POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

Department of Computer Science

Department of Multimedia

Multimedia - Game Programming

Dmytro Maretskyi
S19340

Creation of multiplayer communication protocol and a backend server in a context of a turn-based game

Engineering thesis

Thesis supervisor

Warsaw, PJATK, 02, 2022

Abstract

The work focuses on implementing a custom networking protocol from the ground up tailored for turn-based game. There are existing solutions that help developers add multiplayer support to their Unity-based games, but the majority is optimized for fast-paced games like first-person shooters. They also make big influence on how backend services are written, often requiring dedicated servers to run a headless instance of Unity engine.

The custom protocol is based on WebSocket transport and JSON-RPC layer for serialization and remote call functionality. The protocol is engine-agnostic which allows the architecture backend services to be uninfluenced by Unity.

This work explores implementing a turn-based game using the custom protocol, with the game client written on the Unity engine and distributed via a web build, and a backend server that runs in a .NET Core runtime. Due to the network protocol being lightweight, the resulting backend server turned out to be more scalable than the solutions involving other multiplayer frameworks.

The work also goes into detail on how code sharing and reuse can be achieved between a Unity game and a traditional .NET solution.

Keywords

- Multiplayer
- Unity
- Networking
- Backend infrastructure
- Turn-based games

Table of Contents

1. Introduction

The landscape of multiplayer solutions for the Unity game engine is fast evolving. There are numerous official and unofficial frameworks that aid developers in adding multiplayer functionality to their games. The majority of those solutions aim to be general purpose and as such are optimized towards fast-paced games like first-person shooters. In the context of a turn-based game this leads to increased complexity and load on the game's infrastructure.

This work focuses on a creation of a custom networking protocol tailored for a turn-based game. It explores how the game's logic can be extracted into an engine-independent code. This work describes how a more-performant and simpler backend server could be built in such case.

List of goals:

- Design and implement a client-server networking protocol for a turn-based game on a Unity engine.
- Have the networking implementation be compatible with modern browsers allowing for web distribution of the game.
- Compare the implementation with the existing solutions available.
- Evaluate the impact on the complexity and performance of the client and the server.
- Research deployment and distribution options for the game client and backend infrastructure.

List of technologies used:

- Unity game engine.
- .NET Core runtime and SDK.
- WebSocket protocol as well as client and server implementations.
- JSON-RPC 2.0 protocol.

2. Project description

2.1. Definitions

Computer game – is an application where a player (or a number of players) partakes in an organized form of play. Computer games have a set of rules that are enforced and an objective for player to achieve. They are usually augmented with graphics and sound to give players better immersion into the experience.

Turn based game – a form of a game where players are restricted to making any input only on their turn. During this period of time the player has an exclusive control of the game.

Strategy game – is a game with an emphasis on player's decision making skills. Those games usually require players to think a number of moves ahead to determine an optimal strategy with respects to opponent's responses.

Multiplayer game – is a game where multiple players can play at the same time. This can happen either at the same device locally or by each player having their separate device which are connected via internet.

Network protocol – a specification of how two or more independent programs communicate over the network. Defines the message encoding, what messages are sent, and in what sequence.

RPC – remote procedure call. Is a communication technique for client-server applications, where a client sends a message with a set of parameters and expects to get a response back. This is similar to a normal function call in a computer program, but the callee is located in a different process or machine and the communication is done via network.

Matchmaking – a service in a context of a multiplayer game of creating game sessions automatically in contrast to players manually choosing the server or other players they play with. This often involves players entering a "Matchmaking queue", a state where they are waiting for the server to create a game session for them. Matchmaking services can often use different metrics to pair players in a fair way. They may take into account player's skill level or the strength of their in-game loadout.

Player ranking – a system of measuring the skill level of the players usually resulting in a numeric value used to position the player among others. The systems can be additive, where a player gets a number of "experience points" based on the activity they partake in, or rating-based where points are added for the matches players win and subtracted for the matches they lose.

2.2.Game description

The game (Figure 1) is played between 2 players on a rectangular grid, usually 25 by 25 cells, but other sizes are also supported. Each turn a player places his unit (also called “dot”) on a vacant grid cell. All units are the identical and only differentiated by the color, signifying to which player they belong. If, after placement, by connecting player’s adjacent dots (either orthogonally or diagonally) a cycle can be formed that also surrounds one or more opponent’s dots, the player captures the opponent’s units.

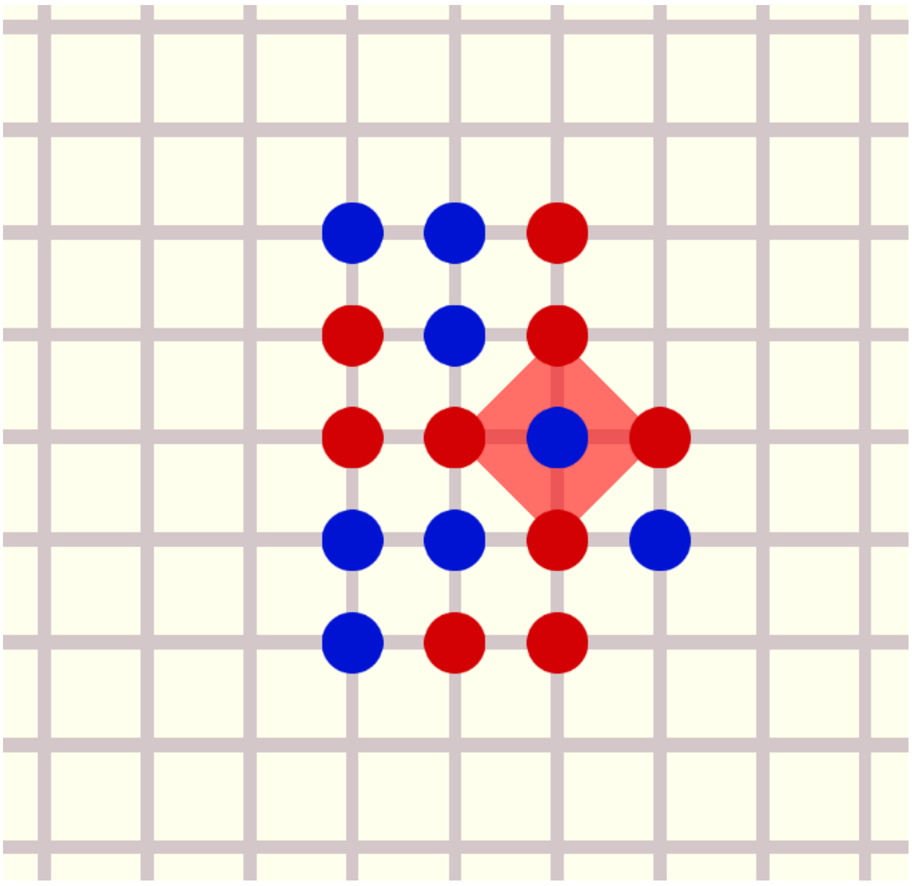


Figure 1. Red units captured a blue unit.

Each captured dot grants the player 1 point, as well as effectively “disables” the enemy unit - it can no longer participate in forming cycles needed for capture. The goal of the game is to get more points than your opponent. The game is played until all players decide there are no more moves that can improve their position or there are no legal moves left.

2.3.Multiplayer aspects

The focus of this work is to research the process of designing a network protocol to add multiplayer functionality to a turn-based game. The game should also provide rich multiplayer experience where you can play against other people over the internet. There should be an automatic matchmaking system, where you can queue up to have a game with the next available player.

The clients will be communicating with a central backend server that accepts connections from game clients, handles the matchmaking queues, and can simulate many concurrent games. The network protocol should be structured in a way that prevents players from cheating (making illegal moves). The games server should hold central authority and validate every interaction made by players.

3. Project architecture

3.1. Overview

The project can be split among two parts: the game client that uses Unity and a backend server that maintains the game state. Both are written in C#. Between them there are several shared classes such as data models describing the game state and the moves made by players. The high-level overview of the project structure is presented on Figure 2.

The backend server hosts a WebSocket API that game clients connect to. Choosing WebSockets as a transport layer provides us with an ability for two-way communication: game server can push updates to the clients and clients can send moves made by players. Also this increases the flexibility for the backend deployment as almost all hosting providers support WebSockets.

There's also an additional test module hosting automated unit-tests for the core data models and algorithms.

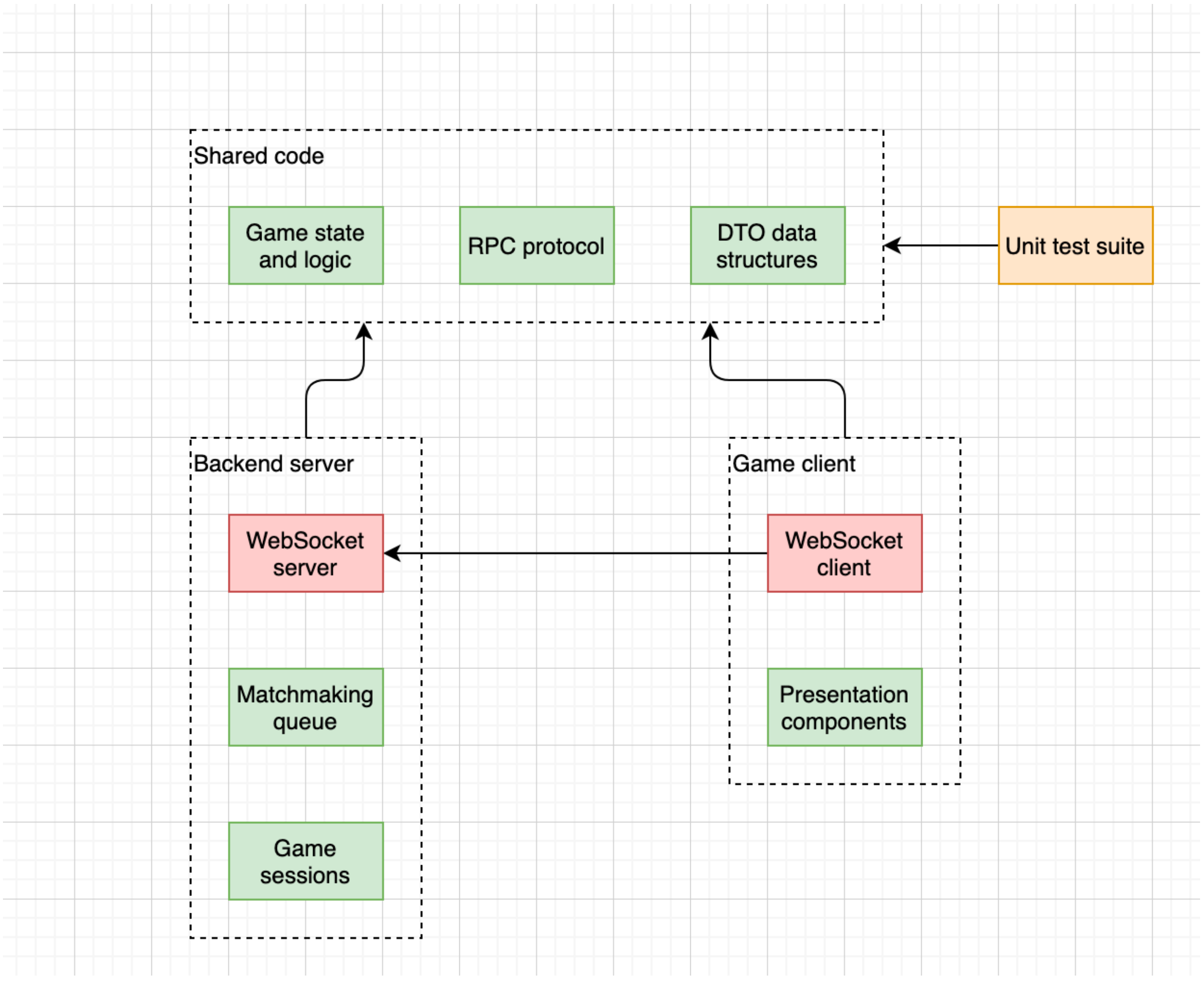


Figure 2. Project overview.

