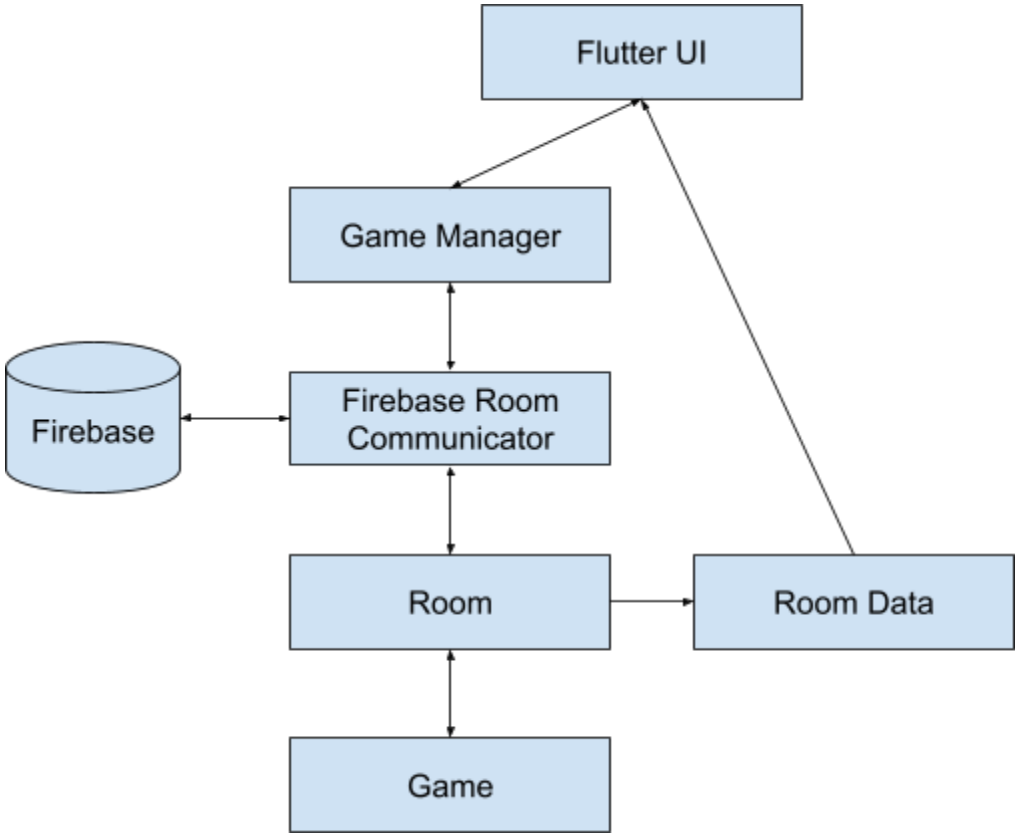

Figure 2. Structure layers diagram

This diagram shows the multiple layers involved in the package.

- The lowest layer is Game. This is used to create game specific logic/rules that define how the game will behave.
- Above Game is Room, which holds all of the game data such as who is playing, what events have been performed, who the host is, and how the game state currently looks like. It is doubly linked to Game since Room must execute the rules defined in game, and those Game rules, such as event processing, will modify the room's game state.
- Next to Room is Room Data, and they are part of the same layer level. Room Data represents all of the data held within Room. Its single use is to encapsulate data to be displayed by the visual element of the game.

- Between Room and Game Manager is Firebase Room Communicator. This layer is where all Firebase communications occur, updating your copy of the room. Events are transmitted and received from this layer.
- Above Firebase Room Communicator is Game Manager. This layer acts as the API for the Flutter UI, providing functions to send and receive data from the current room/game.
- The final layer is the Flutter UI, which is what the user interacts with. It is used to display Room Data to clients and allow them to interact with the game by sending events to the Game Manager.

Game Structure

A “game” is a collection of rules that must be followed in order for it to be played. The game rules give identity to a game; any deviation from these rules would result in a variation of the game or a new game entirely.

Game rules that must be defined include:

- Name – It is the common name for the game. It is what people use to refer to a game.
- Required Players – The count of players that are necessary to start the game.
- Player Limit – The maximum number of players that can be in the game.
- Initial Game State – This rule defines how the game state will look when the game is started.
    - In the package, the game state is modeled as an extensible class to allow developers to define as many fields as they want in the game state by extending it. The initial game state is defined by overriding the “getInitialGameState” function in Game.
- Check Perform Event – This rule defines what conditions must be met before the player can perform an action in the game. An example can be how in the game “Go-Fish” a player cannot ask for a card that does not exist.
    - In the package, this game rule returns a reason for why the event cannot be performed if failed. It is defined by overriding the “checkPerfomEvent” function in Game.
- Process Event – This rule defines what will happen whenever a player has performed an action. It is where the game state will be updated based on the player’s action. An example can be how in “Four in a Row”, whenever a player decides to add a piece to a column, the piece will be placed on top of the rest of the other pieces in the column.
    - In the package, it is defined by overriding the “processEvent” function in Game.
- On Player Leave – This rule defines what will happen to the game state whenever a player has left the room.
    - In the package, it is defined by overriding the “onPlayerLeave” function in Game.
- Check Game End – This rule defines the condition necessary for the game to be marked as over and stopped.
    - In the package, it is defined by overriding the “checkGameEnd” function in Game. The function must return either null if the game has not ended, or a JSON object as a log, which can contain who is the winner, if it's a draw, etc.

Room Structure

A “room” is where all of the content of a game’s session is accessed. It is where players and the current game’s state are kept track of. Actions that modify the game state are directly performed within the room itself. Players are free to join any room as long as they fit inside of it and the game session hasn’t started yet.

The properties of a room are:

- Host – The host is the player that created the game. They have full administrative permissions to modify the room, including deleting the room, starting the game, stopping the game, etc.
- Game – All rooms have a game that they are associated with, where the game defines the rules of play.
- Players – This is the list of players currently inside of the room. These are the people that are playing when the game session starts. Players can leave the room at any time.
- Game State – The game state starts off as the game’s initial game state, and is modified with time as players perform actions. In a game like checkers, the game state can be the board containing all of the pieces and whose turn it currently is.
- Event Trace – It is the ordered sequence of events that have been performed thus far. By reading the event trace, one would be able to recreate the current game state.

Firebase Structure

In this system, data is organized using multiple collections and documents, all starting with the “Room Collections”. Each game has its own collection of rooms, with each room represented as a unique document. The “Room**”** documents contain the following fields:

- host (map) – Contains details of whatever player is assigned as the host, including:
    - id (number) – A unique identifier for the host.
    - name (string) – The name of the host.
- playerCount (number) – The current number of players in the room.
- gameStarted (boolean) – Indicates whether the game has started.
- password (string) – The password for joining the room (null if not password protected).
- lastUpdateTimestamp (timestamp) – A timestamp from when the latest event was sent.
- Events Collection (subcollection) – A subcollection that records events occurring in the room in the form of “Concatenated Event” documents.

Each Room has its own Events Collection that stores all events within that room. The events are stored in documents known as “Concatenated Event” where each document field is an individual event labeled by the event’s id. Each event carries: who sent the event, a unique id, a payload, timestamp, and what type of event it is. The payload is saved as a map and its format can vary depending on the game and the type of event. It holds the actual data the player is trying to send, usually whatever action they are attempting to take. The collection is structured as follows:

- Event (map) – Each event represents an action or update in the game and includes:
    - id (number) – A unique identifier for the event.

- type (string) – The type of event (Player Join, Game Start, Game Event, Other, etc.).
- author (map) – The player responsible for the event, containing:
  - id (number) – The player's unique identifier.
  - name (string) – The player's name.
- payload (map) – The data associated with the event.
- timestamp (timestamp) – The exact time the event occurred.

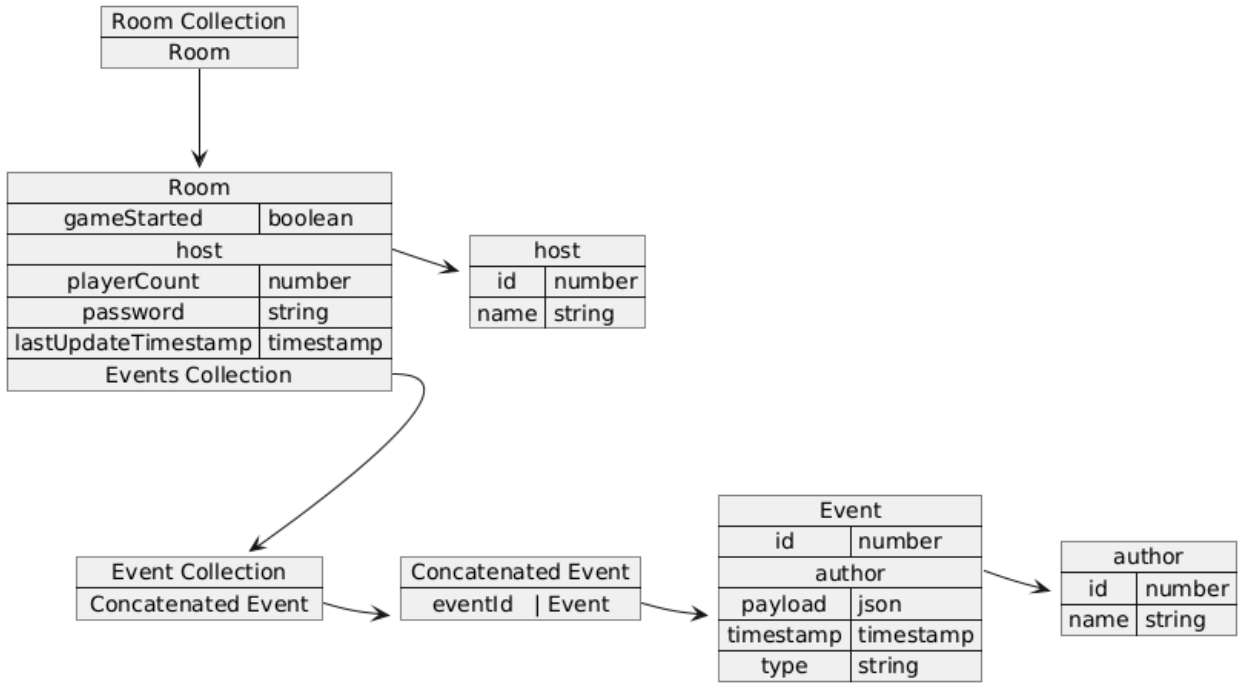

Figure 3. Firebase structure diagram

Game Manager API

The Game Manager is what the Flutter UI uses to send events and read updates to the room and game state. Provided are a variety of functionalities:

- Room Connection Handling – These are functions that control the device's connection to a room.
  - createRoom(Game, [String? password]) – Create a new room for the specified game. Apply a password if included.
  - joinRoom(FirebaseRoomData, [String? password]) – Join the room provided. The room data is obtained by using the RoomsBuilder widget or getRooms function.
  - leaveRoom() – Leave the currently assigned room.
- Firebase Event Transmission – These are functions that allow the device to transmit information over to Firebase and the other players.
  - sendGameEvent(JSON) – Send an event of type GameEvent to all players including the specified JSON as its payload.
  - startGame() – Commence the game for all players if the requisites are met.
  - stopGame() – Force stop the game for all players.

