Use Cases

# Searching for Friends

Sequence Diagram:



Currently it returns all currently **playing** users. They can play any of the optional games as long as they are not in the main menu.

# Creating New Game

Sequence Diagram:



All the new games are created and controlled on the server side. Offline mode is currently unavailable. The relevant parameters which are passed from client to server are **<GameType>** and **<DifficultyType>**.

# Joining Friend’s Game

Sequence Diagram:



All currently played games are stored within the server. Each client reports which session he belongs to, thus all the information that passed to the server by the requesting client, is which session he want to be part of.

The response message is **<MessageResponseSession>** which few other uses in the application. It will be explained later.

# Setting a Move

Sequence Diagram:



The set move use-case is completely asynchronous operation. The whole operation just updated the current game DB in the server, and during the polling technique which will be explained later, all the session members will receive the update.

# Sending a SMS

Sequence Diagram:



Sending a text message initiate a process which pushes this text message to all participants in the session (in the game). The server receives the message, and iterates over all relevant clients, and push the message to them.

Design Principles

# Client/Server

As it seen in the class diagram, system architecture is standard client/server, with centralized server control. Means:

* No option for offline playing.
* There is no direct communication between clients.
* All operations, starting game, joining, and operating in games is through the server.
* Server hold centralized DB with the current state in each and every game.

Client and server communicate using Google Endpoint API (as explained later), and simple messaging system (as explained later).

# Google Endpoint API

The **<MyEndpoint>** class in backend defines the contract of the API.

It defines single operation **<sendMessage>** and receives an object of the defined class **<Packet>** (which gains it’s replica at the client).

A **<Packet>** is actually a stream of bytes representing a compressed message, and a timestamp when it was sent.

Full API signature:

***Pakcet[] sendMessage(Pakcet packet);***

Packets are returned only when the message is polling message.

# Messaging

All communication in the system is working with messages. We have two ways of message communication in the system:

1. **Inner messaging communication –** Each activity defines inner class which implements **<ICallback>** with **<recieveMessage>** method, and passes instance of that class to other components (like network component/service component). Messages received can be extern (originated from the server), or intern (like connectivity messages).
2. **Client/Server messaging communication –** Sending messages to the server using the Client/Server API (Google Endpoints API). We pass single message to the networking component, it compress it, and send to the server.

We can divide messaging feature into multiple components:

## IMessage class

Interface with three methods:

1. ***MessageType getMessageType()* –** returns enum of the concrete message class. For each concrete message class, there is an entry in the enum. Using this method, one can cast the interface to the appropriate class without knowing its dynamic type.
2. ***Client getClient()*** – returns the end-user Client who relevant to this message. If it’s request message, then this is the sending client, if its response message, then it is the destination client.
3. ***UUID getId() –*** returns the message ID, or the ID of the request it responds to.

Each of the methods returns the appropriate variable that must be defined in concrete message classes.

All the message communication (inner client, and client-server) is working with the **<IMessage>** interface (to keep the networking interface generic and scalable), and the receiving component takes care according to the message type received.

## Concrete Messages

Each concrete message definition requires:

1. Be defined serializable.
2. Define public member client.
3. Define public member id of the message.
4. Return appropriate value in the **<getMessageType>** method.

We can divide concrete messages into three groups:

### Request Messages (prefix **<MessageRequest\***>)

All the messages created by the endpoint client, and meant to be send to the server.

Client variable receives the sending client.

ID receives an unique identifier.

### Response Messages (prefix **<MessageResponse\*>**)

All the messages returned from the server, and raised to the event handler at the proper activity.

Client variable received the destination client.

ID receives a unique identifier, or the request ID if it’s a response.

### Inner Messages (prefix **<MessageInner\*>**)

All the messages that are not being passed in network. Usually raised from the application modules to the variant activities.

## MessageCompression class

Singleton class with two methods:

* Byte[] compress(IMessage message);
* IMessage decompress(byte[] stream);

The compression/decompression using standard java serialization using ObjectOutputStream/ObjectInputStream, thus for the server is required to hold the exact same classes in same packages to decompress it correctly.

# Session

A session describes a single multiplayer game and all the information attached to it. Session properties:

* ***UUID \_sessionId*** – unique identifier for the session. Controlled by the server. It’s the only information endpoint client have describing their session.
* ***LinkedList<Client> \_clientList*** – list of clients currently playing.
* ***GameType \_gameType*** – the game being played in the session.
* ***SudokuGameData \_gameData*** – the game information regarding this session. This game data object contains the most updated game board, and updating rest of the players in the game about this board. (TODO: soon to be generic GameData)

Endpoint clients gain update about the session with the appropriate **<MessageResponseSession**> message, and after receiving this message and validating its credibility, they update their gaming board and the participating clients in the session.

Occasion when **<MessageResponseSession>** is being sent:

* Every polling interval. After each polling request, this message is being returned (explained later).
* When creating a new game – after sending **<MessageRequestNewGame>** to the server, the server returns session message with the newly created game.
* When joining existing game (friend’s game) – after sending **<MessageRequestJoin>** to the server, the server returns session message with the newly joined game.

# Polling Technique

System is implementing Client/Server architecture with one-sided communication. Means, the server can’t push notification to the client, but the client has to poll for new information from the server.

Polling steps:

1. Every **<POLL\_TIMESTAMP>** seconds (defined in settings class) background service (explained later) send synchronous special poll message (request type) to the server.
2. The synchronous call returns list of packets. These packets are all the unread messages for the specific client.
3. The client raises all the messages up in the chain (to the activity) for handling.

Clients need to have the most updated board all the time, that’s why each time client polling for update, the server returns him **<MessageResponseSession>** message in addition to the packets waiting for him. With this session message the client updates the current connected peers in the game, and updates board if there were any changes.

# Service

Safety Measurements

# Server Architecture

Sgads

# Message ID Validity

Each request message is sent with unique identifier created by the client. All the request message are asynchronous, thus response messages need to be validated using that id. The validation goes as follows:

Searching for friends

**<MessageResponseClientList>** Id will be same as **<MessageRequestAvailableClients>** Id.

Creating new game

**<MessageResponseSession>** Id will be same as **<MessageRequestNewGame>** Id.

Joining friend’s game

**<MessageResponseSession>** Id will be same as **<MessageRequestJoin>** Id.

Polling messages

**<MessageResponseSession>** Id will be same as **<MessageRequestPollMessageQueue>**