3.2. C# build pipeline

One of the difficulties when sharing code with a Unity project is that Unity does not support multi-assembly builds. Meaning that all C# sources must be a part of a single C# project when compiling the game client. Also, C# project and solution files are auto generated, which means making any manual changes are unfeasible, as they would be lost next time Unity Editor is refreshed.

This poses a challenge, because a usual way to share code between the backend server and a client would be to introduce a third class-library project with shared classes, that two other projects depend upon. Since that's not possible the only solution is two have two parallel project structures: one for the backend and the second one for the Unity build. The shared sources would be included in both structures and compiled separately.

Unity requires all C# sources to be placed inside "Assets" directory. This also includes the shared classes. This would force us to move away from then canonical project structure where each module resides in a separate directory on the top level. There were numerous attempts to go back to the flat structure utilizing filesystem symlinks. Unfortunately, Unity as well as C# build tools do not have good enough support for them, and often they were not recognized as a link to another directory.

Ultimately the following project structure was settled on:

- Backend project directory.
    - Backend "csproj" file.
    - Backend sources.
- Test project directory.
    - Test "csproj" file.
    - Test sources.
- Unity project directory.
    - Assets directory.
        - Unity assests.
        - Core project directory (shared sources).
            - Core "csproj" file.
    - Unity autogenerated "csproj" file.
    - Unity autogenerated solution file.
- Solution file for the backend and core projects.

3.3. Dependency management

It wouldn't make sense for every project to re-implement from scratch common standardized components such as serialization, networking and others. So, to speed-up and ease development we often rely on existing solutions and already developed libraries and frameworks. Initially, and often in older programming languages like C and C++, developers would copy the library into the source tree of the project.

This approach poses many problems, among which is difficulty of installation, as it must be performed manually. And the difficulty of configuration as there's no enforced standard for how the reusable library code should be structured. Upgradability is also an issue: when a new version of the dependency gets released, developers would need to perform the same steps of copying the library code into their project's codebase and configuring the compilation pipeline. The process gets even more difficult when transitive dependencies come into play. Transitive dependencies are dependencies if the library or package your project depends on directly.

To solve those issues package managers were created. Package manager is a tool responsible for installing and upgrading project dependencies and their transitive dependencies. Code is typically structured into packages that are identified by their unique name. Packages are versioned and the project's manifest specifies what version of the package it requests to be installed. The most common versioning schema is "SemVer" (semantic versioning) (1).

SemVer versions consist of three integer parts: major, minor, and patch number. For example, 1.0.12 – would be a valid SemVer version. Each part has its own meaning to represent the type of changes that were introduced. The major number is incremented when breaking changes were introduced – changes that, after upgrade, would break existing code that depended on the previous version of that package. The minor version is incremented when new features were introduced, but in a backwards-compatible way, where it wouldn't break the existing code after an upgrade. The patch number is reserved for bugfixes.

This type of versioning scheme allows developers to specify not concrete versions of their dependencies, but whole ranges of versions they are compatible with. Most commonly, developers allow minor and patch numbers to change up to the next major release. 2.4.12 <= x < 3.0.0 would be an example of such a range.

Such standards allow packages to be distributed easily and often package managers come with their own package repositories, where developers can publish their packages and others can download and install dependencies with a package manager. The installation usually is as easy as issuing a command to the package manager, to install a dependency, by specifying its name and the requested version. In such a setup, only the manifest file is committed to the VCS, and package files are excluded from source control, as every developer can easily recover them from the manifest using a package manager.

The de-facto package manager for C# (and the whole .NET ecosystem) is NuGet (2). It comes with its own package repository hosted on https://www.nuget.org/. As of the time of writing, the repository contains over 280 thousand packages. Packages are versioned using SemVer.

This project uses NuGet to manage dependencies for the backend server. Among which is the JSON (3) serialization library and web-socket networking library. For Unity, a different package manager is used: Unity Package Manager (UPM for short).

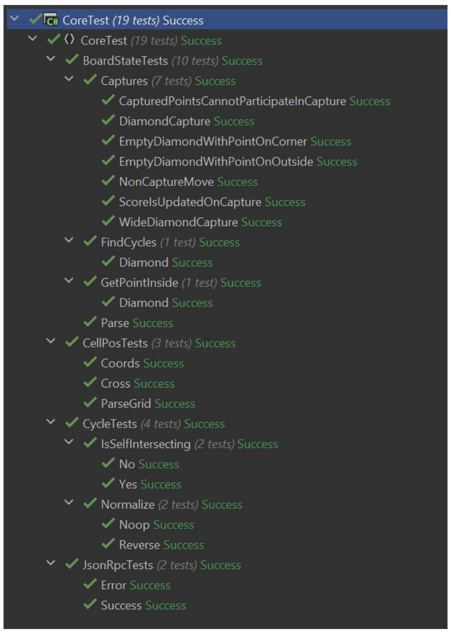
UPM has a package distribution method. NuGet distributes packages in their compiled form: as a DLL (Dynamically Linked Library) [add reference]. This has the advantage of the source already being compiled, which avoids any compilation errors in packages and reduces the size of installed dependencies. There's however a disadvantage: assets including textures, models and prefabs cannot be distributed in such a way. UPM installs packages with their C# source code and assets. The package code is typically included in VCS.

The UPM ecosystem is quite young and there are often packages missing. While there's no official support for NuGet in unity, there's an open-source unofficial plugin to replicate NuGet package manager behaviour for Unity projects (4).

3.4. Unit testing

The game state and associated algorithms outlined in the section 4 of the work can get quite complex and have many edge cases. Testing them manually is a very time-consuming and error-prone process. For those reasons an automated test suite is used to verify the correctness of the algorithms. A unit test suite is contained in the "CoreTest" submodule. It consists of a series of functions each executing a piece of game logic following a specific scenario and asserting that the output matches the expectation. The tests might contain general scenarios such as modifying game state by making moves or test concrete functions such as querying game state for cycles in the graph of units.

Since unit tests are set-up as individual methods, a test-runner is needed to orchestrate the test execution. It should execute each test methods, gathering the results or handling any thrown exception and then report the test execution result. NUnit is used for that purpose. It is widely adopted in the .NET ecosystem. Has IDE integrations (Figure 3) and supports running subsets of tests or running individual tests with debugger enabled.



```
CoreTest (19 tests) Success
  CoreTest (19 tests) Success
    BoardStateTests (10 tests) Success
      Captures (7 tests) Success
        CapturedPointsCannotParticipateInCapture Success
        DiamondCapture Success
        EmptyDiamondWithPointOnCorner Success
        EmptyDiamondWithPointOnOutside Success
        NonCaptureMove Success
        ScoreIsUpdatedOnCapture Success
        WideDiamondCapture Success
      FindCycles (1 test) Success
        Diamond Success
      GetPointInside (1 test) Success
        Diamond Success
      Parse Success
    CellPosTests (3 tests) Success
      Coords Success
      Cross Success
      ParseGrid Success
    CycleTests (4 tests) Success
      IsSelfIntersecting (2 tests) Success
        No Success
        Yes Success
      Normalize (2 tests) Success
        Noop Success
        Reverse Success
    JsonRpcTests (2 tests) Success
      Error Success
      Success Success
```

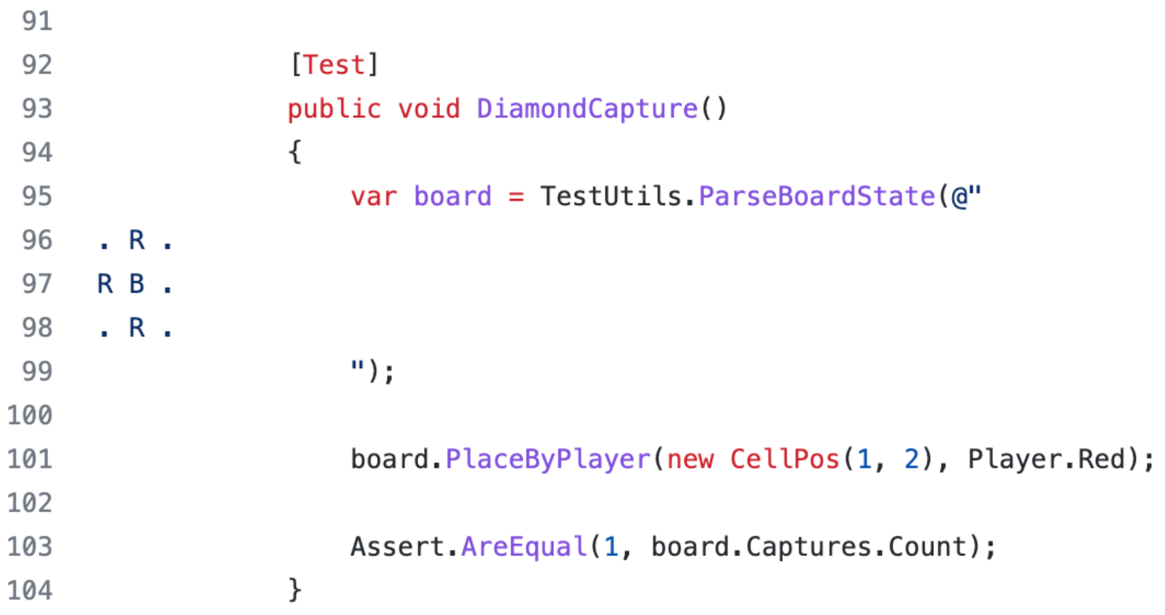
Figure 3. Successfully passing unit tests.

