ColourMe

Distributed Real-time Multiplayer Game

Developed by

Ahad Choudhary (301287058)

Arsalan Macknojia (301294147)

Asim Himani (301276136)

Osama Elsamny (301291447)

CONTENTS

1. Introduction

ColorMe is a distributed, real-time multiplayer game. It is a client-server application that supports synchronization, concurrency and fault-tolerance by replication. The following report provides an overview of the application components, data-flow between components, user interaction, distributed characteristics, design decisions implemented, and the trade-offs considered during the development of the game.

The report has been divided into four sections: Design, Architectural Choices & Trade-offs, Characteristics and Conclusion.

2. Design

2.1 Architectural Diagrams

The following diagrams provide a detailed layout of the game architecture.

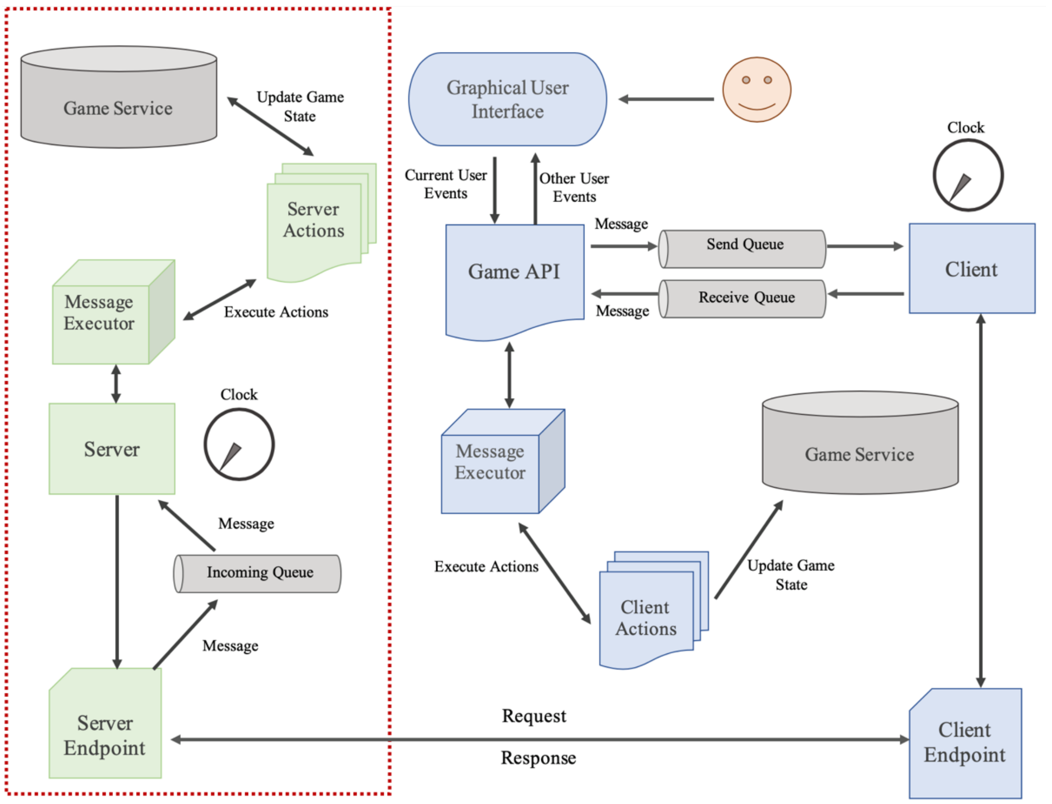


Figure 1. Game Architecture

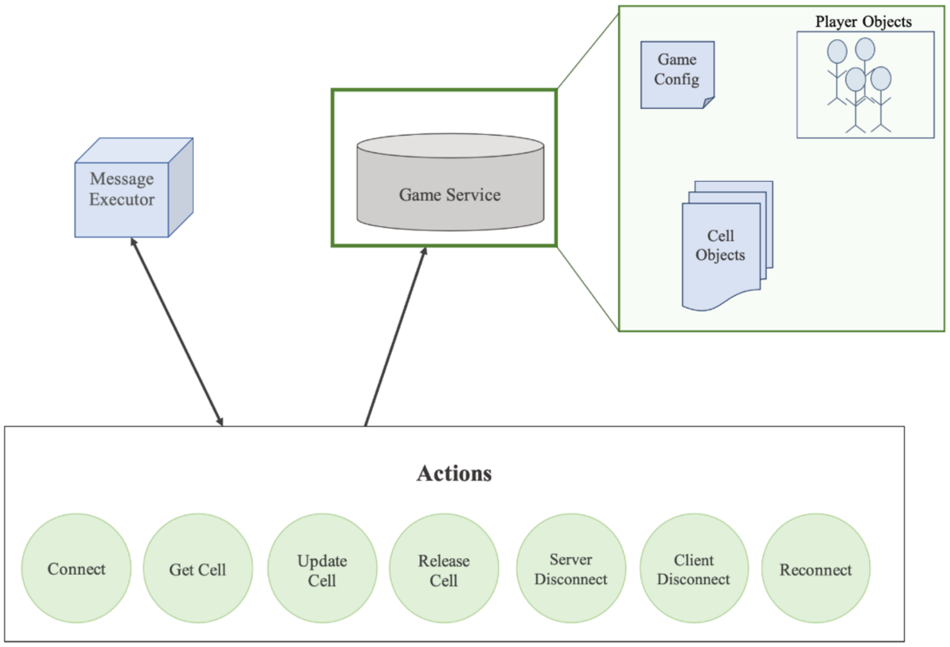


Figure 2. Game State Architecture

2.2 Components

The game architectural diagram above includes numerous components. Below is a glossary of individual components and the function they perform.