    - sendOtherEvent(JSON) – Send an event where the action is specified by a callback rather than the package. Completely user defined what this type of event will do.
- Data Streams – These are functions used to obtain streams that contain updated snapshots from the Game Manager.
    - roomDataStream getter – Return a stream that provides snapshots of the assigned room's data, including the players and game state.
    - getRooms(Game) – Return a stream that provides snapshots of all of the game's rooms.
- Offline Game Interactions – These are functions that query the game rules without the use of Firebase.
    - getGameResponse(JSON) – Returns either a value or failure based on the given request. It is defined with the game rules.
- Event Callbacks – These are functions used to set functions that get called based on specific event types.
    - setOnPlayerJoin(callback) – Set a function to be called whenever a player joins.
    - setOnPlayerLeave(callback) – Set a function to be called whenever a player leaves.
    - setOnLeave(callback) – Set a function to be called whenever you leave.
    - setOnGameEvent(callback) – Set a function to be called whenever a game event has been received and processed.
    - setOnGameEventFailure(callback) – Set a function to be called whenever a game event send request fails due to checkPerformEvent returning a CheckResultFailure.
    - setOnGameStart(callback) – Set a function to be called whenever the game is started.
    - setOnGameStop(callback) – Set a function to be called whenever the game is stopped. This function includes the log returned by checkGameEnd if the game was stopped via the specified game rules.
    - setOnHostReassigned(callback) – Set a function to be called whenever the host transfers ownership to a different player. This function includes the new and old hosts.
    - setOnOtherEvent(callback) – Set a function to be called whenever an event of type "other" is received.
    - setOnGameResponseFailure() – Set a function to be called whenever getGameResponse returns a CheckResultFailure.

Event Types

- Game Event – This event type is used to state that an action has been performed in the game and the game state must be modified. The actions performed by this event are specified by Game's "processEvent" function.
- Player Join – This event type is used to notify all players that a new player has joined and update their room instance to reflect that change.

- Player Leave – This event type is used to notify all players that a player has left the room and update their room.
- Game Start – This event type is used to notify all players that the game has begun and to start the game from their room.
- Game Stop – This event type is used to notify all players that the game has stopped. The game rules can cause the game to stop by returning a log via the “checkGameEnd” rule.
- Host Reassigned – This event type is used to notify all players that the host has changed.
- Other – This event type is used to send any other data not handled by the framework.

Firebase Room Communicator

The Firebase Room Communicator is in charge of dealing with all interactions between the device and Firebase, it updates the client’s copy of the room via the reception of events. The communicator also allows the ability to send events to all other devices connected to the same room.

The workflow of using the Firebase Room Communicator is as follows:

1. A new Firebase Room Communicator is created and assigned to your device whenever you link yourself to a game, whether it be via creating or joining a room.
    a. During creation, it sets up all necessary fields, such as the events stream subscription, which is in charge of reading new events from Firebase and processing them.
2. The Firebase Room Communicator is capable of sending any event type internally, though it exposes specialized functions such as sendGameEvent, joinRoom, leaveRoom, etc. that do specific logic to send events.
    a. Sending events is equivalent to adding the event to the events collection in the Firebase room representation.
3. When a new event is added to Firebase, it will send a snapshot including the new events to all players, which the Firebase Room Communicator will then process and update the device’s room data.
    a. Events are processed differently depending on their event type, such as a game event running the game rule for processing events, adding or removing players for player join and leave events, starting and stopping the game, etc.
4. After an event is processed, a callback function is called for that specific type of event. The Firebase Room Communicator exposes setter functions for assigning the callbacks.

State vs Events Transmission

The initial implementation prior to the start of this research course was to transmit the game state to Firebase and all players would receive it. This has two drawbacks:

1. Only one person is allowed to send their state to avoid conflicts, which means all games must be sequential, turn-based.
2. The full game state data must be transmitted, making games with lots of game data become unviable, or even reach the maximum Firebase document size limit.

For these reasons, an event based approach was implemented instead. Transmitting events instead of state allows multiple users to write simultaneously to the database while allowing the game state data to remain synchronized, which means that turn-based games are no longer a limiting factor for the types of games that can be made. It also allows all actions to be stored in their own document, reducing the amount of data that each transmission will take, while maintaining the same amount of reads to Firebase.

Event Concatenation

While an event based transmission system addresses many problems and allows the incorporation of real-time elements in games, it is not without its own flaws. When reading data in realtime from Firebase, a new document in a collection will be read only once, so a new event will be read once by each player when it is received. However, every 30 minutes Firebase will read all of the documents in a collection again, a very costly operation if the number of events in the collection is great. The solution to this problem was to store multiple events in a singular file rather than having individual files per event. With this system, instead of files being uploaded for every event, users directly update the existing file in Firebase, adding their event to it. It's possible to do this without worrying about multiple users modifying the same file simultaneously thanks to Firebase having measures to deal with the problem. This allows new users who joined the room to read the full history of events that have occurred so far, while also reducing the amount of documents stored for events in Firebase. We limit the amount of events stored on each file by creating a new one once the previous file has reached a specified number of events. This is done to prevent reaching the size limit for firebase.

Previously, events were stored in a list inside of the concatenated events document, however with the change introduced by the fix for simultaneous event ordering, that is no longer the case. Individual events are now stored as a map occupying fields inside of the concatenated events document, where each field is a key-value pair of the event id as the key and the event data in a map as the value.

Purely Event Based System

The first implementation of an event based system replaced the transmission of the game state in favor of the transmission of events that would modify a local copy of the game state. Since events were only used to modify the game state, a players collection was used to keep track of changes in the room's players, such as players joining and leaving. As the need to start and stop a game arose with the introduction of allowed player count ranges, more ways to communicate to other players were necessary. The solution that was chosen is to add an event type field to events and modularize events to be used for specific purposes, such as game events, players joining and/or leaving, the game starting and/or stopping, etc. Due to this change, the need to store a players collection in Firebase became obsolete, and now instead each device will hold a copy of a players list that gets updated on player events.

Host Approval System

When multiple players try to join a room at the same time, the players will not know that another player has joined and will add themselves to the room. This causes problems when, say a room currently has 3 out of 4 players, and 2 players join at the same time, resulting in 5/4 players. The proposed solution for this problem was to have a host approval system, where players would send a player join request event and wait for the host's response, it being either a player join event or a player join denied event. This system causes issues since it relies on another player approving events, causing delay, extra writes and reads in Firebase, and flakiness due to a host momentarily or permanently disconnecting. In the end, this implementation was scrapped in favor of having a system that does not query the host before joining a room. Instead, hosts are not allowed to start the game if it goes overcapacity, having to wait for another player to leave before starting the game.

Host Reassignment

The host is the player tasked with control over the room, who has the ability to start and stop a game session. This means that if the host leaves, no game can be played in that room anymore unless the host rejoins. The problem regarding the host leaving could be solved via two ways:

1. Deleting the room when the host leaves
2. Reassigning the host role to the next player in line

The chosen alternative was to reassign the host. Whenever the host leaves the room, a hostReassigned event is sent containing the player to be assigned as the room's host. All of the players in the room will receive this event and update their local copies of the room with this new data. This system ensures that even when the host leaves, players that remain in the room can continue playing the game.

Simultaneous Event Ordering

Whenever a document change is submitted to Firebase, the change is reflected first on the user's device and then to the other collection snapshot subscribers; the sender's device receives the change in around 7 milliseconds compared to 200 milliseconds for subscribers. This causes problems whenever two players send an event at the same time, resulting in that both players receive their own event first than the other player's. This caused games where players can act simultaneously to break, showing each player a game state where they each thought that they were the one to perform the action first. An example of this is a game show type game where players must be the first to click on the buzzer. If two players click the buzzer at similar times, they will both think that they have the right to answer the question, when only one of the players should be allowed to do so. The solution to this problem was to force all devices to process all events in the same order.

The first iteration of attempts to establish simultaneous event ordering made use of the "includeMetadataChanges" attribute when subscribing to a Firebase collection. This option allows the device to read a document again whenever it is done processing by reading the document metadata's "hasPendingWrites" field, allowing the device to only process events whenever the document has been fully processed by Firebase first. However, this solution is not

perfect, resulting in event processing discrepancies in certain edge cases, such as when Firebase sends a batch of simultaneous events to one user as individual events, but as an aggregation to another user. Since the timestamps stored inside of each event were the sender's local timestamp, then it does not necessarily match with Firebase, and each device may process the events in a different order.

This posed a problem though, since although Firebase allows users to access the timestamp when Firebase received the event via the use of "FieldValue.serverTimestamp", this value cannot be used within a list, and all individual events were stored inside of a list in one of the concatenated events. Due to this problem, it was not possible to save the server's timestamp directly into the events, only being storable within the document or concatenated event itself. An important detail to note for this is that the original event concatenation system has the limitation that whenever two players send an event at the same time and all of the existing concatenated events are full, then a new concatenated event will be created for each user. When such a case happens, the flow of execution is as such:

1. Two concatenated events get created, adding the event that each player wanted to send to each concatenation.
2. Afterwards, all players will make their current concatenation point to the first one shown in Firebase.
3. All events sent will be added to that concatenation until it is full.
4. Once full, all players will make their current concatenation point to the one that was previously additionally created. This concatenation will now contain the event that caused it to be created and all of the new events after the other concatenation was filled.

Because the second concatenation will contain a singular old event and the new events, if the document timestamp only reflects the changes for the new events, whenever a new player joins, they will process that old event as a new event, breaking the chain of execution of events, resulting in an incorrect game state. The fix for this problem would be to prevent multiple unfilled event concatenations from ever existing in the first place, or to consolidate the multiple unfilled event concatenations into one over time. As more design ideas for these fixes were developed, more problems arose, leading to the abandonment of this idea.

Through experimentation implementing various of the aforementioned designs, many new resources were learnt and used in the final solution:

1. While FieldValue.serverTimestamp cannot be used for objects inside of a list in Firebase, you are allowed to use it for objects inside of a *map*.
2. Firebase's transactions system forces users to wait for a document to finish being written to before they can write to it.

These two concepts are fundamental for the current design for dealing with simultaneous event ordering. How the current system works is as follows:

1. Events in an event concatenation will no longer be stored in a list to overcome the server timestamp issue, they are now stored directly inside of the concatenated event document as a field, with its value being represented by a map, and inside of each map is where the server timestamp is stored.

2. Sending events must now always be done via a transaction to ensure that multiple users cannot modify the same document at the same time, letting Firebase handle any conflicting writes.

This implementation resulted in a document reception system where all users receive updates at the same time and their game states will reflect the same values. The downside that this system incurs is that now the sender device must also wait the same amount of time as the receiving devices, roughly 200 milliseconds of delay.

Seed Sharing

Many developers may want to rely on randomized data for their games, however how does one ensure that all players generate the same random values on their local devices? Previously, there was no way to do this, if a developer wanted to rely on random data then they would need to generate the values and send them through an event to the other players. The proposed solution to this problem was to send a randomly generated seed whenever the game is started to all the players, populating a Random object on their devices that uses that seed. Whenever a developer wants any game rule to use random data, they use the room's Random object to generate a new value. The only game rules that are allowed access to the Random object are those that execute at the same time for all users, such as getInitialGamestate, processEvent, checkGameEnd, but not those that run once locally for each user, such as checkPerformEvent.

Backend Data Communication

Games may have scenarios where the front end needs to validate actions before sending an event in order to show the player unique information on their screen only. To avoid running game logic in the frontend itself, a new system was implemented that allows the UI to call functions from the game itself. GameManager now has a new function, getGameResponse, that allows developers to create functions that are run at the request of the UI. The UI can send a request to the game, giving any necessary parameters, and the game will return either a value set by the developer, or a CheckResultFailure. That data can then be used by the UI to display information unique to that player (since no event was sent to the other players). This includes the ability to show unique error messages in the case it returns a CheckResultFailure, such as informing the player if their attempted action is not possible. This feature also gives developers the option to validate events with game responses instead of checkPerformEvent; if the UI checks if any actions are valid before sending the event, then the UI has the option to always send a valid event.