Every unit test usually consists of three sections:

- Assemble – setting up the initial state of the tested module. For testing state transitions it might be the state before making a mutation.
- Act – run the code which is being tested. For example: make a move on the game board.
- Assert – assert that the result of the executions matches your expectations.

In the unit test displayed on the Figure 4 the three stages are apparent. The “assemble” stage sets up the game state from a predefined scenario. In this case it’s a state just before a capture. Next step is to make a move by calling the API. The move is for the Red Player to place the final unit completing the cycle. This will mutate the board state in reaction to the move. In the final stage we assert that the game state after the move has recorded that capture.

To achieve good impact from unit-tests it’s important to have them cover edge cases and different branches in code. This might involve describing many different test cases, which increases the amount of code used for testing. Maintaining a large amount of test code can cause problems, so it’s important to optimize the test code size as much as possible.



```

        [Test]
        public void DiamondCapture()
        {
            var board = TestUtils.ParseBoardState(@"
. R .
R B .
. R .
            ");

            board.PlaceByPlayer(new CellPos(1, 2), Player.Red);

            Assert.AreEqual(1, board.Captures.Count);
        }
```

Figure 4. Example of a unit test.

In the example above a utility function is introduced – “ParseBoardState”. It is used to create an initial game state with a minimal amount of code. The way it is achieved is by parsing board state for a string. This also has a benefit of providing good visualization of the board state. The string format consists of a rectangular grid of characters separated by spaces. Each character corresponds to a cell on the grid. Grid size is determined automatically from the input string: by measuring the number of lines, and the number of characters in the first line. Only “.”, “R”, and “B” characters are allowed, specifying empty cell, cell occupied by the red player, and a cell occupied by the blue player respectively.

“TestUtils” class is used to host aforementioned and other utility functions. A different function can be used to generate a sequence of grid coordinates which is used for testing algorithms related to cycle search. In that case the similar string format is used, but instead grid cells have numbers which represent indexes of grid cells in the resulting array.

3.5.Deployment and hosting

A couple of factors were taking into account when choosing a hosting platform:

- Support for the chosen architecture – ultimately the hosting platform would need to be able to run a .NET Core binary which backend server is compiled to.
- Flexibility – hosting platforms vary in the amount of control they give to the users. It can range from hosting lambda-functions (5) where users define individual request handlers and don’t have any much control, to dedicated servers where a bare-metal machine is provide and users can configure anything starting from the installed OS.
- Cost
- Ease of use – the platform should be easy to manage and not pose any roadblocks. This usually correlates inversely with flexibility as the most flexible systems require the most configuration. Although trying to fit a system into a hosting platform full of incompatible restriction could be a challenging or even impossible task.

With that in mind, three hosting providers were examined:

- Hetzner – they provide dedicated servers. A wide selection of operating systems can be installed. Installation can take upwards of 24 hours after an order is made. An installation fee is also taken on every order. This is by far the most flexible hosting solution, although it comes with a price of management overhead and higher cost.
- Digital ocean – they provide VPS (Virtual Private Server) where users get a virtual machine on a shared dedicated server. Users can pick the amount of resources they need allocated to their VPS: CPU, RAM, disk space. A machine can be ordered from their web UI. Installation is automatic and the server is usually ready within minutes. Costs start at $5 per month for the most basic configuration.

- Heroku – they provide servers but rather provide hosting for apps. There are a number of tech stacks they support, including Node.JS, .NET Core, and Ruby on Rails. Users don't have control over the machine used to run the app. The app is provided with the configuration parameters to use, such as the port number for it to listen. A number of optional services can be also added, for example a PostgreSQL database. Interestingly, they offer a free plan where the hosting is free-of-charge but the app networking throughput is limited and the app is put to sleep when it is not used continuously. The paid plans start at $5 per month.

While Heroku was considered as a strong candidate, the amount of flexibility having a separate server offers was a deal-breaker, especially considering that the pricing was similar. Using a free plan on Heroku would significantly hinder the game performance, as the backend server would be put to sleep from time to time.

In the end, Digital Ocean was chosen as a hosting platform, running an Ubuntu operating system. Provisioning can be done through SSH. The server's firewall is configured to accept TCP connections on a provided port. And a .NET Core runtime is installed to run the game server binary.

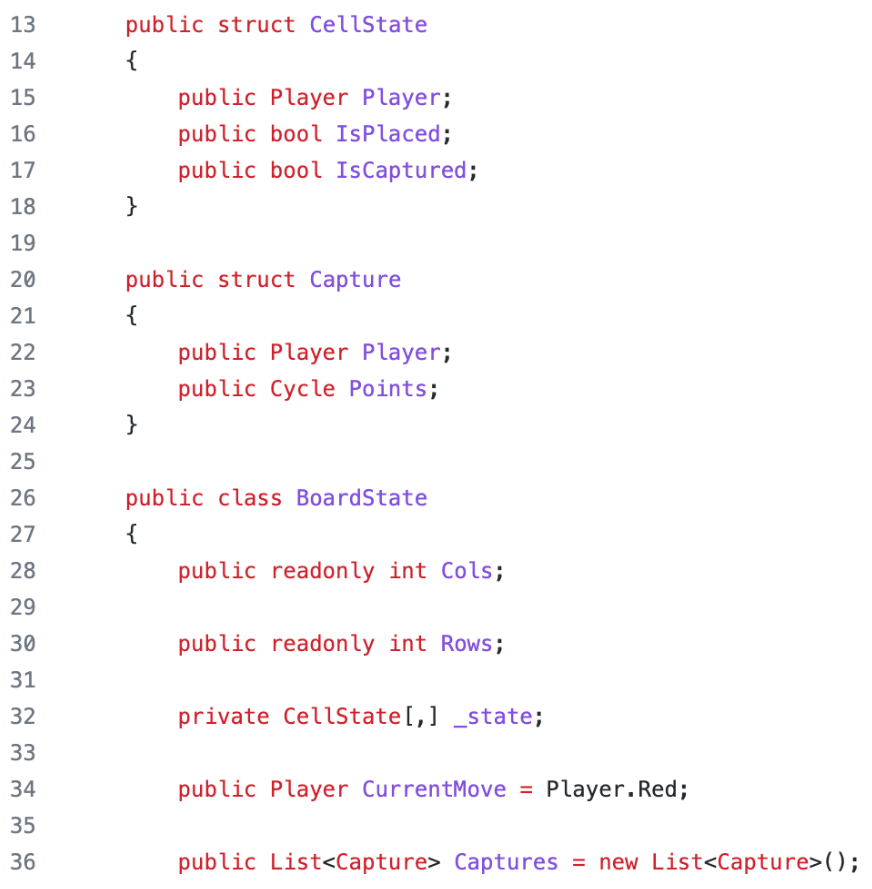
4. Game state

4.1.Game state data structure

The game state consists of a 2-dimensional array describing each grid cell position as well as a list of captures that were performed during the game. The code snippet defining the data structure is depicted on the Figure 5**.**

Each cell state consists of:

- Vacancy flag - whether this cell was claimed by the player.
- Their player that claimed this cell - only in case this cell is claimed.
- Captured flag - whether the dot placed in this cell is captured by the opponent.



```
public struct CellState
{
    public Player Player;
    public bool IsPlaced;
    public bool IsCaptured;
}

public struct Capture
{
    public Player Player;
    public Cycle Points;
}

public class BoardState
{
    public readonly int Cols;

    public readonly int Rows;

    private CellState[,] _state;

    public Player CurrentMove = Player.Red;

    public List<Capture> Captures = new List<Capture>();
```

Figure 5. Board state.

Each capture entry in the list includes a player that made the capture as well as the list of grid coordinates of dots that participated in the enclosure of enemy units.

4.2. Updating the game state with new move

With every move we perform a cycle search to find new enclosed areas. Dots placed on the square grid form an effective graph with their neighbours. A depth-first search algorithm is used to find cycles in this graph. Cycles are also validated to not be composed from multiple larger cycles.

After the cycle is found a variant of flood-fill algorithm is used to select all grid coordinates that are enclosed by this cycle. Then, if any of those contain enemy units, those units are marked captured, and the cycle is added to "captures" list.

4.3. Representing cycles

One of the core data-structures used by the algorithm is a cycle. Cycle is closed sequence of distinct points on a grid. Each point must be adjacent to the previous and next points in the cycle either horizontally or diagonally. In-memory cycles are represented as arrays of grid coordinates (row and column).

There are many ways that a single set of points can be ordered in an array: different points can be chosen as the begging of the cycle, and the enumeration direction can be either clockwise or counter-clockwise. This makes it difficult to perform equality comparisons, as the algorithm would need to account for those variations. In the same way, it makes it difficult to build a stable hash function, which would be useful for performing efficient lookups in "Set" or "Map" data structures.

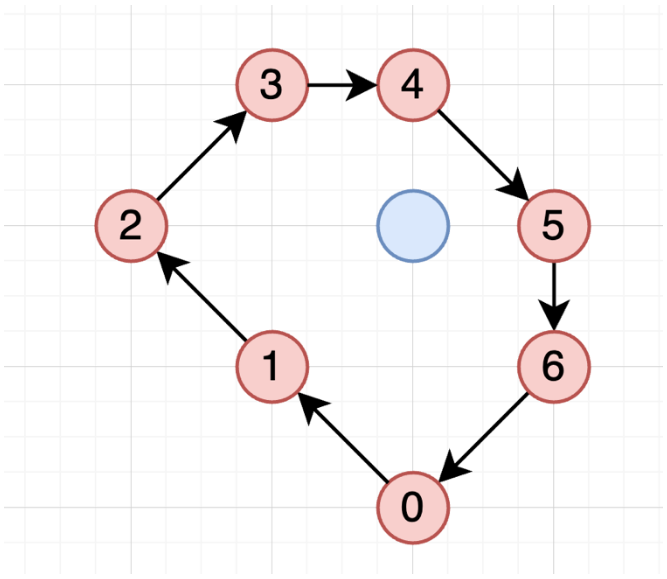


Figure 6. A cycle of 7 red points around a single blue one.

For those reasons cycles are always stored normalized. Normalized cycles must have their points ordered clockwise. And the starting point must be the one with the smallest row coordinate. If there are multiple points on the lowest row, the one with the smallest column coordinate is taken. As a reminder, rows and columns are counted from the bottom-left corner of the game board, starting from (0; 0). Figure 6 displays a normalized cycle. Numbers in points depict the index of said point in the internal array representing the cycle.