Client: Provides a networking interface for sending requests to and receiving responses from the server.

Server: Provides an interface for receiving game requests, and broadcasting game state updates to the connected clients.

Actions: Implements the core game logic to perform game operations and is separated into multiple action entities for modularity.

Messages: Defines a common data format for messages that flow through the application. It exposes an API that allows invoking relevant actions, based on a given message type.

Marshalling: Provides data marshalling and unmarshalling for client requests and server responses.

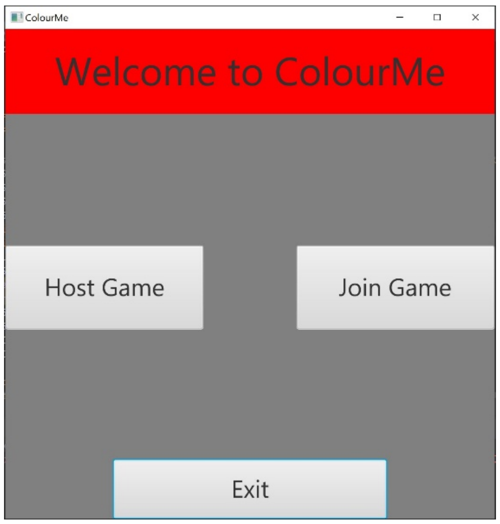
Game State: Comprises of the data storing the state of the game at any point in time. This includes player objects, game configuration, cell states on the game board and so on.

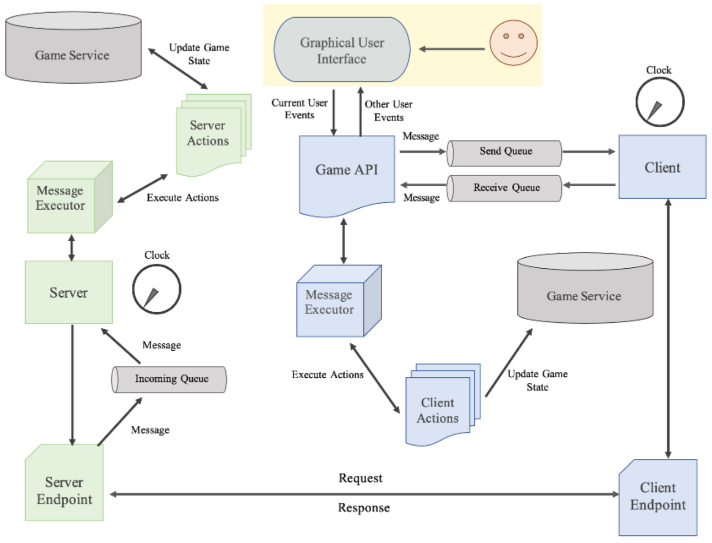
GUI: Displays the game to the user, invokes user events and re-renders the game with new updates at regular intervals. The GUI communicates with the Client through the Game API.

Utilities: Consists of interfaces for logging, json parsing and exception handling libraries.