Password Protected Rooms

Previously, there was no room access control, allowing any random player to join the room whether the host wants them to or not. To address this, the ability to assign a password to a room was implemented, allowing only those with the password to join the room. The password is stored as a new field within the Firebase room data. The room creation and joining functions now accept a new optional parameter for the password. If a password is provided when a room is created, that password will be stored in the room and only players who try to join using that

password will be granted access. If a password is not provided when creating a room, the room will be considered open, allowing any player to join.

GameState Class

Initially, the game state was modeled as a map so that the developer was able to add as many fields as they wanted, using the accessing operator to get and set field values. This approach has the following problems:

- Lacks types – When you access a value from an object of type Map<String, dynamic>, it is not possible to statically determine the type of the specified field, eliminating the benefits of using a typed language.
- Flutter Javascript conversion issues – When Dart code is compiled to Javascript, certain data types within a Map<String, dynamic> change, such as lists becoming JSArrays. This forces the developer to cast the field to List whenever they want to use the object as a list.

These two problems resulted in the creation of the GameState class.

The Flutter package defines GameState as an abstract class with an empty body, where developers can extend the class and add game specific fields to their GameState subclass.

Builder Widgets

Previously, games' UI code was created by using a StreamBuilder that listens to updates to the room data stream from the game manager. However, the introduction of the GameState class brought a new problem with this approach, making it so that you must cast the game state within the room data to the game's specific game state type in order to access game specific values. The team was opposed to forcing the developer to cast to a different type every time that they wanted to access game state fields, therefore a solution to this problem must be found. Taking inspiration from the BLoC package, which uses a BlocBuilder to display updated app state content and takes in a generic for specifying the state type, we created our own GameBuilder widget that will display updated room data using the specified game state generic type.

The GameBuilder simplifies the logic used to display updated room data, condensing it into a singular widget. The game builder works by using a StreamBuilder to listen to the given game manager's room data stream. If a game manager is not provided by the developer, it will use the device's static instance of GameManager instead. The game builder requires two functions to be supplied, the "game started builder" and the "not started builder". The game started builder determines what widget will be built whenever the room data shows that the game session has started. The not started builder will be used only when the game session has not started, and does not have access to the game state as there is no game state if there is no game session. The game started builder makes use of the game state generic type by casting the room data's game state to that generic type, allowing the builder function to have access to the specified generic's specific game state fields.

With the creation of the GameBuilder used to simplify the room data stream subscription process, a RoomsBuilder was also implemented which is used for listening to a game's rooms stream. Previously, to show the list of available rooms the developer had to manually listen to the Firebase room documents stream. This forces the developer to know the exact structure of the Firebase room documents in order to display specific data to the users. It is in our best interest for the developer to not be required to know how the data is stored within Firebase, so a FirebaseRoomData class was created. With this new data type, we could create a RoomsBuilder widget that would use a specific game and a builder function to display the list of rooms for that game. The builder function has access to the Firebase room documents as a List<FirebaseRoomData> for ease of accessing document fields. The RoomsBuilder operates similarly to the GameBuilder, where it runs the builder function every time that a new stream snapshot has been received using a StreamBuilder internally.

Removing Game from GameManager

Previously, GameManager featured a field used to specify what game room operations would be associated with, such as getting the stream of rooms, creating a room, and joining a room. Within the Room class, Game is used to know what game rules to apply whenever any events are received, making it necessary to store the game that is being played within the room. However, since GameManager only uses the game for room based functions, the game field could be completely removed, keeping it only within the room itself. This simplification improved the usage of the game manager, since you no longer have to set the game and then create or join a room. Now, you only need to add the game as a parameter to those functions.

Compatible Games

Most turn-based games should be compatible with this package. Games with game states where players make actions independently of one another, whether the choice they're making is simultaneous with other players (as with games such as rock-paper-scissors) or not (as with games like checkers), will be easily handled by the framework.

Real-time games are also possible to develop with this package, albeit with some limitations that are discussed in the following paragraphs. In general, the package is equipped to handle games where players perform simultaneous actions, whether conflicting or non-conflicting. There was an issue in the past that prevented the implementation of real-time games with conflicting actions, but the introduction of simultaneously ordered events has solved this problem.

The live transmission of information, which refers to cases where data is constantly being updated in real time, raises concerns pertaining to Firebase read and write limits. Suppose there is a game where two players can move on the same field at the same time: both players' positions should be updated in real time so that they can react to each other's actions. Such cases require frequent transmission of events; the more events are sent, the smoother and more accurate the experience is. While there aren't any problems sending and processing these

events, the problem comes from the cost of reading and writing to and from Firebase. The frequency of these events can cause Firebase to quickly reach the daily limits for these actions.

Because Firebase needs to process received events before sending them to all the other players, this introduces latency to all actions involving events. This can make games that require a player to immediately react to the actions of another player unsteady and difficult. For example, the game Pong could be modeled in our framework by sending an event every time the ball reaches the edge of the screen, calculating the angle of the bounce or if the ball scored a point. However, as this is a real time game, the game would need to wait for the opponent to send an event so that it can display the ball's bounce. Visually, this means that the ball would most likely stay frozen in place once it reaches the opponent's edge of the screen, and it will stay there until the device receives an event telling it what reaction occurred. Cases like these can make gameplay unappealing, as the delay makes the game go against expected behavior (in this example, the player expects the ball to bounce instantly from either paddle).

- The delay also introduces problems with the live transmission of events. If too many events are received, they will be queued up and slowly processed one by one at a fixed interval, unlike the different intervals events can be received in. In some games this is not a big problem. For example, in a drawing-guessing game where the drawing is transmitted to players in real time, the process of drawing can be modeled by sending each pencil stroke as a unique event. If a player draws many strokes in quick succession, Firebase will not process all the events as quickly as they were drawn, and thus the other players' screen will show the strokes in a slower succession. However, this will not severely impact gameplay as all non-drawing players are still receiving the same picture at the same time. That being said, there are cases where gameplay is greatly impacted, especially fast-paced games. Continuing the previous Pong example, another way to model the game would be to frequently transmit the position of each player's paddles, and updating the enemy paddle's position on a player's screen in real-time. However, the delay will cause the enemy paddle to not reflect the other player's quick, fine-tuned movement, which will cause desynchronization issues. Additionally, this will run into the problem of requiring a large amount of reads and writes, as previously discussed.

Collections in Firebase have a size limit of 1MB per document, so there is a limit to the amount of data that can be sent through an event. However, the size of an event is typically so small that in practice it would be very difficult for a game to ever reach such a limit.

Lastly, games are limited by Flutter's capabilities, as the games' visuals and interactive components must be built using the Flutter UI.

Applications of the Package

The purpose of this section is to highlight what this package allows the developer to do and showcase how we have used it thus far. This serves as a guide to show different ways to develop the game a developer may have in mind.

Tic Tac Toe

The structure of Tic Tac Toe is very simple, a playing board consisting of a 3x3 grid, player actions consisting of the specification of a grid position, a way to determine whose turn it currently is, a way of determining a winner or draw, and a way to display it all to the user. In terms of the developed package, Tic Tac Toe can be represented as such:

- Game State – The game state consists of a 3x3 grid and who the current player is.
    - When providing the initial state to all players, it will generate a game board filled with “-1”, representing no player has occupied the slot, and with the “currentPlayer” field as 0, representing the first player’s index.
        - This initial game state is provided by the game rule “getInitialGameState”
- Game Events – There is only one game event that can be performed in Tic Tac Toe, it being specifying what position in the grid the player would like to mark as their own.
    - Before sending a game event, the event must be validated using the “checkPerformEvent” game rule, returning whether it is valid or not. For this game, there are two possible types of check failures:
        - NotPlayerTurn – Represents when you are not the current player.
        - PositionAlreadyTaken – Represents when the position you are trying to occupy has already been occupied.
    - After validation, the event is sent to all players and processed. The processing of events in Tic Tac Toe consists of occupying the position specified in the event’s payload in the grid with their player number, and setting the current player to be the next player.
        - The processing of events is specified by the “processEvent” game rule.
- Game End – In Tic Tac Toe, the game ends whenever any player has marked 3 slots in a row in the game board. To determine this, an algorithm that determines the size of matching slots in any direction is run, declaring that the game has ended whenever it counts 3 slots in a row.
    - This is specified in the “checkGameEnd” rule, returning a log, in this case containing data pertaining to if the game was a draw or who the winner was.
- Flutter UI – The UI is what is presented to players and what they will interact with when playing the game. All interactions done here are through the Game Manager.
    - Displaying room updates – The package provides a GameBuilder widget that can automatically update the screen with any new updates to the room and game.
    - Performing moves – The Game Manager provides the “sendGameEvent” function that can be used to validate if the move can be performed and to send it to all users to be processed based on the specified game rules above.
    - In the UI, the 3x3 grid is represented as a grid of buttons, where each button will display who has occupied that grid position, and they will act as the medium to send game events, where tapping a button will try to occupy that position.

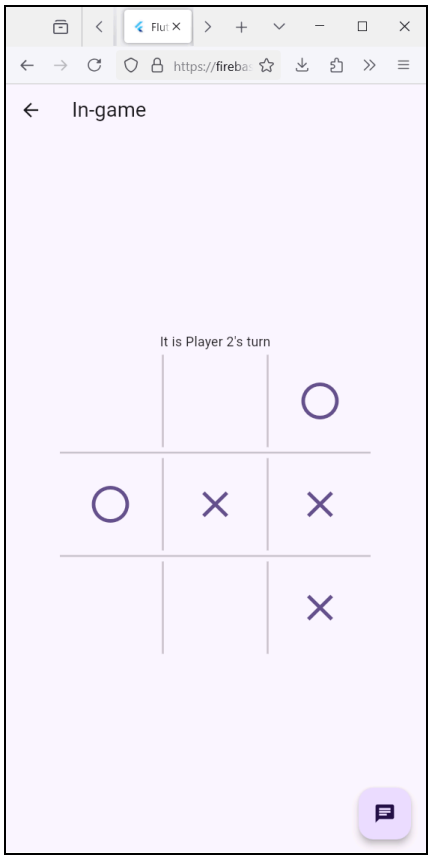

Figure 4. Tic Tac Toe Flutter UI

Four in a Row

The implementation of Four in a Row is very similar to Tic Tac Toe, the only changes being the size of the grid from 3x3 to 6x7, how an event is performed, needing to score 4 in a row instead of 3 in a row to win, and how the UI looks and functions.

- To perform an event in this game, the user will click one of the buttons at the end of each column, specifying that they would like to add a piece to that column.
    - The ColumnOverflow check result replaces the PositionAlreadyTaken of Tic Tac Toe since many pieces can be added to a column, but it cannot if it is filled.
- Instead of displaying a sectioned off grid like in Tic Tac Toe, only the columns are sectioned off. Each column has a button to interact with as opposed to each grid item being interactable like in Tic Tac Toe.