Normalization is performed using "Normalize" method in the "Cycle" class. It will mutate the instance it is called upon to have it be normalized. Normalization is done in two steps:

- Determining the current enumeration direction (clockwise or counter-clockwise) and reversing the array if the direction is counter-clockwise.
- Searching for the minimal point in the array (using aforementioned comparison rules) and rotating array elements so that the minimal point is at the index 0.

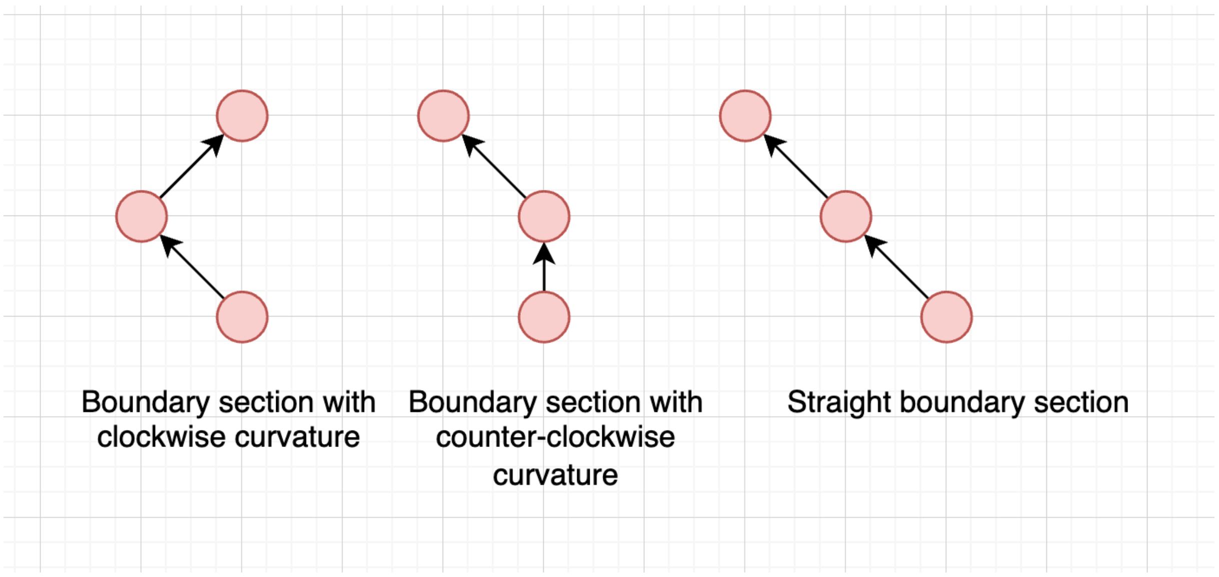

Boundary section with clockwise curvature
Boundary section with counter-clockwise curvature
Straight boundary section

Figure 7. Examples of different curvatures at a given point in cycle boundary.

For the first stage, while reversing an array is easy using the standard library function, determining an enumeration direction can prove tricky. An operation similar to vector cross-product is used. Algorithm is as follows:

1. For each point in cycle, consider a vector pointing from the previous point in the cycle to the current one, and a vector pointing the current point to the next one in the cycle.
2. Normalize the two vectors so their lengths are equal to 1. This is required because diagonal connections will have a length of $\sqrt{2}$, while horizontal ones will have a length of 1. For the next steps it is important that all vectors are of equal length.

3. Extending the vector space to the third dimension, take a cross product of two vectors: $v_{prev} \times v_{next}$. This vector is guaranteed to be parallel to the third-dimension basis.
4. The z coordinate (third dimension) of that vector will be positive if the curvature in that point of the cycle is clockwise, negative if it is counter-clockwise, and 0 if the line connecting cycle points is straight in that spot. Refer to Figure 7 for more details.
5. Summing up z coordinates calculated for each point in the cycle, we get a total metric for the entire cycle. It will be a positive number if ordering in the cycle is done counter-clockwise, and negative otherwise. This algorithm will also be resistant to any local fluctuations in the cycle boundary curvature, such as when the boundary shape is concave.

After obtaining the metric for the whole cycle it is just the matter of reversing the array if the current ordering is counter-clockwise. Next step in normalization is positioning the minimum point at the index 0. This is achieved by first, finding the current index of the minimum point by linearly searching the whole array. And second, copying all of the points to a newly allocated array of the same size. Copying is performed with an offset equal to the index if the minimum point, with indexes being calculated modulo array size. This could be done more efficiently by performing a series of swaps in-place, which would allow to get rid of extra array allocation, but performance boost gained is not significant in the context of this application.

There's another, implicit, assumption: all cycles must not be self-intersecting. Meaning the only points allowed to be adjacent are the ones directly following each other in the cycle. The cycles that are self-intersecting can be split into two or more different cycles with some common points. This is not enforced in the "Cycle" class directly, but the following algorithms only operate on non-self-intersecting cycles. And self-intersecting ones are discarded immediately.

4.4. Cycle search algorithm

Depth first search is used to find cycles in the graph. Search is always started from the dot placed on the current move. This optimization can be made because all cycles that were formed without the dot placed on the current move were already discovered when running the algorithm on previous moves.

To implement the algorithm a recursive approach is used with a stack to store a sequence of coordinates being currently processed. First a recursive function is executed on the starting point - the dot placed on the current move. It pushes the coordinates to the stack and then executes itself on all neighbours that can participate in capture. If a valid cycle is found, i.e. we reached the starting point, the sequence of points is stored as one of the cycle candidates. After all of the neighbours have been processed the current position is popped from the stack, and the execution flow is returned to the caller.

```
        public IEnumerable<Cycle> GetCycles(CellPos from, Player player)
        {
            var cycles = new HashSet<Cycle>();

            var stack = new Stack<CellPos>();
            stack.Push(from);

            void Rec()
            {
                foreach (var neighbour in GetActiveNeighbours(stack.Peek(), player))
                {
                    if (neighbour == from) // Found a cycle
                    {
                        Console.WriteLine(stack);
                        if (stack.Count > 3) // Prevent short cycles
                        {
                            var cycle = new Cycle(new List<CellPos>(stack.ToArray()));
                            if (!cycle.IsSelfIntersecting())
                            {
                                cycle.Normalize();
                                cycles.Add(cycle);
                            }
                        }
                    }
                    else if (!stack.Contains(neighbour)) // Prevent self-intersecting cycles
                    {
                        stack.Push(neighbour);
                        Rec();
                        stack.Pop();
                    }
                }
            }

            Rec();

            return cycles;
        }
```

Figure 8. Cycle search algorithm

If during the execution we reach a neighbour of any point that already in the stack, but not the initial one, we terminate early, because that cycle could have been formed without the initial dot. And hence, was already processed on previous moves. Figure 8 shows the code snippet implementing the algorithm.

Additional optimization can be achieved by utilizing the union find algorithm (6).

4.5. Encircled unit search algorithm

When we detect a cycle, we can search all points inside that cycle to find out whether a capture was performed. To make the search a way to enumerate all points contained in the area bounded by the cycle is need. First a single point known to be inside that area is taken, and then a flood-fill algorithm is used to enumerate all points within that area (7). Each point is then examined to determine if it's valid for capture.

The first step, calculating the starting point for flood-fill in the area bounded by the cycle, can be achieved using the same reasoning as for cycle normalization. We already defined a function to examine relative direction between neighbouring points. Assuming all cycles are normalized, their points must be enumerated clockwise. This means that, relative to the direction of the boundary, points inside the region would be to the right side, and outside the boundary – to the left. Picking a point on the cycle's boundary and examining all of its neighbours to satisfy above conditions allows us to calculate the starting point for the flood-fill algorithm.

The flood-fill itself is an algorithm that recursively visits all neighbouring points from a given starting position (7). It is implemented using a queue and an array to mark visited points. Queue contains a list of points to be visited. It is bootstrapped with the initial point. The algorithm goes as follows:

1. Pop the point from the front of the queue. If the queue is empty algorithm is finished.
2. Set the visited flag for that point.
3. Perform any operations meant for visited points. In this case the point is checked for any potentially capturable units.
4. Enumerate all neighbours of the point. If they weren't visited already and satisfy conditions for flood-fill (not part of cycle boundary), push them to the back of the queue.
5. Go back to step 1 and repeat the algorithm until the queue is empty.

Figure 9 shows the C# code for the algorithm.

4.6. Iterator methods in C#

In C# programming language there's a feature called iterator methods (8). This is a special case of a method that instead of returning a single value, can generate a possibly infinite sequence of values. This is done by specifying the return type of that method as "IEnumerable<T>" and using "yield return" keyword. In this case the execution of the method will continue past the return statement, potentially allowing it to return multiple values. In practice this is achieved by returning a generator object, containing a suspensible version of the method code. This object can be repeatedly queried, for example using a “foreach” loop, to produce the next value in the sequence. Each query will run the portion of the method code up to the next “yield return” keyword. Then the execution will be suspended, and method scope variables will be serialized.

```
public IEnumerable<CellPos> EnumeratePointsInCycle(Cycle cycle)
{
    var inside = GetPointInside(cycle);
    if (inside == null)
    {
        yield break;
    }

    var cyclesPlayer = Get(cycle.Points[0]).Player;
    var visited = new bool[Rows, Cols];
    var queue = new Queue<CellPos>();
    queue.Enqueue(inside);

    while (queue.Count > 0)
    {
        var next = queue.Dequeue();
        visited[next.Row, next.Col] = true;
        yield return next;

        foreach (var neighbour in GetDirectNeighbours(next))
        {
            if (!visited[neighbour.Row, neighbour.Col] && !CanParticipateInCapture(neighbour, cyclesPlayer))
            {
                queue.Enqueue(neighbour);
            }
        }
    }
}
```
Figure 9. Example of an iterator method.

There are numerous applications for such construct, but in this work it is used to optimize algorithms based on sequence manipulation. Consider this: many steps in the previously described algorithms can be considered either acting on a sequence of values. To find what units have been captured after a move we need to consider points in all new captured regions. To get all new captured regions, we first iterate over all cycles created by the newly placed point. To get all cycles we recursively iterate over neighbours of the starting point. Those seem like logical boundaries by which the algorithm can be decomposed into different functions. The problem here is that many of those functions return sequences or sets of values: getting neighbours of a given point returns a set of points. Enumerating cycles formed around a given point returns a set of cycles. Applying a flood-fill algorithm returns a set of points inside a given cycle. Using dynamically allocated containers such is array or Set here would have a drastic impact on performance, as there would be a big overhead in memory allocation. Also the entire resulting set would need to be pre-computed and stored in memory before returning from the function. This is often not needed, as the resulting elements are consumed one-by-one and access to the whole set at a time is not required.