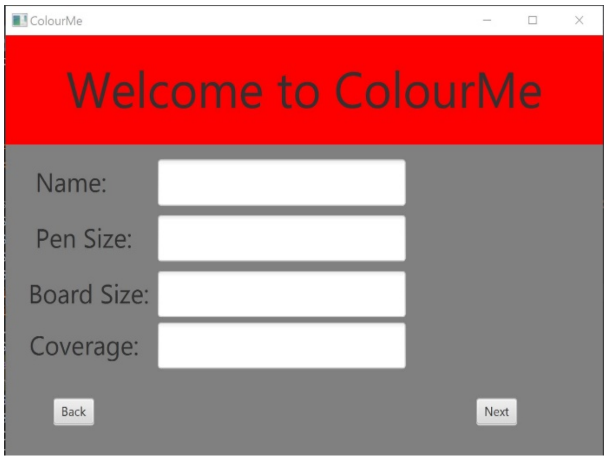
2.3 Dataflow and User Interaction

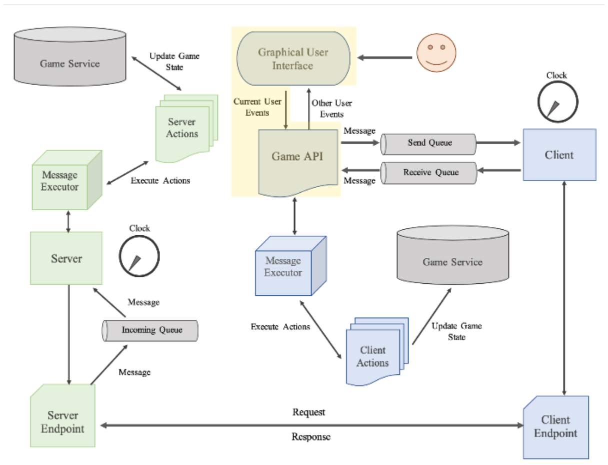
User inputs dictate the game state through GUI.



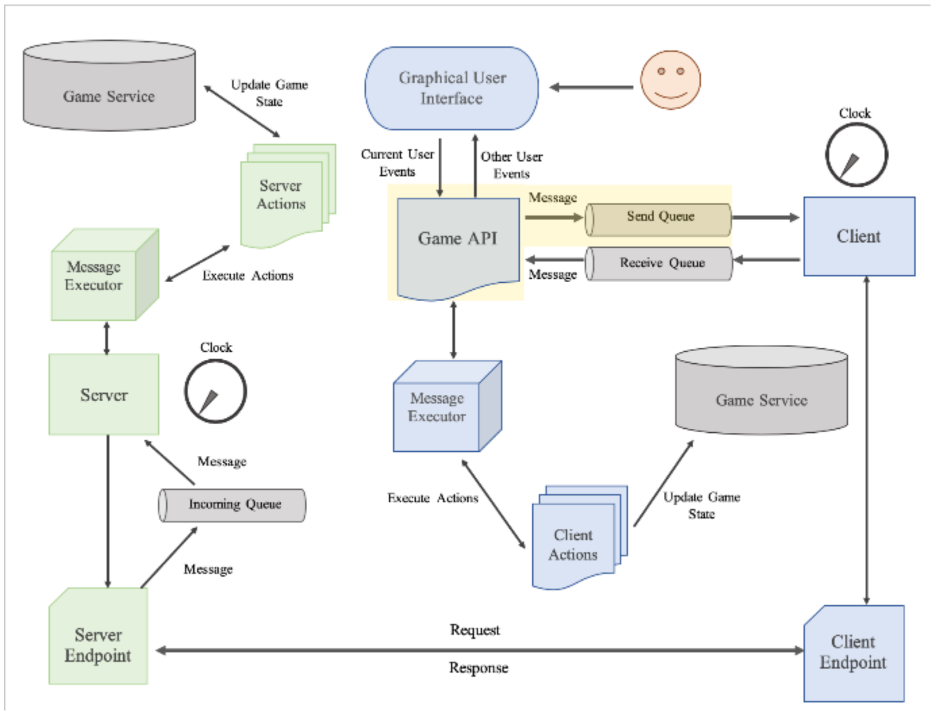


GUI sends game events to GameAPI.