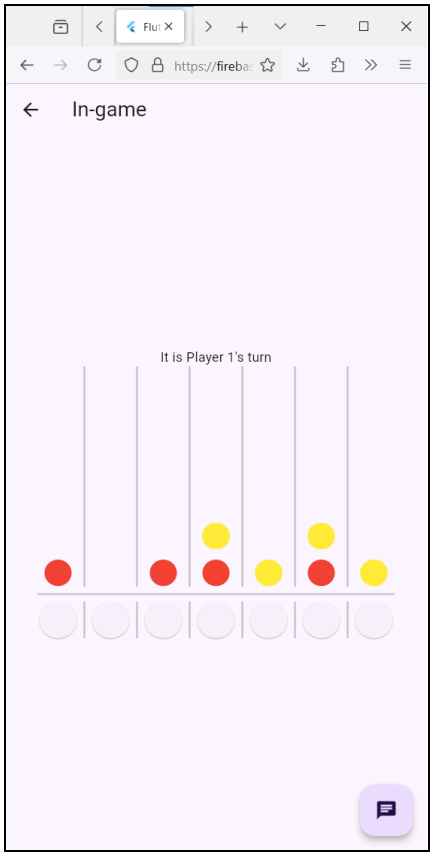

Figure 5. Four in a Row Flutter UI

Checkers

- Game State – Similar to Tic Tac Toe and Four in a Row, Checkers is also a player-alternated, turn-based game that is played on a game board. This means that the initial game state is very similar in structure to those games, only varying in some positions in the game board being occupied with game pieces.
    - However, what occupies the game board itself is more complex, storing objects representing Checkers pieces rather than a number representing a player's turn.
- Game Events – In Checkers, a game event will contain the route that the piece will travel throughout the board, removing any captured pieces along the way.
    - Event Validation – Unlike the previous games, most of the validation is performed by the Flutter UI calling the "getGameResponse" function instead of the "checkPerformEvent" function that happens when events are sent. This is because all the possible valid routes a piece can take must be calculated in order for the UI to show the available moves to the player. There are multiple types of request that the UI can make in order for it to display necessary information:
        - isCurrentPlayer – Checks if it's currently the user's turn
        - playerOwnsPiece – Checks if a specified piece belongs to the user
        - getPossibleRoutes – Gets all the valid moves of the specified piece. The game will check if the piece can move to or capture enemy pieces in

nearby tiles. In the case of a capture, it will recursively do the same check again on the new tile; it will continue doing this until the piece can no longer move. This function will return a list of all the valid routes the piece can take; routes contain a start and end position, as well as any intermediate steps if multiple captures occur in the same move.
        - getPieceColor – Gets the color of the specified piece

    Originally, all of these requests were implemented directly in the frontend. The problem with that approach is that having some game logic code be separate from the rest of the game's logic presents an organizational problem. Since many games have scenarios like this, where the validation of actions is done at the request of the UI before any event is sent, the need for dedicated support for doing these actions was discussed among research members and later implemented. Checkers is the first game to make use of getGameResponse.
    - Event Processing – The game rule for processing an event is defined as that the chosen route will be iterated through, removing all enemy pieces along the way, finally moving the piece from the start of the route to the end of the route. If the piece lands on the opposite side of the board where they started, they will be promoted to a king, allowing it to perform a greater variety of possible moves. After finishing this logic, the next player will become the current player.
- Game End – The game will end whenever any player no longer has any possible moves, meaning that either none of the player's pieces have any available moves or that all of the player's pieces have been captured
    - When the game ends, a log will be returned containing the winning player.
- Flutter UI – The game's UI consists of the game board, which is a 8x8 grid of tiles that can be interacted with, essentially acting as 64 buttons. A tile may display a checkers piece, a gray dot representing a possible move, or be empty. Like the other games, the UI listens to updates to the game state and updates the screen accordingly.
    - Selecting a piece – When a player taps a tile containing a piece of their color when it is their turn, it will "select" the piece. The game calculates all the possible moves of the selected piece, taking into account its position and the current state of the board. This is done to display gray dots in the tiles a piece can move to, so that the player can later tap one and complete the move. Tapping a different piece will deselect the previous piece and select the new one. Tapping a noninteractive tile (i.e. an empty tile or an opponent's piece) will always deselect a piece.
    - Performing moves – When a player taps a gray dot after selecting a piece, the game sends an event containing the route of the piece's movement.

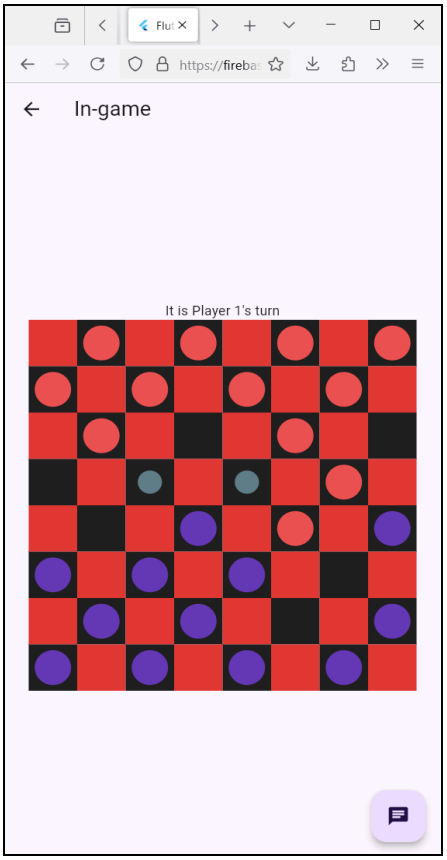

Figure 6. Checkers Flutter UI

All three of these games fall under the player-alternated, turn-based category. Due to the prevalence of these types of games, the package may include dedicated support for making games such as these in the future, allowing developers to more easily define them.

Rock Paper Scissors

Rock Paper Scissors is the first implemented game that does not fall under the player-alternated category for turn-based games and is instead simultaneously executed. This means that this game does not have the restriction that the previous ones had where a current player must be added to the game state, all players can send events regardless of any order sequence.

- Game State – For this game, the game state consists only of a list of choices that the players have selected, one slot in the list per player. An enum class was made to represent these choices.
- Game Events – In Rock Paper Scissors, an event is modeled as the selection of one of the 3 given choices.
    - Event Validation – Players are not allowed to perform an event if they have already made their selection. This is determined by if they have their dedicated slot filled in the game state's choices list.
    - Event Processing – When an event is received, the player's index is obtained and their choices slot is filled with the choice they made.

- Game End – Whenever all players have made a choice, it is checked if their choices are the same, and if not, which player's choice beats the other's. The created enum class contains a getter to obtain what choice it beats, and using that result is how it is determined whether one choice beats the other.
- Flutter UI – The UI for this game consists of a list of 3 buttons, one for each choice. When the player clicks on any button, and the other player has yet to make their choice, they will be presented with text saying that they are waiting for the other player.

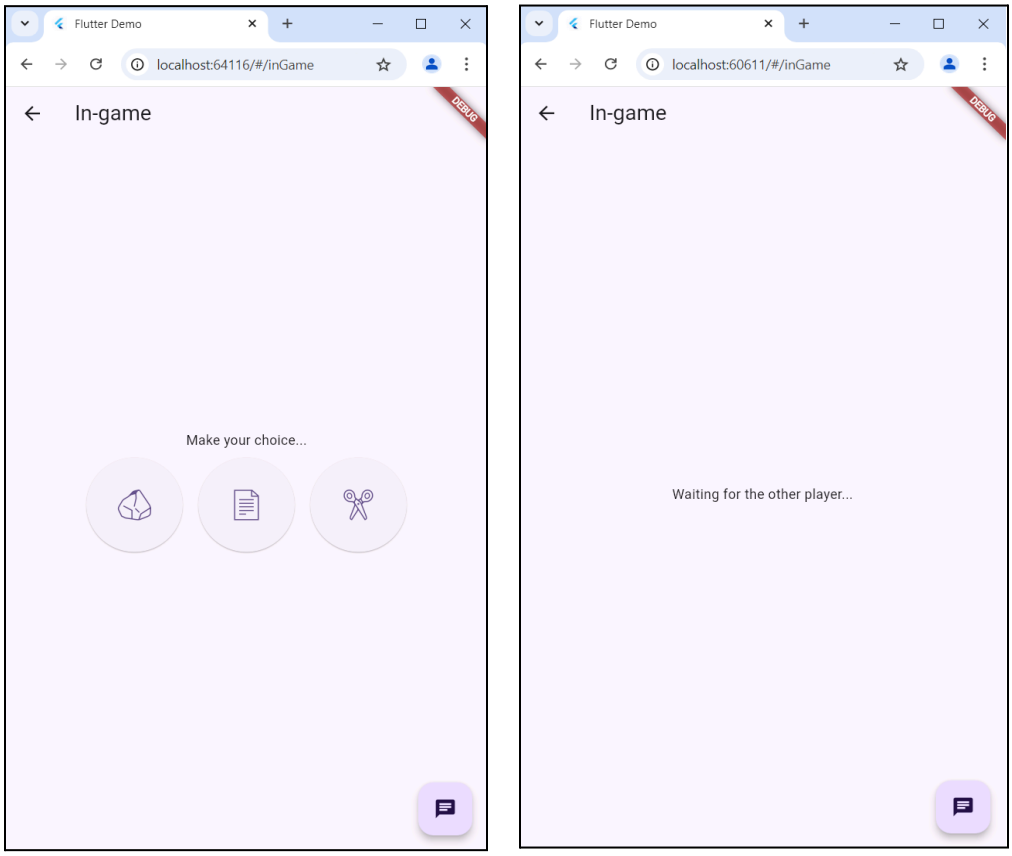

Figure 7. Rock Paper Scissors Flutter UI

Last Card

Last Card is a game where every player starts with a number of cards in their hand and they take turns placing cards into a pile until one player is left with an empty hand. Due to some cards having special properties, such as the "Skip" and "Reverse" cards, the order that players perform their moves is variable, making this game be the first implemented game that falls under the variable-order, player-alternated, turn-based category.

- Game State – The game state consists of a pile where players place cards onto, all of the player's cards, the current player, and the direction of player order. Cards are modeled as objects, where each card has a value and color.
- Game Events – This game has 2 types of events: "place" and "draw". The "place" event is used to place a specified card from the player's hand at the top of the pile, while the "draw" event is used to indicate that the player drew a card from the deck into their hand.

    - Event Validation
        - Place – To place a card, one of the following conditions must be met:
            - Color – The color of the card must match the color of the top card or must be colorless (Wild card).
            - Value – The value of the card must match the value of the top card.
        - Draw – Drawing a card is only possible if the player's hand does not contain a card that can be placed.
    - Event Processing
        - Place – When a card is placed, certain logic will be performed depending on the value of the card, whether it is a number, skip, reverse, +2, wild, or wild +4. When a wild card is selected to be placed, instead of placing the card directly, a new card will be created with the specified color. That card is then placed at the top of the pile, allowing cards with the same color to be placed on top of it. After the card is added to the pile, the card is removed from the player's hand, and the current player is updated.
        - Draw – When a card is drawn, the card is added to the player's hand. If that card can be placed on top of the pile, the current player remains the same. If not, then the current player is updated.
- Game End – The game will end whenever any player has placed all of their cards, resulting in an empty hand. That player is declared the winner of the game.
- Flutter UI – At the top of the screen, the list of other players is shown, where you can see each player's card count and the turn direction. At the center of the screen, the card at the top of the pile and the deck is shown. The deck is interactable and will send a "draw" event when clicked on. At the bottom, the list of your cards is shown as a horizontally scrollable list of custom card widgets that will send a "place" event when clicked on. The name of the player whose current turn it is is highlighted in bold and in a larger font size.

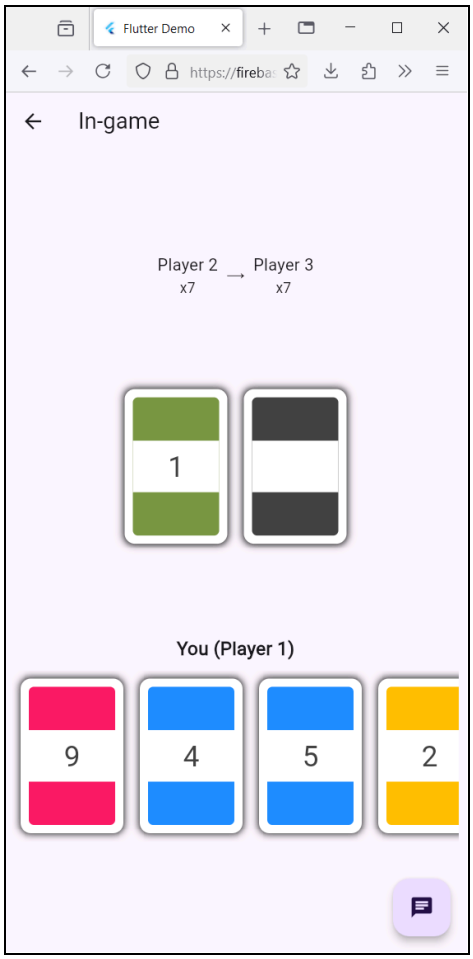

Figure 8. Last Card Flutter UI

Memory Match

Memory Match is a game where cards are placed face down and each player takes turns flipping two cards at a time to try and find the matching pairs. Every time a player finds a matching pair, the pair is removed. If the flipped cards do not match, they are flipped face down once again and the turn passes. The game finishes once all cards have been paired; the player with the most pairs wins. This is a player-alternated, turn-based game with a fixed order that won't change during gameplay.