This is where iterator methods are useful. They allow us to use the same principles for code decomposition, without sacrificing on performance. Figure 9 shows an example. This method enumerates all points inside a boundary of a given cycle using a flood-fill algorithm. “GetDirectNeighbours” is also an iterator method. Here it is consumed by a “foreach” loop.

5. Game client

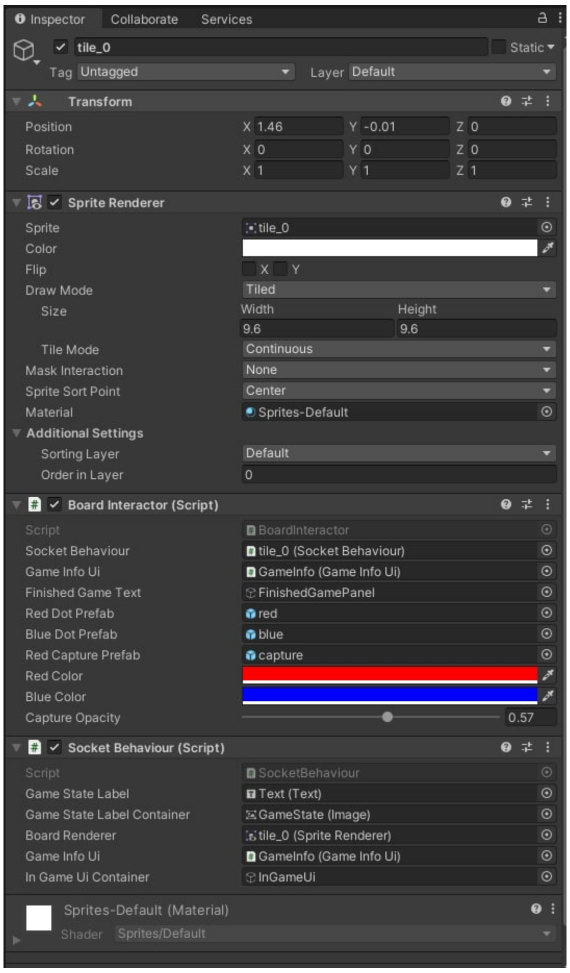
5.1. Unity game engine

Unity is a cross-platform game engine for making 2D and 3D games, animations, immersive environments and more. Unity applications can run on a wide range of platforms including desktop environments like Windows, MacOS, and Linux-based operation systems, as well as running inside a browser and on consoles and mobile phones. C# is used as the primary scripting language that is compiled and executed in the Mono cross-platform runtime. Unity also previously had a JavaScript-based scripting language called UnityScript that was since deprecated.

Unity game can have a number of scenes, with one of them being active at a given time. Each scene contains an object hierarchy, where each game object contains a number of components that influence its behaviour (Figure 10). Unity provides a number of built-in components.

Transform – is one of the core components that defines game object's position on the scene, rotation, and scale, as well as defining the object hierarchy. Game objects inherit position, rotation, and scale from their parent. Which means that effectively each game object also defines its local coordinate system.

Other types of components include mesh and sprite renderers for 2D and 3D games, physics colliders, particle effects, sounds emitters, etc. Unity also has a game UI system that works on the same principles.



```
tile_0                                   Static
Tag Untagged               Layer Default

Transform
Position        X 1.46    Y -0.01   Z 0
Rotation        X 0       Y 0       Z 0
Scale           X 1       Y 1       Z 1

Sprite Renderer
Sprite                    tile_0
Color
Flip                      X  Y
Draw Mode                 Tiled
  Size                    Width     Height
                          9.6       9.6
  Tile Mode               Continuous
Mask Interaction          None
Sprite Sort Point         Center
Material                  Sprites-Default
Additional Settings
  Sorting Layer           Default
  Order in Layer          0

Board Interactor (Script)
Script                    BoardInteractor
Socket Behaviour          tile_0 (Socket Behaviour)
Game Info Ui              GameInfo (Game Info Ui)
Finished Game Text        FinishedGamePanel
Red Dot Prefab            red
Blue Dot Prefab           blue
Red Capture Prefab        capture
Red Color
Blue Color
Capture Opacity           0.57

Socket Behaviour (Script)
Script                    SocketBehaviour
Game State Label          Text (Text)
Game State Label Container GameState (Image)
Board Renderer            tile_0 (Sprite Renderer)
Game Info Ui              GameInfo (Game Info Ui)
In Game Ui Container      InGameUi

Sprites-Default (Material)
Shader  Sprites/Default
```

Figure 10. Example of a game object with attached components.

Developers can create their own custom components to add game logic. Each custom component is a C# class that extends MonoBehaviour class. Component defines a number of fields that are accessible from the game-editor and overrides hook methods that allow to run custom logic on game updates.

Usually, games re-use the same configurations of game objects and components across many instances: game might spawn multiple enemies of the same type. Or place the same asset (like a door in a building) multiple times in the game world. Unity allows developers to save pre-created configurations in the form of prefabs. Prefabs are pre-configured hierarchies of one or more game-objects with their components and properties saved to a file. Those prefabs then can be simply instantiated during the game.

Unity also recently announced a new ECS (Entity Component System) architecture that aims to drastically increase performance for games with large amounts of entities by increasing data-locality and opportunities for multi-core computation. Since those performance concerns are not important in the context of this work, ECS was not used here.

5.2. Main menu scene

The game is split between two scenes: the main menu, and the game scene. The main menu (Figure 11) scene loads first and allows the player to enter their name and start the matchmaking process. It hosts a text input element for the player's name and two buttons: one to play the game, and a second to exit the game.

Scene logic is governed by the "MainMenu" script. It is attached to the primary UI canvas. It binds the event handlers for the buttons and the input field. When the user clicks the "Play" button it will load the game scene.

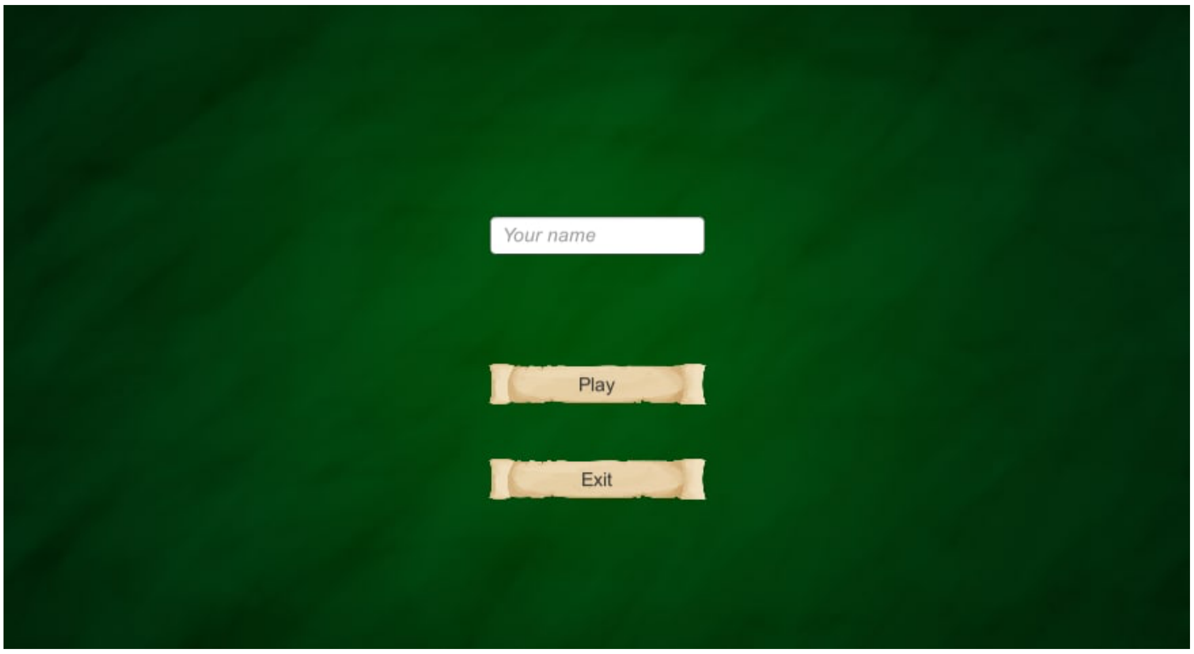


Figure 11. Screenshot of the main menu.

One tricky aspect of having multiple scenes is passing data, namely player's name in our case, between scenes. Unity doesn't provide a straightforward way to do this. The common solution for this is to utilize the fact that static properties on classes in C# persist between scene changes. “StateManager” class hosts a number of static properties. One of them is the string recording the player’s name that will be later used by the game scene.

5.3.Game scene

Game scene (Figure 12) is a lot more complicated than the main menu. It hosts the main functionality of the game as well as networking components. The following elements are present on the scene:

- Main game panel – located in the centre.
- The players panel – located in the top-left corner. It displays the name of the players in the game, their score, and also displays the player currently making the turn, by highlighting their name in bold.
- Finish game button – located in the bottom-left corner. Player can press it to signal that they do not wish to make any more moves.
- Status text – displayed at the centre of the screen when there’s a message to show. Messages include loading indicators, matchmaking status, or the game-over message.

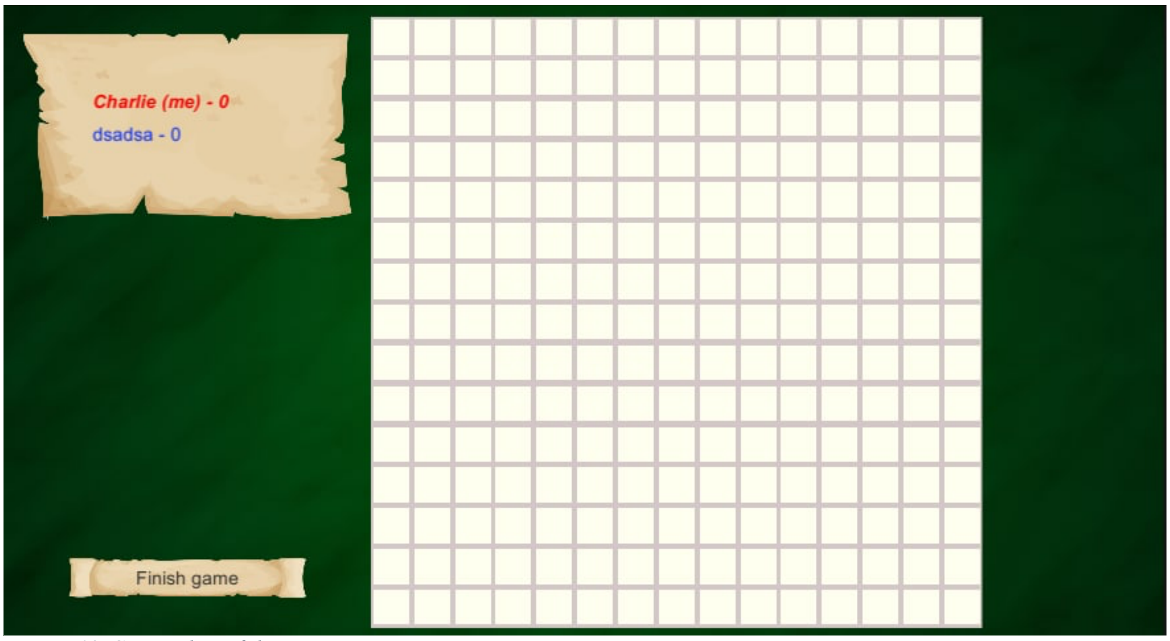


Figure 12. Screenshot of the game scene.