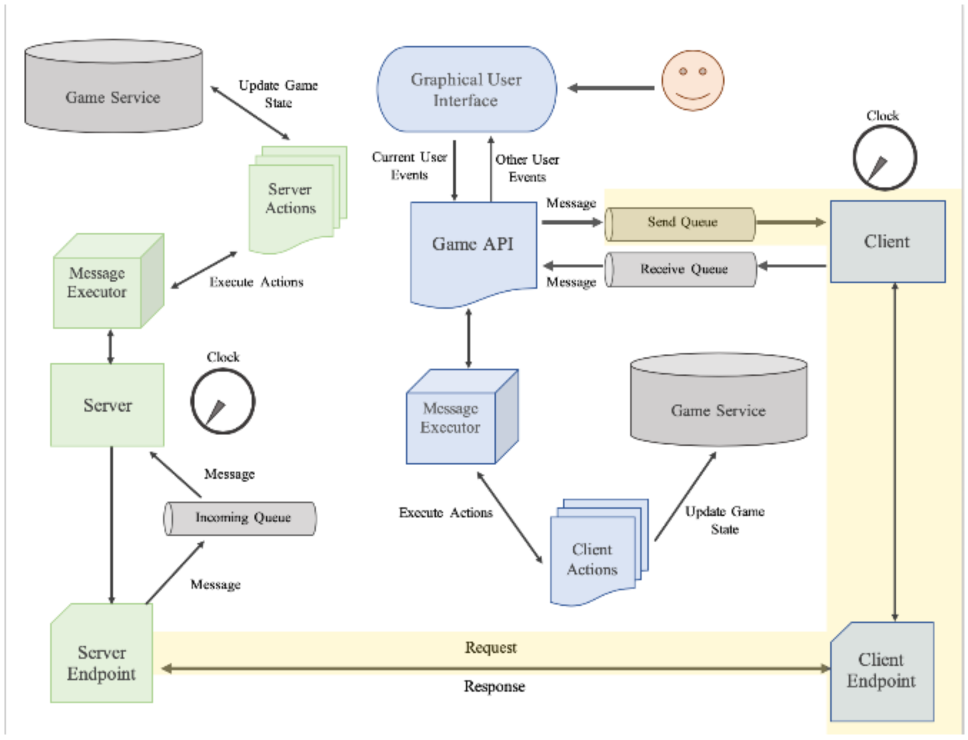




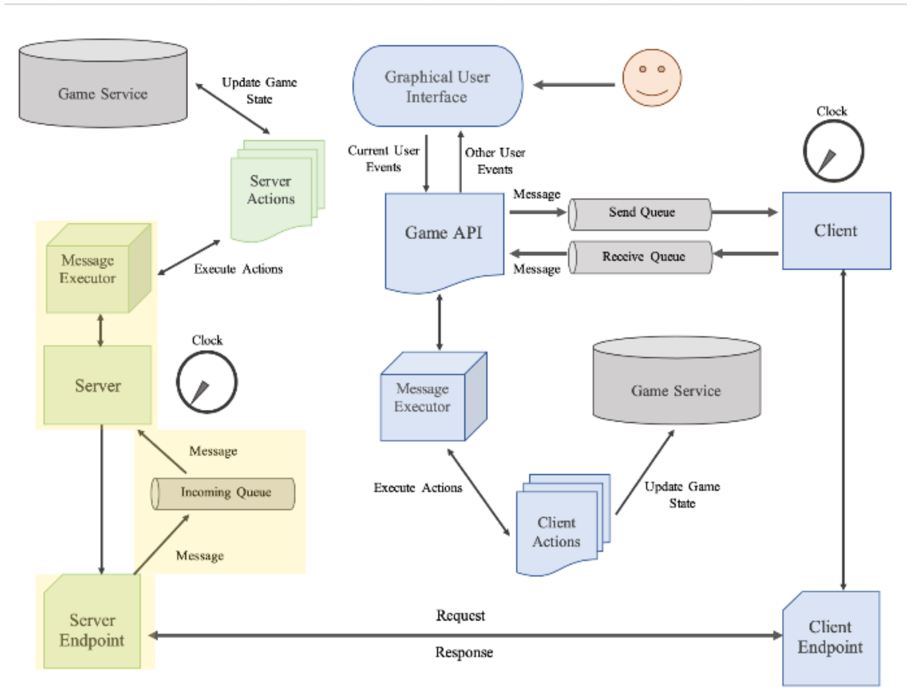
GameAPI prepares client requests and adds the messages to the send queue.



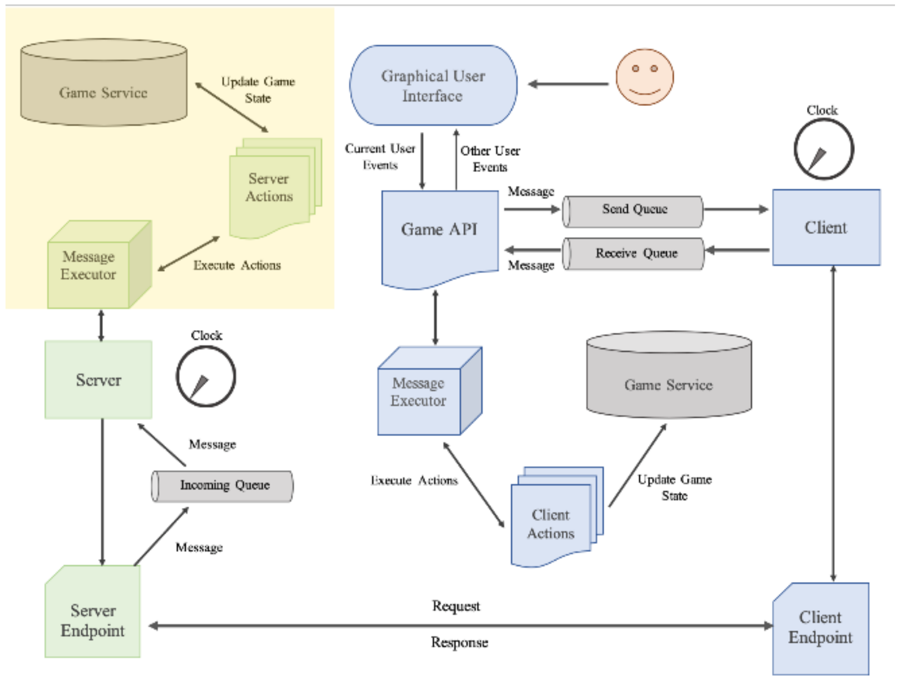
Client thread retrieves messages from send queue and sends it over the network to the server.