- Game State – The game state consists of the current player taking their turn, the layout of the cards, and which cards were flipped during the current turn.
- Game Events – This game only has one type of event: sending the position of the card to be flipped. Once two cards have been flipped, the game automatically sends another event after a set delay to either flip back the card if they aren't a match or to remove them. The delay is added to give the players time to see the cards before they are flipped back or removed.
    - Event Validation – Players aren't allowed to flip a card if it isn't their turn, if they have already flipped two cards, or if the card they are trying to flip is already flipped or doesn't exist.

    - Event Processing – Once all the validation is complete, it processes the event by flipping the card in the desired position. When two cards have already been flipped, it checks if they match to know if to flip them back or if to remove them, and if it should pass the turn or not.
- Game End – The game will end once all the cards have been matched, with the winner being the player that paired the most cards.
- Flutter UI – The top of the screen shows whose turn it currently is. In the center, all the cards are laid out in a grid. Players can tap on whichever card they want to flip when it is their turn.

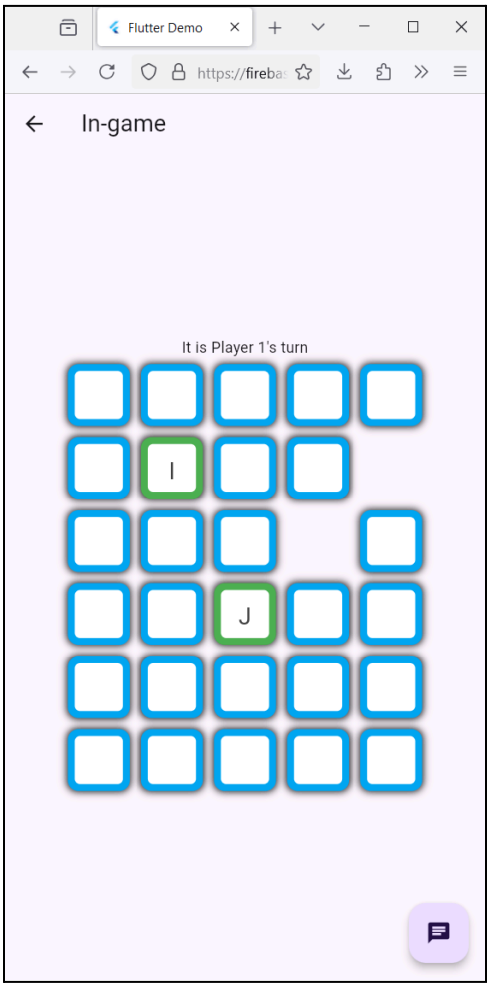

Figure 9. Memory Match Flutter UI

Endangered

Endangered is a “game show” style game where players select a question by category and difficulty from the board and can quickly buzz-in afterwards. Players take turns in ordering each question, the order being determined by the order of buzz-in time. This game shares elements of both turn-based and real-time games, where question selection and answering is done by

turns, however buzzing in is done in real time where the order of simultaneous events is important.

- Game State – The game state consists of many properties, some being global while others are based on the current play state.
    - Categories – The set of selected question categories for the game session.
    - Scores – The map of players and their scores.
    - Answered Questions – The set of questions that have already been answered.
    - State – Represents the current state of the game, determining what actions can be performed.
        - Selecting – This state allows a player to select the next question.
            - Current Selector – The player whose turn it is to select a question.
        - Buzzing – This state allows players to buzz in to determine the turn of answering.
            - Current Question – The question that was selected.
            - Buzzed In – The ordered list of players that have buzzed in.
        - Answering – This state allows players to take turns answering the selected question. This state also accesses the “Current Selector” and “Buzz In” properties.
            - Current Answerer – The index marking whose turn it is to answer. An index of the “Buzz In” list.
            - Current Answers – The set of answers that have already been submitted by other players for the current question.
- Game Event – This game has 3 types of events: “select question”, “buzz in”, and “answer question”. Select question specifies the question category and difficulty to select, and answer question specifies the selected index of the question’s answers list.
    - Event Validation
        - Select Question – The state must be “selecting”, you must be the current selector, and the question must not have already been answered.
        - Buzz In – The state must be “buzzing” and you must not have already buzzed in.
        - Answer Question – The state must be “answering”, you must be the current answerer, and the selected answer must not have already been picked.
    - Event Processing
        - Select Question – Saves the specified question as the current question and moves to the “buzz in” state.
        - Buzz In – Adds the event author to the “Buzzed In” list. If all players have buzzed in it will move to the “answering” state.
        - Answer Question – If the answerer responds correctly to the question, add the question’s difficulty to their score, make them the next question selector, mark the question as answered, and move to the “selecting” state. If the answer is incorrect, move to the next player in the buzzed in list. If the author is the last player, move to the next state similarly to when

answered correctly, however no players will gain any points and the current selector will be the first person that buzzed in.

- Game End – The game will end whenever all questions have been answered. The winner is determined by who has the greatest score.
- Flutter UI – The UI changes based on what the current state is:
    - Selecting – At the top every player's score is shown. In the center the name of the current selector is shown. Finally, the board showing all of the possible questions is shown. The board displays the categories as column names and the question difficulty changing per row. Each grid item is a button that will send a "select question" event specifying that question. A question will be marked whenever it has already been answered.
    - Buzzing – At the top it will show the question text. The buzzer button is displayed below it, sending a "buzz in" event when pressed.
    - Answering – At the top it will show who the current answerer is. The question text will not be displayed to incentivize players to read the question before buzzing in. The list of possible answers is displayed below, where each item will send a "select answer" event. Answers that have already been picked will be marked.

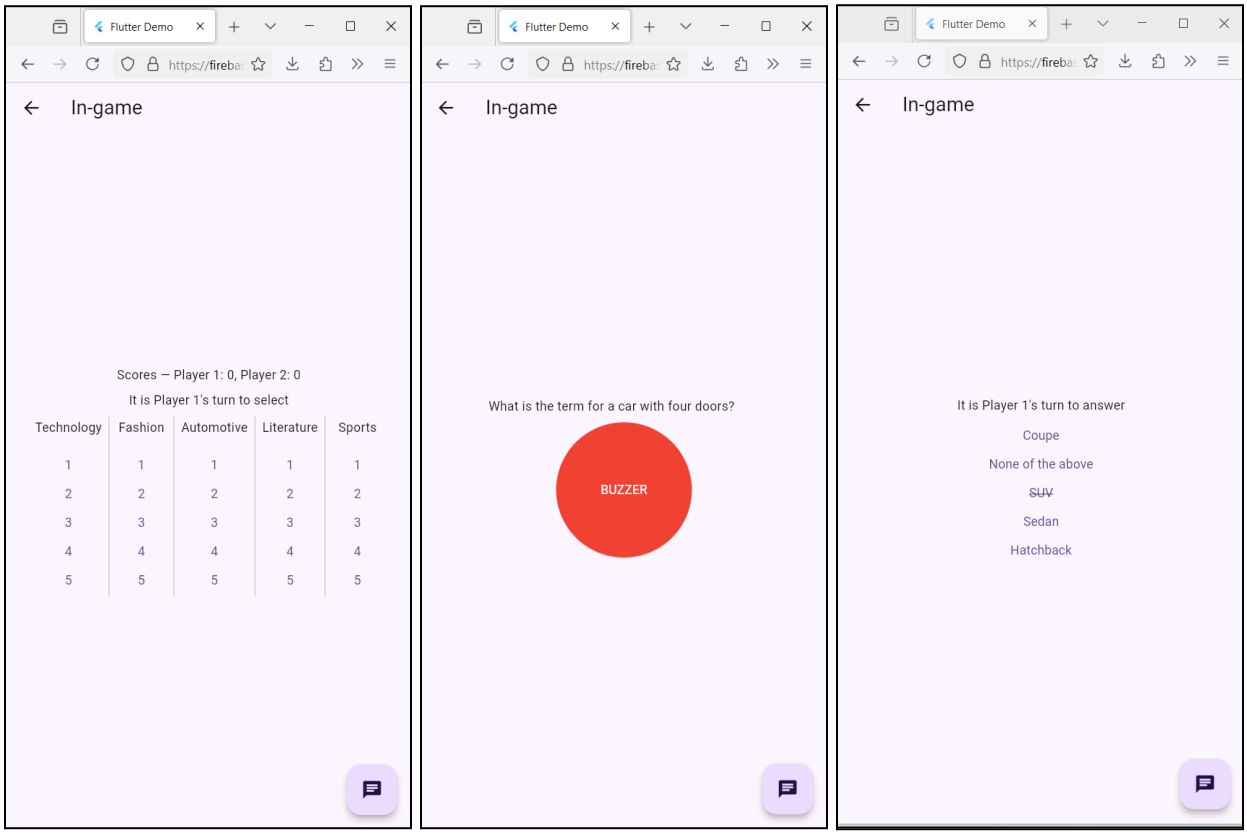

Figure 10. Endangered Flutter UI

Guess the Draw

In this game, players take turns for who has the ability to draw an image while the other players have to guess it. This game is the first implemented game where the gameplay is mostly real-time, where the drawer can draw at the same time that guessers can guess.

- Game State – The game state consists of many properties, some being global while others are based on the current round.
    - Global
        - Scores – Maps each player to their score.
        - Round Count – Contains the number of rounds that have been completed.
    - Round Based
        - Selecting Word – A boolean that indicates whether the drawer is currently selecting the word (true) or that they are currently drawing (false).
        - Current Drawer – The player whose turn it is to draw this round.
        - Word Options – The selection of words that the drawer can pick from when selecting a word.
        - Current Word – The word that was selected by the drawer, it is used when drawing has started.
        - Current Drawing – The list of strokes that represent the player's drawing.
        - Current Guesses – The complete list of guesses that any guesser has submitted.
        - Current Scorers – The ordered list of players who have correctly guessed the word.
- Game Events
    - Event Validation
        - Select Word – Event is only allowed when the state marks that a word is currently being selected and that the author is the drawer.
        - Draw, Undo, Turn Change – Events are only allowed when the state marks that a word has been selected and that the author is the drawer.
        - Guess – Event is only allowed when the word has been selected, the author is not the drawer, and they have not successfully answered the word already.
    - Event Processing
        - Select Word – Sets the current word to the one that was selected and marks in the game state that a word has been selected.
        - Draw – Add a stroke to the current drawing. A stroke is represented as a list of points.
        - Undo – Remove the last stroke from the current drawing.
        - Turn Change – Resets the round dependent variables and moves the current drawer to be the next player in the players list.
        - Guess – If the guessed word is correct, add 3 points to the author's score if they were the first to guess and 2 points to the drawer, or 1 point if someone else had already correctly guessed, and add them to the current scorers list. If the author guessed incorrectly, add their guess to the list of

current guesses. When all players have guessed correctly, trigger a turn change.