Depending on the current game state only a subset of elements might be present. During the loading screens only the status message is shown. When the game is ongoing the board and associated UI elements get rendered. This scene’s logic is split between three scripts:

- “BoardInteractor” – responsible for rendering of the game board and also handles user input. This behaviour stores the current state of the game. It handles moves by optimistically applying local updates and forwarding the moves to the server.
- “SockerBehaviour” – bootstraps and handles the lifecycle of the server web-socket connection. It accepts moves from “BoardInteractor” and sends them as RPC calls to the server. This script also accepts updates from the server and governs the current state of the game as described further in chapter 6.1.
- “GameInfoUi” – encapsulates UI components that are visible during the game. This script holds references to the required UI components and provides convenience methods to apply updates without getting into details of UI composition.

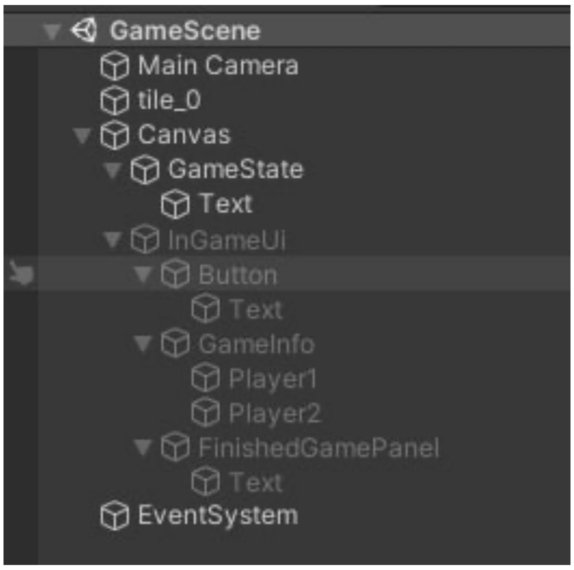


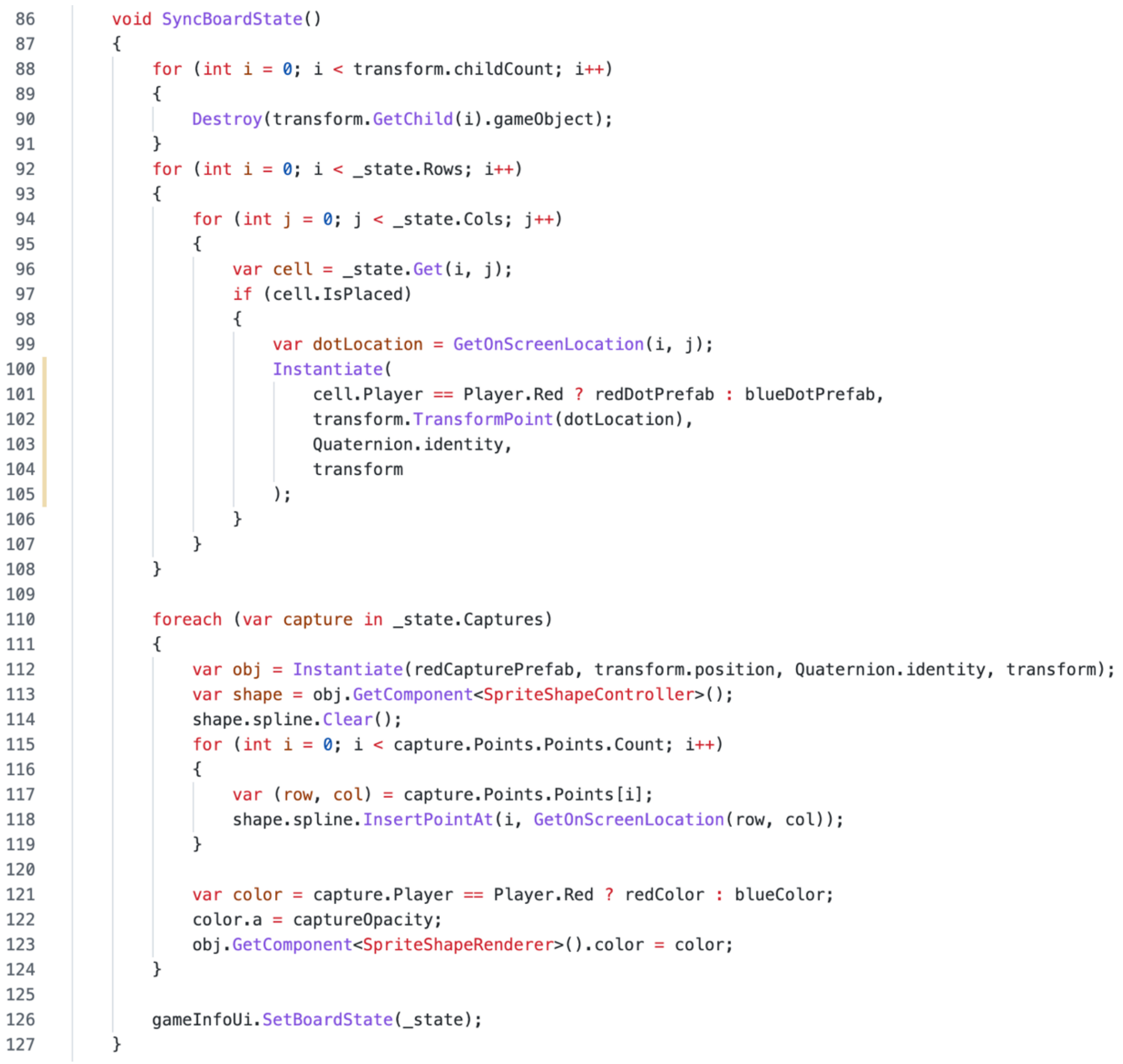
Figure 13. Game object structure.

Game-object structure, which is illustrated on the Figure 13, also supports this separation. The UI components are located under the Canvas object. Subtrees of UI can be enabled or disabled depending on the game state. The “GameInfoUI” component is attached to the “GameInfo” panel. “BoardInteractor” and “SocketBehaviour” are attached to the game board (in this case named “tile_0”).

5.4.Board rendering

Figure 14. Tile sprite.

Board rendering is governed by the “BoardInteractor” script. The board background consists of a sprite with repeated tile texture (Figure 14). Setting Wrap Model to Repeat in the texture settings causes the image to be tiled on top of the sprite game object, when the dimensions of the sprite are bigger than dimensions of the texture. Sizes are measured carefully so that the texture repeats exactly 26 times, drawing a grid.



```
void SyncBoardState()
{
    for (int i = 0; i < transform.childCount; i++)
    {
        Destroy(transform.GetChild(i).gameObject);
    }
    for (int i = 0; i < _state.Rows; i++)
    {
        for (int j = 0; j < _state.Cols; j++)
        {
            var cell = _state.Get(i, j);
            if (cell.IsPlaced)
            {
                var dotLocation = GetOnScreenLocation(i, j);
                Instantiate(
                    cell.Player == Player.Red ? redDotPrefab : blueDotPrefab,
                    transform.TransformPoint(dotLocation),
                    Quaternion.identity,
                    transform
                );
            }
        }
    }

    foreach (var capture in _state.Captures)
    {
        var obj = Instantiate(redCapturePrefab, transform.position, Quaternion.identity, transform);
        var shape = obj.GetComponent<SpriteShapeController>();
        shape.spline.Clear();
        for (int i = 0; i < capture.Points.Points.Count; i++)
        {
            var (row, col) = capture.Points.Points[i];
            shape.spline.InsertPointAt(i, GetOnScreenLocation(row, col));
        }

        var color = capture.Player == Player.Red ? redColor : blueColor;
        color.a = captureOpacity;
        obj.GetComponent<SpriteShapeRenderer>().color = color;
    }

    gameInfoUi.SetBoardState(_state);
}
```

Figure 15. Procedure to update the rendered board state.

On top of the rendered grid a number of game objects are located to represent in-game entities. There are prefabs for dots placed by players as well as for the rendering of the captured areas. “SyncBoardState” function (Figure 15) in the “BoardInteractor” is responsible for updating those elements in relation to current board state.

The elements that are placed on the board are all child game objects of the root game object that is rendering the board background grid. The same root game object also hosts the “BoardInteractor” component. Having the on-board elements organized like that is extremely useful for tracking them as Unity allows scripts to enumerate children of game-objects. The other benefits from this approach are that the position, rotation, and scale of children are calculated in the reference frame of the parent object, and we also achieve encapsulation to a certain point: the whole board is rooted at a single game object and the implementation details (on-board elements) are tucked away as children.

The procedure for syncing the board state start with deleting all of the game objects from the board, effectively clearing all elements placed on top of the background. This doesn't cause performance issues due to the relatively low amount of child game objects. At worst case its around $25^2 = 625$ but most games will have less. Then, new elements are placed from scratch including replacing most of the ones that were removed in the previous step. This is inefficient but also simplifies the process quite a bit, since it removes the necessity to generate a diff of what has changed since the previous update.

After the objects from the previous update are removed, all red and blue dots are placed on their respective positions. To calculate the in-game position from the board coordinates the following formula is used:

$$p_{game} = \left(p_{board} + \begin{bmatrix}1\\1\end{bmatrix}\right) * CellSize - \frac{1}{2}BoardSize$$

Cell size is set to 0.46 and the board size is dynamically queried at runtime as the size of the sprite-renderer rendering the board.

Rendering of captured areas is a more difficult problem. They can be in a variety of shapes, so it would be impossible to render them using the standard Unity rectangular sprites. The implementation was performed using the SpriteShapeController component. It allows to manually define the sprite geometry as series of vertices that make up the sprite's boundary. The vertex locations are calculated using the same formula as for the dot positions. The captured area is then rendered as solid-colored semi-transparent overlay.