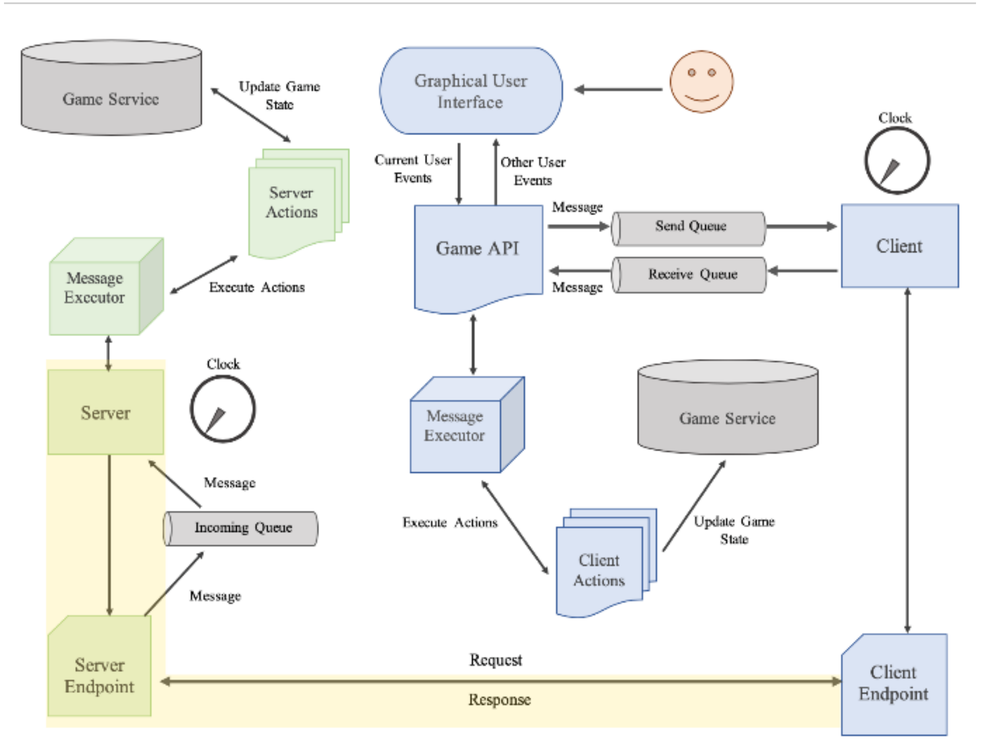
Server thread retrieves messages from its incoming queue and passes the messages to the Message Executor for processing.



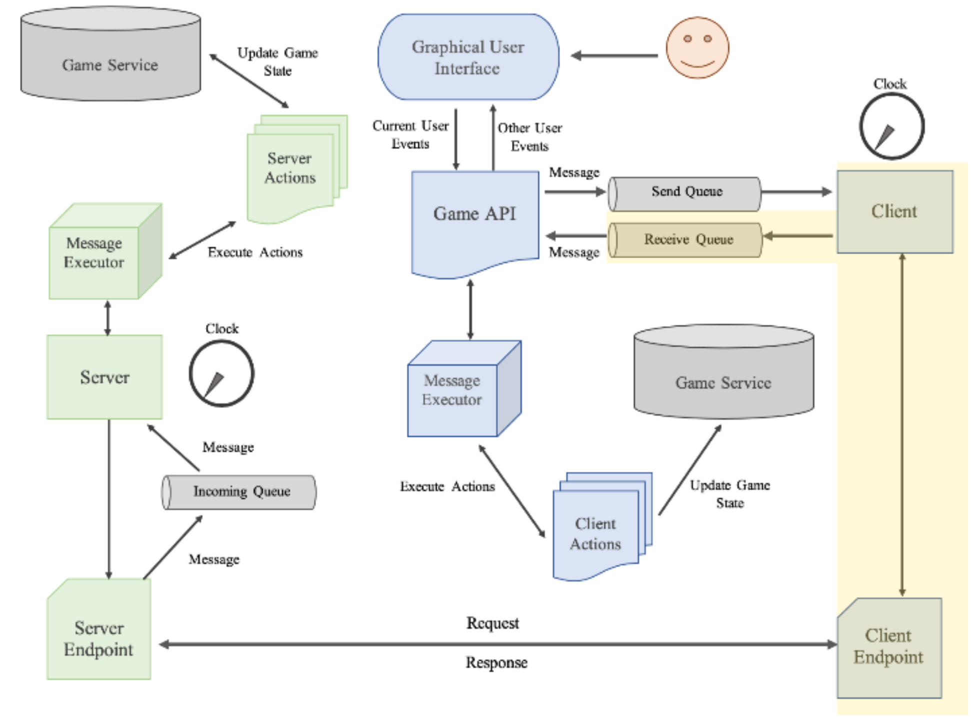
Message Executor invokes the relevant Game Action based on the message type. The Game Action updates the Game Service, prepares a broadcast response message and returns it to the Server through the Message Executor.