- Game End – Once all rounds have been completed, where the count is determined based on the players count, then whoever has the most points is declared the winner.
- Flutter UI – When selecting a word, the UI will show a list of 3 possible words to select from for the drawer, while showing the other players text telling them to wait for the drawer to select their word. When a word is selected, a timer will be initiated that will send a turn change event when 120 seconds have elapsed. Once drawing, both the drawer and guessers will be shown the rounds count, the time limit, the word to draw, the space to display the current drawing, and an area to display the current guesses. For the drawer, the current word is shown and they have the ability to add strokes to the drawing. For the guessers, the current word is omitted, only showing the outline of the word, and they can submit guesses from the current guesses area.

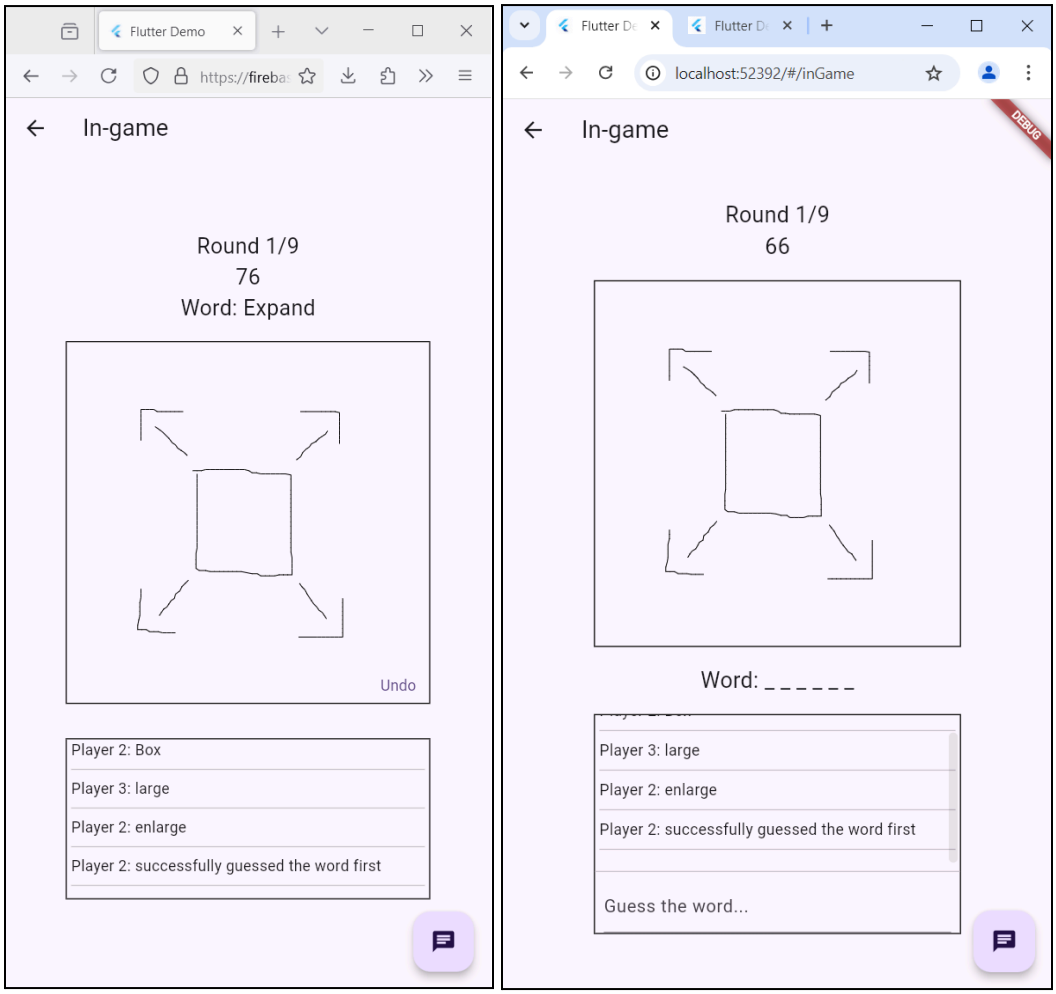

Figure 11. Guess the Draw Flutter UI

Ping Pong

This is the first game to use the supplementary game engine mentioned in the appendix section.

- Local (Game Engine)
    - Widget
        - Init – When the game widget is initialized, all entities are instantiated and the Firebase game event listener is assigned, where it will trigger the onHit and onMiss Puck functions for the “hit” and “miss” events correspondingly.
        - Update – Updates the puck’s room data field with the current room data.
        - Draw – Display each player’s score on the center of their side of the screen.
    - Objects
        - Puck – The main object within the game. The puck will bounce from side to side when colliding with a wall, wait for the opponent’s event to be received when it reaches the enemy barrier, send a “hit” event when it collides with the player paddle, and send a “miss” event when the player paddle is not hit. Hitting the player paddle will launch the puck upwards though with a horizontal direction based on the position on the paddle where the puck hit. When the paddle is missed, the puck is reset, making its position be the center of the screen and a downwards direction towards the player paddle.
            - onHit – Override the position and velocity of the puck with what the opponent has provided in their event.
            - onMiss – Reset the puck, making its position be the center of the screen and an upwards direction towards the enemy paddle.
        - Wall – The horizontal walls that the puck will bounce of from
        - Player Paddle – The device’s paddle. It will follow the pointer’s horizontal position when the screen is clicked on.
        - Enemy Paddle – A simulation paddle meant to represent your opponent’s paddle. It will always follow the puck since sending the opponent’s live position is slow and expensive.
        - Enemy Barrier – The area on the screen where the device will wait for the opponent’s response when the puck reaches it. It is placed at the top of the screen with the enemy paddle.
- Shared (Firebase)
    - Game State – The game state stores the score of both players, who made the last action, and the last recorded position and direction of the puck for each player.
    - Game Events
        - Event Validation – Event validation is not done through the package’s functionalities, due to it being handled by the game engine.
        - Event Processing
            - Hit – When the puck is hit, the position and direction are recorded for the event’s author, and they are marked as the player who made the last action.

        - Miss – When the puck is missed, add to the opposing player's score and mark the event author as the player who made the last action.
    - Game End – The game ends when either one of the players scores 7 points, making them the winner.

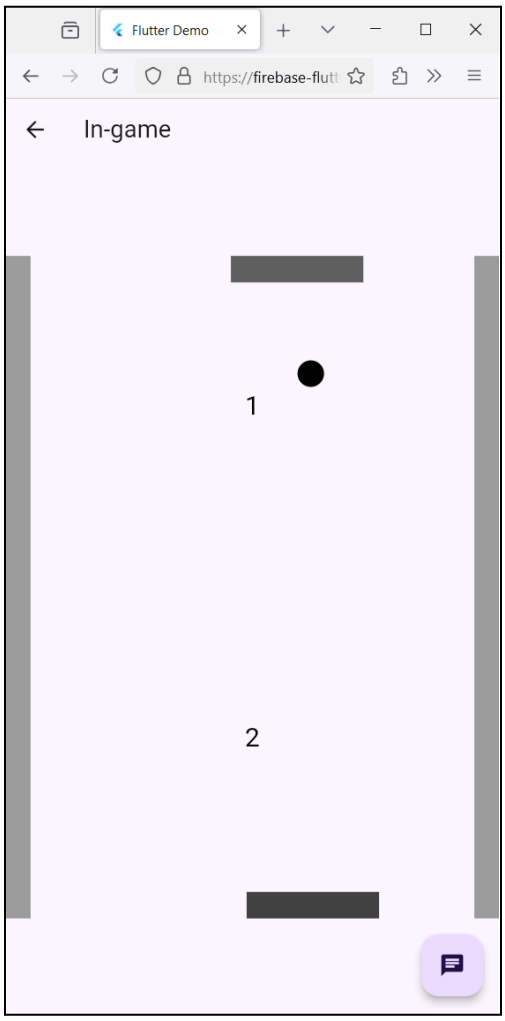

Figure 12. Ping Pong Flutter UI

Use of Callbacks

The framework provides developers the ability to specify callback functions to be called whenever certain events occur. Uses for these include pop ups, snackbars, etc.

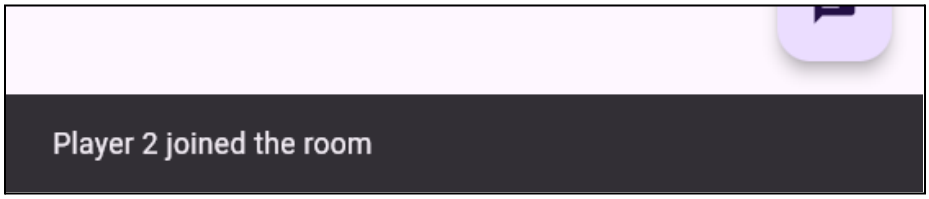

Figure 13. Snackbar displaying when a player has joined the room

Making use of the “other” event type, one can even create a chat messaging system for an in-game chat between players.

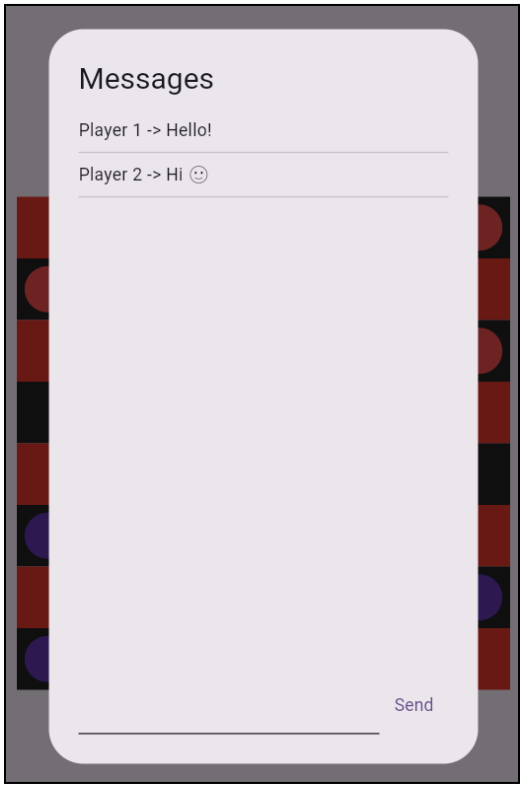

Figure 14. Chat interaction between players

Use of Password Protected Rooms

The game hub makes use of password protected rooms as follows:

- When creating a room, users are prompted to either enter a password to make the room password protected or leave the field blank to create an open room.
- When joining a password protected room, users are prompted to enter the correct password; only then will they be granted access. If the room does not have a password, users will not be prompted and can freely enter the room.