5.5. User input handling

"BoardInteractor" script also handles user clicks on the board. The script checks if the left mouse button has been presed in the current frame. The check is performed in the script's update callback. If the user performed a click inside the board's bounds, the clicks coordinates in the game's grid are calculated using the following formula:

$$p_{board} = Round\left(\frac{p_{game} + \frac{1}{2}BoardSize}{CellSize}\right) - \begin{bmatrix}1\\1\end{bmatrix}$$

$BoardSize$ and $BoardSize$ are the same as in the previous section. p_{game} are the coordinates of the mouse in the coordinate system of the board. They are calculated using "Camera.main.ScreenToWorldPoint" and „transform.InverseTransformPoint„ from Unity's standard library. After the coordinates for the move are calculated the board state is updated, and the move RPC is sent to the game server. This flow is described in further detail in the next chapter.

6. Client-server interaction

6.1. Overview

Networking model is built around client-server architecture with JSON RPC protocol over a WebSocket transport. This allows for persistent sessions to be established with two-way asynchronous communication between game-client and server.

Instances communicate establishing a connection with an associated session and issuing Remote Procedure Call (RPC) requests (9). Each request has an associated method and payload, which are forwarded to the handler which processes the request. Both client and server expose a set of supported RPC methods with respective request handlers. Handler can respond with an optional response to the request.

It is important to note that while there are distinct server and client roles, they are only used in the context of establishing the connection: client dials the server and originates a new session. After the session is established, both client and server can issue RPC requests to the other party.

6.2. Unity multiplayer networking solutions comparison

Unity offers different existing frameworks aimed at helping developers in networking code implementation for they multiplayer games.

Netcode is the latest official framework released by the Unity team. The 1.0 pre-release version was posted in Q2 2021. The framework works on top of ECS and allows for syncing of subsets of game objects from the server to the client. Client then can issue updates via an integrated RPC mechanism. There's an option for a headless server build, which would act as dedicated server, running the game simulation, but not perform any rendering, saving on resources.

There are many other official and unofficial networking implementations available (10). The notable ones are the now deprecated UNET which was an inspiration for unofficial forks such as Mirror and NLAPI. UNET was official release by the Unity team which provided a similar high-level game object syncing mechanics, but for component-based games.

The described frameworks are general-purpose solutions and are optimized for fast-paced games. This also leaves a heavy performance strain on the backend server which needs to run the whole simulation for the game and store a subtree of game objects in-memory. The Unity engine binary also needs to be loaded into memory for each game session.

For a turn-based game, where the core game logic can be extracted into engine-independent code, such backend implementation is simply unnecessary. A custom networking implementation allows for much leaner backend server, that can handle many concurrent games simultaneously.

6.3. Connection states

During the lifecycle of the connection the client goes through a number of states as depicted by the diagram on the Figure 16.

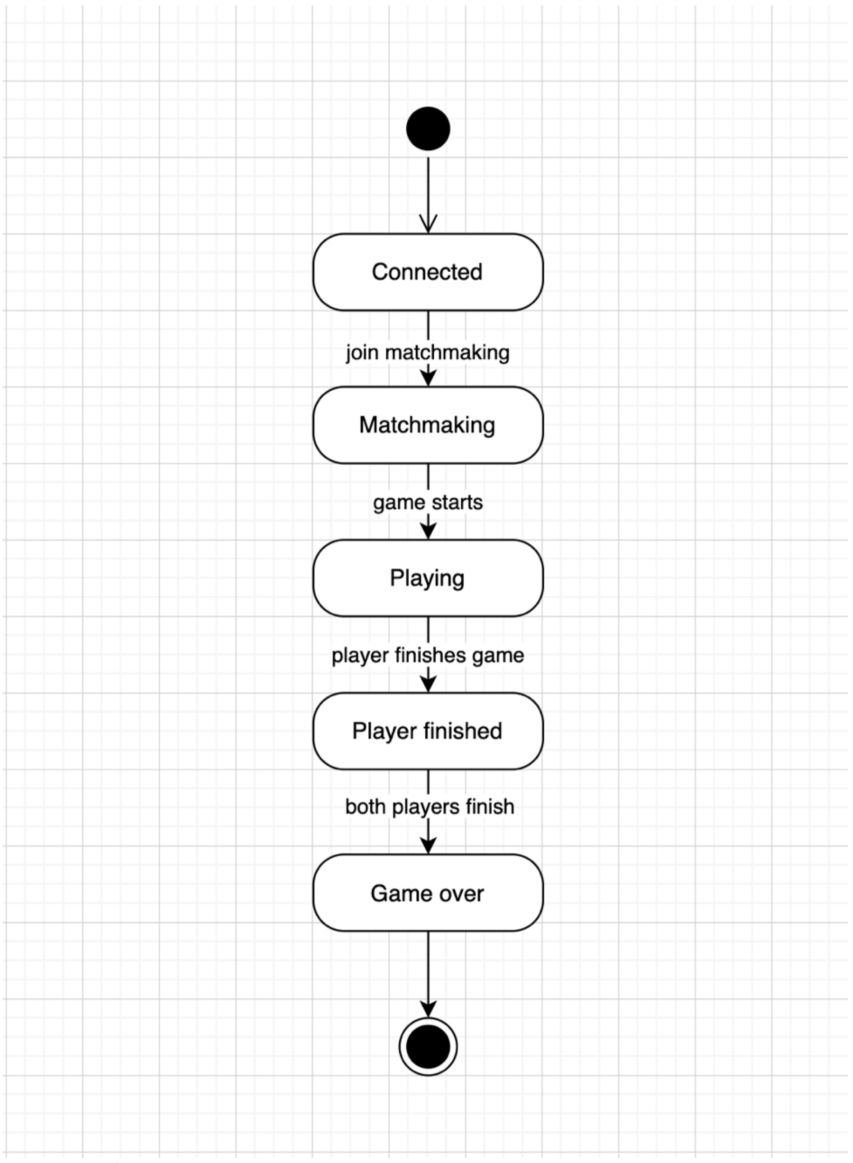


Figure 16. Client state diagram.

6.4. WebSocket protocol

WebSocket is a communication protocol allowing for bi-directional messaging. They are located on the layer 7 of the OSI model (Figure 17) and depend on TCP at layer 4. Websocket endpoints are usually placed on ports 80 and 443, which are the same ports as for HTTP and HTTPS protocols. The protocols are related and usually the same infrastructure and proxies that are used for HTTP can also be used for WebSocket. The specification also defines “ws” and “wss” as two URI schemes to be used for unencrypted and encrypted connections respectively.

WebSockets are different from HTTP by not requiring the server to be first requested by the client and allowing it to originate messages. This is important for games because updates can be pushed from the server to the client at any time.

Other significant advantage of WebSockets is a message framing. In contrast to stream-based protocols like TCP, messages in WebSockets have concrete size and are guaranteed to arrive in the same way. They won’t be chunked into smaller messages, and no extra data will be present before or after the frame. This simplifies the application logic specifically in the context of RPC calls as they work based on messages and not streams.

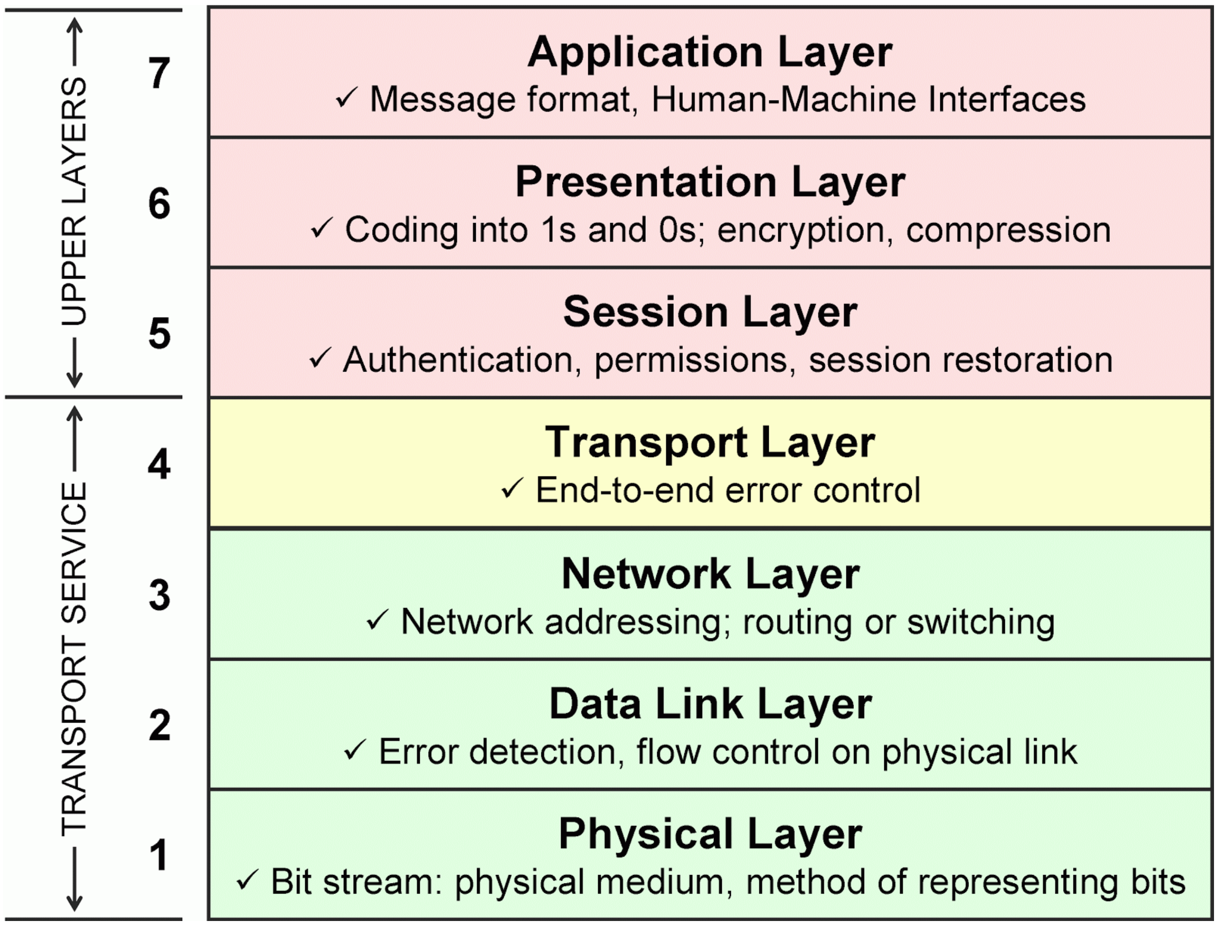

UPPER LAYERS
TRANSPORT SERVICE
7
Application Layer
✓ Message format, Human-Machine Interfaces
6
Presentation Layer
✓ Coding into 1s and 0s; encryption, compression
5
Session Layer
✓ Authentication, permissions, session restoration
4
Transport Layer
✓ End-to-end error control
3
Network Layer
✓ Network addressing; routing or switching
2
Data Link Layer
✓ Error detection, flow control on physical link
1
Physical Layer
✓ Bit stream: physical medium, method of representing bits

Figure 17. The OSI model.