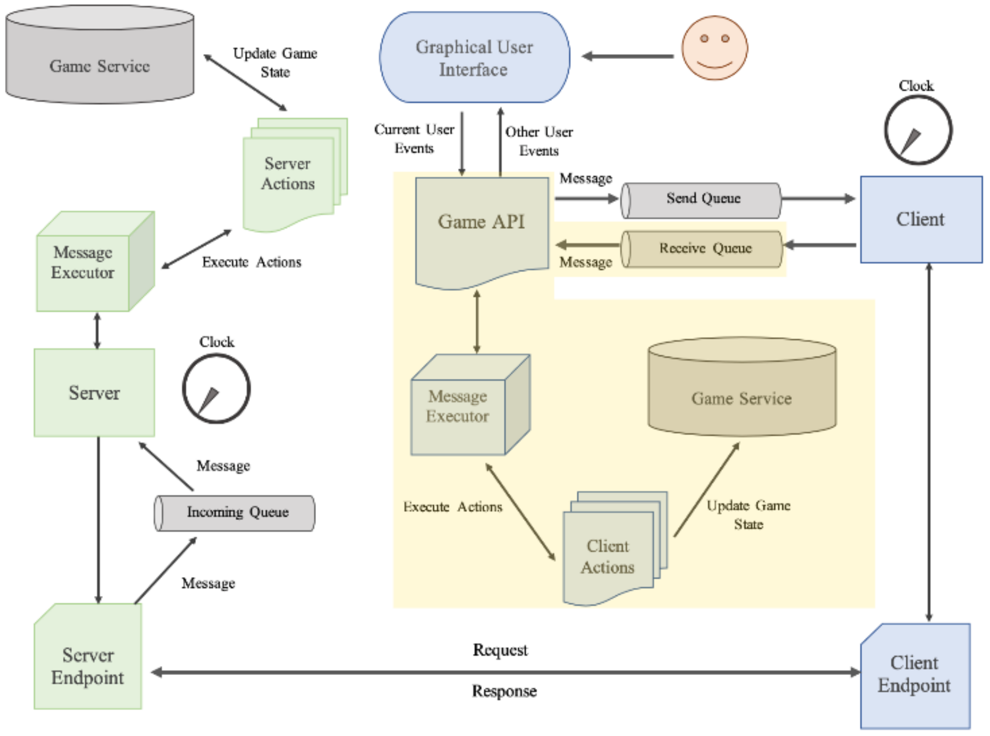
The returned response message is broadcast to all clients.



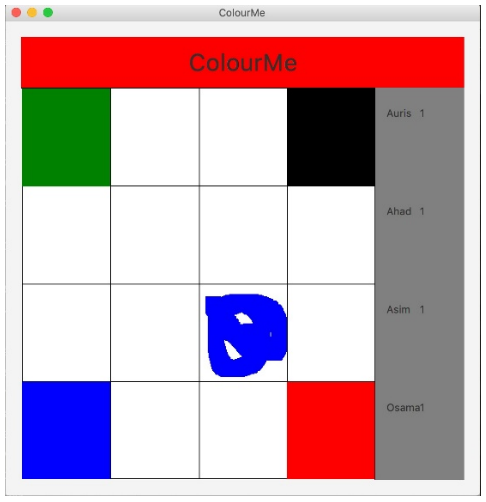
On receiving the broadcast message, each client pushes the message to receive queue.

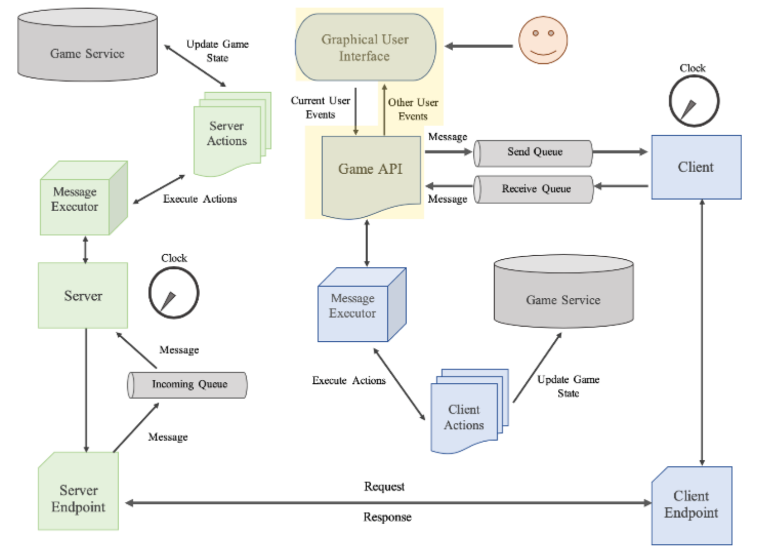


Game UI loop invokes the GameAPI to process any queued messages at regular intervals.