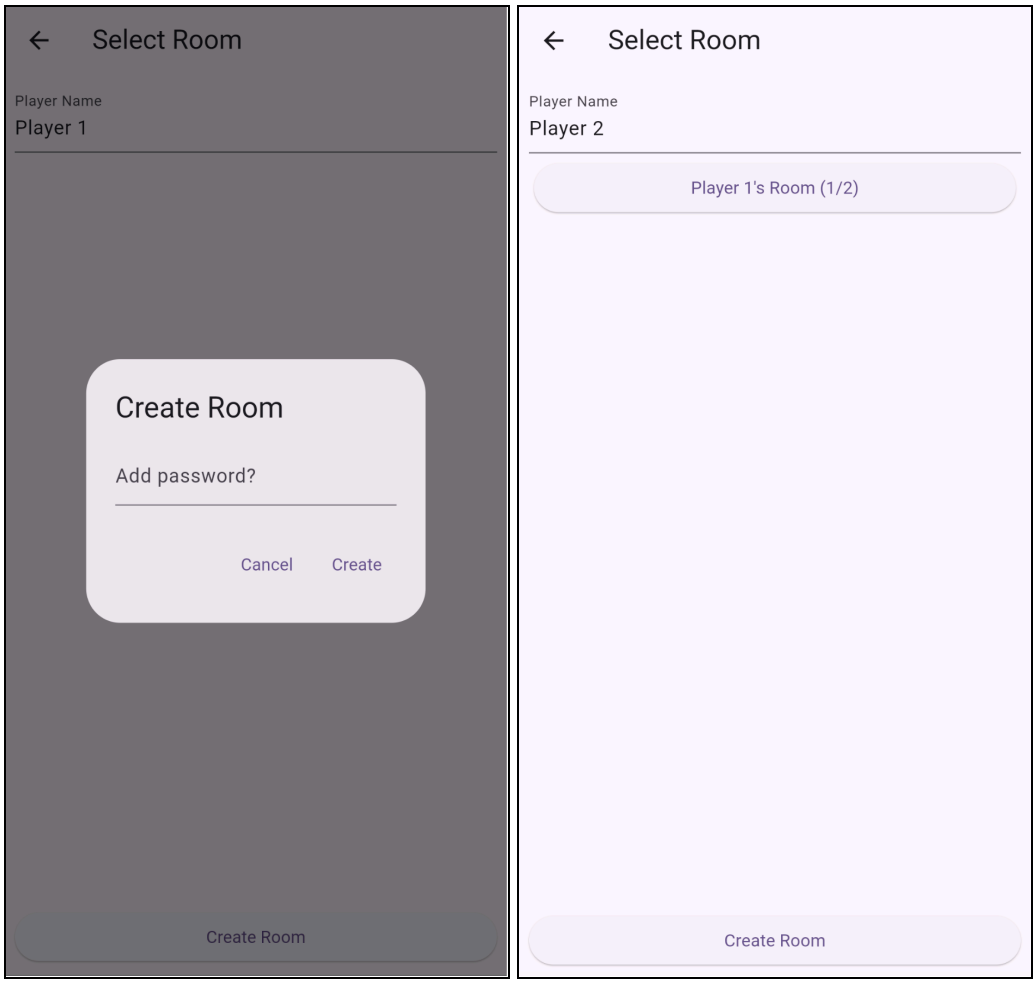

Figure 15. Creation of an open room

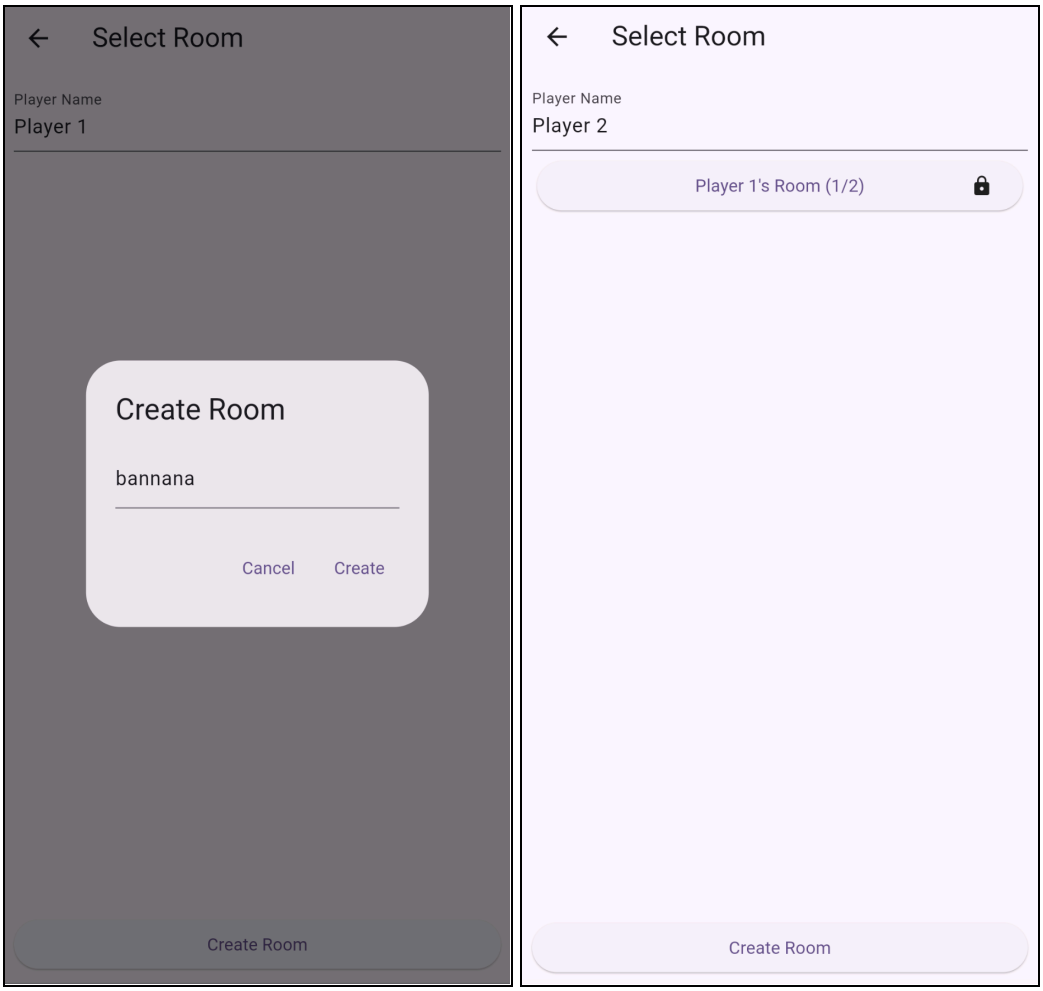

Figure 16. Creation of a password protected room

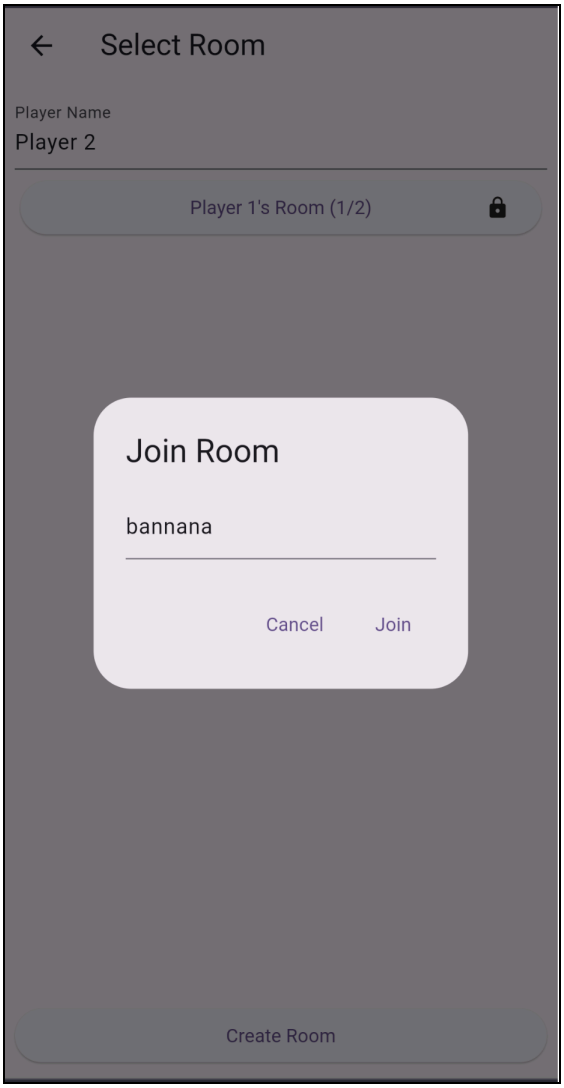

Figure 17. Password prompt when joining room

Performance

Firebase Communication Delay

Since events need to be transmitted by a player and then received by other players, event transmission delay is an important metric that must be measured as it can define what types of games cannot be made with our package.

Sent Epoch (s)	Received Epoch A (s)	Received Epoch B (s)	Received A - Sent (s)	Received B - Sent (s)
1725147308	1725147308	1725147308	0.011999846	0.162999868
1725147311	1725147311	1725147311	0.00999999	0.196000099

1725147312	1725147312	1725147313	0.002999783	0.292999983
1725147314	1725147314	1725147314	0.013000011	0.167999983
1725147315	1725147315	1725147315	0.003000021	0.173000097
1725147317	1725147317	1725147317	0.002000093	0.173000097
1725147318	1725147318	1725147318	0.002000093	0.171000004
1725147319	1725147319	1725147320	0.009000063	0.35800004
1725147320	1725147320	1725147320	0.002000093	0.15199995
1725147326	1725147326	1725147326	0.00999999	0.161000013
1725147327	1725147327	1725147327	0.009000063	0.169000149
1725147328	1725147328	1725147328	0.008999825	0.196999788
1725147329	1725147329	1725147329	0.007999897	0.162999868
1725147330	1725147330	1725147331	0.013000011	0.229000092
1725147331	1725147331	1725147331	0.004000187	0.164000034
1725147332	1725147332	1725147332	0.002000093	0.24000001

Table 1. Values for measured timestamps and comparisons (Before simultaneous events fix)

The table above shows the recorded timestamps and their comparisons, including the difference between the time elapsed when sending vs receiving an event on the same device, and the difference between the time elapsed when sending vs receiving an event on a *different* device *before* the implementation of the fix for simultaneous events processing. On average it took 6.938 milliseconds for a device to receive the event that they sent to Firebase, and it took 0.198 seconds on average for the device that did *not* send the event to receive it.

After the implementation of the simultaneous events processing fix, considerable delay was introduced to the communication between Firebase and the event sender's device.

Sent Time	Received Time A	Received Time B	Received - Sent	Received B - Received A
1728342848	1728342848	1728342848	0.203000069	0.198000193
1728342862	1728342863	1728342863	0.253999949	0.253999949

1728342864	1728342864	1728342864	0.22300005	0.210000038
1728342866	1728342866	1728342866	0.217999935	0.217000008
1728342868	1728342868	1728342868	0.203000069	0.203999996
1728342870	1728342870	1728342870	0.186000109	0.187000036
1728342872	1728342872	1728342872	0.197999954	0.197999954
1728342876	1728342876	1728342876	0.191999912	0.18599987
1728342877	1728342878	1728342878	0.18599987	0.184999943
1728342879	1728342879	1728342879	0.16899991	0.184000015
1728342879	1728342879	1728342879	0.194000006	0.187000036
1728342883	1728342883	1728342883	0.25	0.266000032
1728342884	1728342885	1728342885	0.35800004	0.210000038
1728342886	1728342886	1728342886	0.342000008	0.188999891
1728342887	1728342887	1728342887	0.17599988	0.197999954
1728342888	1728342888	1728342888	0.333999872	0.325999975
1728342889	1728342889	1728342889	0.170000076	0.177999973
1728342890	1728342890	1728342890	0.165999889	0.161999941

Table 2. Values for measured timestamps and comparisons (After simultaneous events fix)

The above table shows the same statistic as the previous table, with the difference being that these values were recorded after the application of the aforementioned changes. As shown by the timestamps, both the sender and receiver devices get an event update from Firebase after a similar delay, averaging 0.223 seconds for the sender device and 0.208 seconds for the receiver.

Firebase Read-Write Consumption

A large concern regarding the use of Firebase for a game development infrastructure is the associated daily read and write limits. The free-tier allows 50k reads and 20k writes per day.

Three different games were selected for testing the read and write consumption of the package: Tic Tac Toe for its game design simplicity, Last Card for its complexity and ability to host more than two players, and Ping Pong for its large frequency of game events. The compiled results

show that on average Tic Tac Toe consumes ~50 reads and ~25 writes per game session, which totals to ~800 games before the write limit is reached. Last Card consumes ~150 reads when two players are playing, ~300 reads when four players are playing, and ~75 writes per game session, totalling to ~250 games for two players before reaching the daily write limit and ~150 games for four players before reaching the daily read limit. Ping Pong consumes ~400 reads and ~200 writes per minute, totalling to ~100 minutes before the daily write limit is reached. In summary, games with greater amounts of events will reach a daily limit first, and the number of players will determine which of the limits is reached first, with a lower player count approaching the write limit quicker while a higher player count will reach the read limit first.