GameAPI fetches the message from the receive queue, processes it, and updates the client's local replica. The GUI update handler is invoked, which displays the new changes from other clients to the user.





3. Architectural Choices & Trade-offs

3.1 System Architecture Model

Evaluating the requirements of the game led to two model choices for the architecture: Client-Server and Peer-to-Peer.

Client-server is a centralized architecture with two components: client and server. A client requests a service from the server. The server is responsible for processing incoming client requests and sending the appropriate response. Peer-to-peer is a decentralized architecture model where every client acts as both a client and a server with access to shared resources.

An advantage of the client-server approach is the ability to centralize updates to the application state, which can prevent errors that might be caused due to inconsistent state. Centralized resource sharing on the server restricts direct access to the game state and provides better security compared to the peer-to-peer model. However, the client-server approach has a single source of truth, which can lead to server bottlenecks. To overcome this limitation, the server can be distributed across multiple machines or the application state can be replicated.

Requirements for the ColorMe application included synchronization between clients and fault-tolerance by replication. The peer-to-peer model requires multicast communication to coordinate the states of connected users. The client-server model provided better flexibility and simplicity and hence was a better fit for our application.

3.2 Programming Language

The most important aspect of any project is choosing a programming language, which best fits all the requirements. The two most viable options for this project were Java and Python.

Java is a statically typed, general-purpose, object-oriented programming language. Java code runs on the Java Virtual Machine (JVM), making it portable with as few dependencies as possible. In contrast, Python is a dynamically-typed, general purpose, programming language. It supports multiple programming paradigms including OOP and has a comprehensive graphics library which allows the creation of fully-featured games.

Java seemed better suited for our project due to its multithreading capabilities, faster execution time and a more robust debugging framework. While Python has a Global Interpreter Lock (GIL), which prevents simultaneous access to objects by multiple threads, Java offers better concurrency and parallelism support.

Java also has a faster execution time due to its Just-In-Time (JIT) compiler, which improves performance by compiling bytecode into native machine code. In contrast, Python is an interpreted language. Determining the variable type on runtime increases the workload of the interpreter, slowing down the program.

Python leaves objects vulnerable to mutation and bugs introduced aren't found until the code is executed. This can lead to operational breakdowns and extend turnaround time. In contrast, Java does not allow object mutations. Thus, Java was chosen for this project.

3.3 Network Protocol

WebSocket is a protocol that runs on top of TCP. It uses the HTTP connection establishment and then switches to its lighter WebSocket protocol. It offers higher performance than HTTP short/long polling due to its lighter header size (roughly 2 bytes). Applications involving WebSockets are real-time and send updates to multiple clients frequently. Clients must use their endpoint socket to connect to the server endpoint socket that is hosted on an HTTP server. Once connected, a session is established between the client and server, which they can use to communicate with one another.

Because WebSockets run at a higher level of abstraction than TCP and provide a better API than TCP sockets. Due to WebSockets' ability to handle different events asynchronously, its interface provided superior integration for our application. WebSockets also offer features such as built-in URL parameters, made broadcasting messages easier and provided higher performance than HTTP. Hence, WebSockets was the optimal choice for our project.

Note: Though the assignment definition states only UDP or TCP, we have received permission from the professor to use WebSockets.

3.4 Threads

One of the design trade-offs considered was the choice between a multi-threaded vs single-threaded server. The decision to perform and synchronize game state updates on the server required an efficient server implementation to prevent the server from becoming a bottleneck due to network and processing delays.

Upon benchmarking the number of requests needed to be processed by each client, it became evident that a single-threaded server could lead to lags for a user. Moreover, spawning a separate thread for each client did not seem feasible given the exponential increase in context switch time with each additional client. Considering these limitations, a hybrid design approach was taken.

In our design, three threads are spawned on the server: network, processor and timer thread. The network thread listens for incoming messages and adds them to a priority queue by timestamp. The processor thread processes incoming requests synchronously. A timer thread runs a logical clock and broadcasts server time at regular intervals to ensure the client clocks synchronized. The performance testing results for this design approach indicated 100% performance improvement for 10 concurrent clients over other approaches.

3.5 Client-Server Communication

The application requires coordinating player actions across all clients. To achieve this, the server stores a set of client sessions. Updates to game service replicas are synchronized and broadcast by the server. This communication model ensures a consistent game state across all clients.

Point-to-point communication would require a heavier client-pull communication as the players would have to request the newer updates to receive them and may receive many redundant updates.

Our current communication workflow is a hybrid between the client-pull and server push model. In this design, the client makes a request, receives a response and the response is then sent to all connected clients. The server also sends updates that aren't responses to client requests but serve other use cases such as synchronizing the clocks of all clients.

3.6 Message Format

A common shared message format for representing requests and responses is used to simplify the flow of data between clients and the server. Each message consists of a player id, timestamp, message type and json formatted data. The message is then serialized and transferred over the network. This format provides a flexible way to represent different types of data which are identified by message type. It also includes a player id indicating the player session and allows requests to be processed by timestamps.

3.7 Clock Synchronization

The absence of a clock synchronization mechanism can lead to faulty access to shared resources on a system which relies on events occurring at a specific time. The application design implements a logical clock that is used to synchronize client events. Each message from the client and server has a timestamp based on the logical clock. The clocks are synchronized across all clients (even across time zones) by broadcasting server time at regular intervals.

3.8 Thread Communication

Client machine has three threads at any point in time: GUI, game client and client clock. The GUI thread is responsible for rendering the actions of the current user and other active clients. The Game Client sends and receive messages to/from the server. These threads share two priority blocking queues. Priority blocking queue is thread-safe. Messages are prioritized by timestamp in these queues.

GUI thread prepares and enqueues client requests as messages. The game client thread then dequeues and sends these messages to the server. When a response message arrives from the server, the game client enqueues the message, which is later dequeued by the GUI and rendered to the user.

3.9 Network Latency

Cell update data is frequently transferred between clients via the server. To save network bandwidth, the application ignores coordinates of the cursor outside the canvas. Moreover, to minimize the number of client requests, a list of coordinates is sent rather than a single coordinate.

On a slower network, the bandwidth can be optimized further by adding every nth coordinate to the buffer. On average and over a course of several seconds, 50 requests are sent by a single player with n set to 1, and 17 messages with n set to 4. Increasing n and the buffer size comes at the cost of not receiving updates immediately so these parameters need to be optimized. In our tests, we observed responsive gameplay with the buffer size as 8 and n set to 1.

3.10 Space Efficiency

The GUI canvas objects render the strokes of a player using the coordinates of the cursor. The GUI returns a list of coordinates to the application for a cell. The application only stores the state of each cell as available, locked or colored and the associated player Id, as opposed to storing the list of all coordinates. This design has two advantages. First, it enhances the space efficiency of the application. Second, when a server disconnects, locked cells are released, and the cell state information stored is sufficient to recreate the game service.

3.11 Performance

Performance issues were encountered due to synchronous request processing on the server. Asynchronous, non-blocking colouring design has been employed on the client to enhance the colouring performance for users. The cell acquisition and release requests are synchronous to allow only a single client to acquire and release a cell. Cell update requests are asynchronous.

To prevent cell updates from hindering game performance, a client cell update request modifies the local replica of the game state. The update is instantly reflected on the GUI while a background process coordinates the update with other clients via the server. This design approach helped us achieve significant improvement in user interaction performance.

4. Characteristics

4.1 Fault Tolerance by Replication

The GameService maintains a 2D array of cells representing the board state and a hash map of players. The player object stores the player's color, score and IP Address.

Each client receives a list of IPs when it connects to the server. In the event of a server crash, each client reads the next IP for the host server from the list. If the host IP matches the local machine IP, the client releases the acquired locks, replicates its game service and spawns a server thread. Otherwise, it waits for a specified interval and then attempts to connect to the new server. Once all clients are reconnected, the server broadcasts its game service replica. Each client updates its local replica to synchronize game state with the new server and the game play continues.

4.2 Concurrency

ColourMe ensures coordination through mutual exclusion and clock synchronization. The server coordinates game state across clients by maintaining its own game service and preventing two players from simultaneously coloring the same cell. When two clients request acquisition of the same cell, the server prioritizes the request with a lower timestamp.

4.3 Coordination

Coordination in the game application is achieved through clock synchronization. The logical clocks prevent any client from getting unfair advantage due to network delays. Moreover, the cell acquisition and release requests are centralized through the server. This prevents direct access to the cell state by a client.

5. Conclusion

ColorMe successfully supports fault tolerance by replication, coordination and concurrency. Yet, the number of clients concurrently connected to the application is limited by scale. A future enhancement to the application can entail supporting a larger number of clients at a given point in time by decentralizing the server or using a peer-to-peer model. Furthermore, another enhancement could be to use a real clock for client synchronization as opposed to a logical clock to minimize the margin of error.