Publishing

Publishing to pub.dev was a very simple process. The only requirements to publish a package are to include a license, ensure the compressed package size is less than 100 MB and that all your dependencies are sourced from pub as well, and to own a Google account. With the use of "dart pub publish --dry-run" you can view what files will be uploaded and any warnings, and you can publish the package using "dart pub publish".

File Migration

Since our codebase was in one unified repository containing code for the package and the game hub, we first needed to split the codebase into two separate repositories. With the use of "git-filter-repo" we were able to migrate the package code to its own repository while maintaining the git history for our changes. In the end, the "package" repository was refactored into the "fire-game-infra" and "game-hub" repositories within our GitHub organization.

Community Feedback

After publishing the package, it was shared to some personally known developers and to the Reddit community in r/FlutterDev. As of November 27, 2024, no feedback has been provided to us from these sources.

Codebase

- The Github organization for this research can be found here.
- The codebase for the package can be found here.
- The published package can be found here.
- The website for the game hub can be found here.

Conclusions

In conclusion, Firebase was found to be a viable means for game development that is free-to-use and requires little experience to set up and use. The development of this Flutter package allows developers who have experience with Flutter to be able to create their own games with ease due to the layers of abstraction where the developers need only interact with the Game interface and Game Manager API.

To better understand what games could be created with the established package design and what areas we needed to explore more to overcome limitations, a taxonomy of types of games was created that highlighted the time-keeping of games, the concept of turn order, and the concept of simultaneous actions. The original package design was capable of handling only turn-based, sequential games. However through various design changes over the course of this research, the package is now capable of handling simultaneous, real-time games.

The team was able to create a vast array of games to showcase what the package can do, such as simple sequential turn-based games with small event and game state sizes like Tic Tac Toe and Four in a Row, games with a larger game state and more complex game rules like Checkers, games where player actions are simultaneously-executed like Rock Paper Scissors, games with randomized data needing to be synchronized like Last Card, games where the preservation of the order of simultaneous events is vital to the game such as the buzzer system in Endangered and the guessing of a drawing in Guess the Draw, and even games that deviate from Flutter's typical interface style like Ping Pong. The implementations of these games show how capable the developed package is and the great amount of uses aside from the aforementioned ones that it can have.

Performance testing showed that there is a considerable delay between the transmission of an event to when it is received, averaging around 200 milliseconds of delay. Testing also showed the rate of read and write consumption done by different games, where simpler games use up less requests and therefore more game sessions can be held before reaching the daily limit, while more complex games will reach the limit quicker. The number of players in-game is another factor to consider since less players means that the write limit will be reached first while more players means that the read limit will be reached first instead. These statements pose concerns for the types of games that can be developed using this package.

References

[1] "Usage and limits," *Firebase*. https://firebase.google.com/docs/firestore/quotas (accessed Aug. 31, 2024).
[2] Wikipedia Contributors, "Timekeeping in Games," *Wikipedia*, Jun. 13, 2024. https://en.wikipedia.org/wiki/Timekeeping_in_games (accessed Aug. 28, 2024).
[3] Matthias Häsel, "Architectures for Rich Internet Real-Time Games," *IGI Global eBooks*, pp. 226–231, Jan. 2009, doi: https://doi.org/10.4018/978-1-60566-026-4.ch039.
[4] Kuang-Hua Chang, "Sequential Game - an overview | ScienceDirect Topics," *www.sciencedirect.com*, 2015. https://www.sciencedirect.com/topics/engineering/sequential-game (accessed Sep. 16, 2024).
[5] Higher Rock Education and Learning, Inc., "Definition of Simultaneous Game | Higher Rock Education," *www.higherrockeducation.org*. https://www.higherrockeducation.org/glossary-of-terms/simultaneous-game (accessed Sep. 16, 2024).

[6] "Get started with game development using Firebase | Firebase Documentation," Firebase. https://firebase.google.com/docs/games/setup (accessed Sep. 21, 2024).
[7] "either_dart," Dart packages, Sep. 13, 2020. https://pub.dev/packages/either_dart (accessed Oct. 01, 2024).
[8] SyntaxC4, "Azure App Service documentation - Azure App Service," learn.microsoft.com. https://learn.microsoft.com/en-us/azure/app-service/ (accessed Oct. 24, 2024).
[9] AWS, "Amazon Simple Notification Service (SNS) | AWS," Amazon Web Services, Inc., 2019. https://aws.amazon.com/sns/ (accessed Oct. 24, 2024).
[10] "DynamoDB Streams Use Cases and Design Patterns," Amazon Web Services, Jul. 10, 2017. https://aws.amazon.com/blogs/database/dynamodb-streams-use-cases-and-design-patterns/ (accessed Oct. 24, 2024).
[11] Supabase, "The Open Source Firebase Alternative," Supabase. https://supabase.com/ (accessed Oct. 24, 2024).
[12] "flutter_bloc | Flutter Package," Dart packages. https://pub.dev/packages/flutter_bloc (accessed Nov. 03, 2024).
[13] "Publishing packages," dart.dev. https://dart.dev/tools/pub/publishing (accessed Nov. 19, 2024).
[14] Álvaro Manjarres Merino, "Firebase: An all-in-one platform for Web and Mobile application development," Sngular.com, Sep. 18, 2024. https://www.sngular.com/insights/313/firebase (accessed Nov. 24, 2024).

Appendix

Research Domain

Entities

- Game – It is defined by the game rules and played by the other players inside of the same room.
- Game Session – An instance of a game played by some players.
- Player – A person who plays a game.
- Host – The person who is the owner of the room where the game session is being held.
- Room – The space where other players meet to play a game.
- Game Rules – The definition of how a game is to be played, the requirements necessary to play, any win conditions, etc.
- Game State – The current state of the game session. It is modified by players' actions.
- Game Trace – The trace of all the game states within a room from start to finish. Every change in game state must obey the game rules of the room's game.

For any given room, its structure in the domain is:

- Room
    - Game

        - Game Rules
    - Host
    - Player-s
    - Game Session
        - Game State
        - Game Trace

Actions

- Hosting Game – When a person creates a room to play a specified game.
    - (Player, Game) → (Room, Host)
        - This signature states that a player wants to host a room of the specified game, resulting in a room being created with the player as the host.
- Joining Game – When a person joins another person's room to play together.
    - (Player, Game, Room) → (Room, Player-s)
        - This signature states that a player will try to enter the given room for a given game, adding themselves to the room, resulting in a room with an updated collection of players.
- Leaving Game – When a player forfeits from the game.
    - (Player, Room) → (Room, Player-s)
        - This signature states that a player will leave the given room, resulting in a room where the player is not contained within its players collection.
- Starting Game – When the host starts a game session when all players have joined.
    - (Host, Room) → (Room, Game Session, Game State)
        - This signature states that the host will start the session for the room's game, resulting in an updated room with a new game session with its initial game state.
- Stopping Game – When the host decides to stop the game session for everyone.
    - (Host, Room) → (Room, ~Game Session)
        - This signature states that the host will stop their room's game session, resulting in an updated room with its game session deleted.
- Ending Game – When a win condition has been met and the game session should end.
    - (Room, Game Rules, Game State) → (Room, ~Game Session)
        - This signature states that based on the room's game state and game rules, the current game session should be stopped, resulting in an updated room with its game session deleted.
- Performing Move – When a player does some action that impacts the game state.
    - (Player, Room, Game Rules, Game State) → (Room, Game Session, Game State, Game Trace)
        - This signature states that a player will perform an action in the room's game session, following the game's rules, resulting in an updated room with an updated game session. A new game state is created and added to the game session's game trace.

Behaviors

- Populating Game – The act of gathering a room full of players in order to start the game.
    - Action: A player decides to host a game in their room. (Hosting Game)
    - Event: A player has started hosting a game.
    - Action: A player joins the host's room. (Joining Game)
    - Event: A new player has joined the room.
    - …
    - Action: Another player joins the room. (Joining Game)
    - Event: A new player has joined the room. Minimum requirements to start a game session have been met.
- Playing Game – The act of starting a game session and playing it to completion.
    - Action: The host of the room begins the game. (Starting Game)
    - Event: The game has been started by the host.
    - Action: A player makes an action that impacts the game. (Performing Move)
    - Event: A move has been performed by a player.
    - Action: Another player makes an action. (Performing Move)
    - Event: A move has been performed by a player.
    - …
    - Action: The game falls in a state where a winner is decided. (Ending Game)
    - Event: A winner has been decided and the session has ended.
- Going Below Player Requirement –The act of a player leaving the room when the player count is at the minimum player requirement.
    - Action: A player leaves the room. (Leaving Room)
    - Event: A player has left the room.
    - Action: The session is stopped due to a lack of players (Stopping Game)
    - Event: The game has been stopped since player requirements are not met.

Research Members

Name	Roles
Alexander Angueira Bretón	Firebase data modeling and data cleanup
Kelvin González Cortés	Firebase transmission and reception, events processing
Ian Ortiz Cuevas	Games knowledge, package compatibility validation
Dr. Marko Schütz	Research professor and advisor

Table 3. Research members' names and roles

Game Engine

Flutter is designed in a way that encourages users to build their applications using a grid-like interface, where widgets either build on top of each other in the same cell, next to each other in a row, or over each other in a column. While many games can be represented using this

interface design such as Tic Tac Toe, it is much harder to implement other games such as Ping Pong where the puck's movement is not constrained to this grid-like positioning. Due to this conflict, it was decided that a game engine should be created alongside the main goal of this research project to meet these use cases.

The game engine consists of the following components:

- Game Loop – Games run at a tick rate of 60 ticks per second. On every tick, the "update" function is called, which can be defined to specify what should happen every time a tick has occurred, and is definable by the game widget and game objects. The update function also has access to the "deltaTime" property, which is a measure of the time between the current frame and the last frame in seconds, which can be used to correct any time dependent calculations when the rate is not exactly 60 ticks per second.
- Free Drawing – Flutter is typically locked to a grid-like interface system, however using the "CustomPaint" widget, one can overcome this by directly invoking canvas draw functions to insert any shape or image anywhere inside of the widget. The game engine makes use of the CustomCanvas widget to give developers the freedom of drawing anything anywhere on the screen, similar to popular existing game engines. The "draw" function, which can be defined by the game widget and game objects, is called every time the game widget is built. The function has access to the "Canvas" object that the painter uses so that you can directly invoke drawing commands from the game widget or objects.
- Game State Manager – The game state manager is the object used for obtaining any game state specific data or making actions that will alter the current game state. It provides the following functionalities:
    - Game Object Management – A Game Object represents a specific object type where instances of it can be added to your game. Game objects have their own init, update, and draw functions specific for that object type. Defining these functions allows you to define specific behaviors for each type within your game, such as walls used only for collisions or a player object that can be controlled. An instance can be added to the game by invoking the "addInstance" function from the Game State Manager. More instance management functions include instance removal, getting an instance from its id, and getting instances by their object type.
    - Pointer Listening – The game widget's CustomCanvas has a "Listener" widget as its parent, which means that it is capable of listening to pointer events, such as when the pointer has just clicked, when it has moved, and when it has released. The game state manager provides getters to obtain these values, where you can obtain if the pointer has just clicked the screen, if it is currently clicking the screen, if it has just been released from the screen, and its current position relative to the game widget.
    - Collision Detection – Each game object features a "HitBox" used for collision detection. Hit boxes have defined a function for when it intersects with another hit box at a given location and when it contains a specified point. Game objects can make use of these functions by providing a more abstracted version where you

need only specify if the object collides with another object and if a point is contained by the game object, using their own positions as reference. The game state manager provides functions for obtaining global object collisions, where they return a list of the game object instances that collide with the specified object or if they contain the specified point.

The codebase for the game engine can be found here.