6.5.JSON-RPC

JSON-RPC is a popular, lightweight protocol for implementing remote procedure calls. JSON-RPC spec is available at https://www.jsonrpc.org/specification. Version 2.0 is used here.

Each message in JSON-RPC is a JSON [https://www.json.org/json-en.html] encoded object. Request and response format is defined by the protocol. Each request must have the following fields:

- "jsonrpc" – specifies the protocol version. Must always be "2.0"
- "id" – string or numeric request id. Must be unique among other requests. Responses to this request will have the same id, so they can be routed properly.
- "method" – string containing the method name to be called. Method names are application specific.
- "params" – optional field containing parameters for the method invocation encoded in JSON.

The server may respond with either a success response or an error. In any case, the response must contain the same "jsonrpc" and "id" fields. Additionally, for successful responses a "result" field will be present containing the result of the message invocation encoded in JSON. For error responses there will be an "error" field containing the error message. Here's an example of a method invocation in JSON-RPC:

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}
```

The protocol is implemented in „JsonRpc" class. Since both client and server can originate and handle requests the implementation is the same for them. Each one has an instance of the class. Upon creation, a callback is passed to the constructor which allows the class to send a string via the transport socket. Likewise, the incoming messages must be forwarded to the "JsonRpc" class using "HandleMessageFromTransport" method. The „JsonRpc" class itself is transport-agnostic, requiring only that the underlying transport be reliable and delimit messages somehow. Meaning that messages sent using the callback must arrive at the other side without any partitioning or changes.

Since the "JsonRpc" class doesn't define any specific RPC methods that are available and only takes care of routing and serialization, the code making use of this class must define its own handlers for RPC methods. "`Handle<TArg, TRes>(string method, Func<TArg, TRes> handler)`" is the method used to assign a handler for a certain RPC method. When a request with the matching method identifier comes in, the provided handler will be used to process the request. The request parameters would be deserialized. The return value of that method will be serialized to JSON and sent as a response. Parameter and response types are specified using C#'s generics feature. The response message would be sent with the same ID as the request. "JsonRpc" also doesn't check request IDs for collisions, the client is responsible for that.

The code that originates requests is a bit more complex. The algorithm is as follows:

1. Generate a unique id for the request. Each RPC client has an internal counter that starts from 0 and generates sequential ids.
2. Serialize call arguments to JSON. Newtonsoft.Json library is used for that.
3. Construct the request message. This includes request id, method name, and serialized arguments.
4. Create and register the TaskCompletionSource. It will allow us to wait for the response to arrive. This will be described in more detail in the next paragraph.
5. Send the payload constructed in step 3.
6. Wait for the task created in step 4 to be finished, it will wait for the server's response and resolve with the received message.
7. Remove the TaskCompletionSource registered in step 4 to prevent memory leak.
8. If the response was an error, construct the exception object and throw it.
9. If response is success, decode the result using the same Newtownsof.Json library and return it.

The method will block while waiting for the server to respond. To prevent blocking an OS thread which might stop game logic from executing, we leverage C#'s Task asynchronous programming logic (TAP) (11) alongside with async-await syntax. This allows us to make use of so-called "green threads" which logically look like separate execution threads, but don't result in separate OS threads being created. They are also a lot more lightweight than OS threads which allows us to use many of them without performance concerns. They implement cooperative multitasking model rather than preemtive one. This means that the current green thread will yield the control back to the scheduler to execute other tasks. In this case it happens when "await" keyword is evaluated, as show on Figure 18. Scheduler will then wake up the task to continue its execution when the RPC response arrives.

```
public async Task<TRes> Call<TArg, TRes>(string method, TArg arg)
{
    var id = this._nextId++;

    var responseTask = new TaskCompletionSource<JToken>();
    this._requests[id] = responseTask;

    var request = new
    {
        jsonrpc = "2.0",
        method = method,
        id = id,
        @params = arg,
    };
    this._send(JsonConvert.SerializeObject(request));

    var response = await responseTask.Task;

    this._requests.Remove(id);

    if (response["error"] != null)
    {
        throw new Exception(response["error"].Value<string>());
    }

    return response["result"].ToObject<TRes>();
}
```

Figure 18. Method to originate an RPC call.

To integrate request origination and handling functionality a message router must be present. It determines whether a message is an incoming request or a response to a one that was sent. If message has a “method” field, it is an incoming request. In this case a handler for that method is retrieved from the set of the ones that were registered. The handler is executed. And a response message is sent with the same ID.

If the incoming message doesn’t have a “method” field it is considered a response for a request. In this case a TaskCompletionSource for the specified request ID is retrieved, and the green thread waiting for the method response is woken up and forwarded the response message. Figure 19 contains the code for the router.
There are number of improvements that could be made to the current implementation. First and foremost the client does not handle calls that don’t return values. This would correspond to “void” return type in C#. Same goes for methods without parameters. There’s a workaround for this problem: “object” type can be specified and “null” value can be returned

or passed as an argument in such cases. Adding support for this would require a special case in the code. But since the workaround is straightforward the issue is not high priority.

```
public void HandleMessageFromTransport(string msgStr)
{
    var msg = JObject.Parse(msgStr);
    if (msg["method"] != null)
    {

        var method = msg["method"].Value<string>();

        var handler = this._handlers[method];

        var res = handler(msg["params"]);

        var response = new
        {
            jsonrpc = "2.0",
            id = msg["id"].Value<int>(),
            result = res,
        };

        this._send(JsonConvert.SerializeObject(response));
    }
    else
    {
        var id = msg["id"].Value<int>();
        this._requests[id].SetResult(msg);
    }
}
```

Figure 19. Router for incoming messages.

Second improvement can be made to a way how APIs are defined. Currently the server calls “rpc.Handle” with a string identifying the method name and passes the handler function. It is then the client’s responsibility to correctly call that method by name as well as match the parameter and return types. Incorrectly specified types would not be checked during compilation and would result in a runtime error. This means that there’s no way to formally specify the API in code that would be statically checked by the compiler.

The problem can be solved by users having to declare C# interface that describes the available API methods. Then, using reflection, the server could enumerate the interface’s methods, and extract their parameter and return types. Likewise, on a client the RPC implementation could dynamically create an interface instance using the code generation instruments provided by the .NET standard library. The aforementioned interface could be shared between the client and the server and act as a contract between them to keep the API consistent.

6.6. Backend server

Backend server is written in C# using the .Net Core runtime. The choice of C# as backend language offers a significant advantage by enabling code-sharing with the game code. All of the data that is shared over the network is defined in a single module which helps to keep the client-server API consistent. Using .Net core simplifies hosting as it can run natively on Linux-based systems, which are by-far the most common in hosting platforms.

Upon start the backend server starts creates a game-manager instance which tracks all current games as well as the matchmaking queue. The server also starts listening for incoming WebSocket connections on a given port. The port can be configured through an environment variable.

The WebSocket (12) protocol implementation is provided by the "NetCoreServer" library (13). Mainly, it exposes two classes: WsServer and WsSession. The former is used to spawn a WebSocket server that listens from incoming connections on a given port and then creates a session object. The WsSession is used to represent that session. The connection state as well as packet decoding is handled by the library. The two classes can be inherited from to add domain logic that handles incoming messages and can send outgoing ones.

The following (14) methods are available on the WsSession class that are important for the implementation:

- OnWsReceived(byte[] buffer, long offset, long size) – to be overridden by the subclass. Allows to capture incoming data packages.
- OnWsConnected(HttpRequest request) – called when the connection is esablished.
- OnWsDisconnected() – called when the connection is dropped.
- OnError(SocketError error) – called when there's an error in the connection.
- SendText(stiring msg) – Can be called to send a data packet.

The core of the networking implementation is located in the SocketServer and SocketSession classes. The SocketServer extends the WsServer class which listens on a given port and then spawns a GameSession instance. GameSession instances represent a single incoming connection and therefore a single game client that is connected. They are identified by a GUID (14) stored in the Id prop.

GameSession instances also set up a JSON-RPC peer that can handle incoming requests from the client and also originate server-to-client calls. A pair of handlers is set-up to decode and forward binary messages coming from the web-socket to the JSON-RPC peer, and another one to write the response messages into the web-socket channel. A series of handlers is set-up to handle the calls coming from the client. They are described more in detail in section 6.5.

The state of the games is handled in ServerState and GameState classes. The latter represents a state of a single ongoing game as well as the information about players involved. The former hosts the collection of games as well as the waiting queue for players. The separation of those classes from SocketServer and SocketSession was deliberate to separate core domain logic from networking aspects. This makes it so that the domain logic can be tested separately without potentially complex integration with web-socket library in tests which is undesirable. It also makes so that SocketServer and SocketSession are mainly responsible for integration domain classes, web-socket networking and JSON-RPC together, which limits their complexity.

6.7.Client-Server API

Both server and client register a number of JSON-RPC methods with corresponding handlers. On the server side they are registered in the "PlayerSession" class. The following methods are provided by the server for the client to call:

- "JoinMatchmaking" – Request by the client to put in the matchmaking queue. Has a single parameter of a type string – that player's display name. It will be displayed to the opponent when the game starts. Example call is: "JoinMatchmaking("Bob")".
- "MakeMove" – Sent by the client to make a move during a live game. Client issues this command in response to the player clicking on a board. Has a single parameter of type "Move". This and other data transfer objects (DTOs) will be described lower in detail.
- "FinishGame" – Sent by the client when the player clicks the "Finish game" button. This means that the player can no longer issue moves. This method has no parameters.

Additionally, the client provides a number of methods callable from the server. They are defined in "ServerConnection" class. The following methods are provided:

- "UpdateClientState" – Issued to notify the client that its state has changed. Has a single parameter of type "ClientState" that will be described lower.

- "UpdateBoardState" – Called when there's an active game the client is participating in to notify that the state of the board has changed. Has a single parameter of type "BoardState" that was described in section 4.1.

The "Move" class defines a data structure to represent a single move. It has the following fields:

- "Player" – of type Player (enum). Either Red or Blue player that made the move.
- "Row" – the row index of the placed unit.
- "Col" – the column index of the placed unit.

The "ClientState" class defines a data structure that represents a state of a particular client as seen by the server. It has the following fields:

- "State" – enumeration of type "StateEnum". Represents the current state of the client as well as defines what fields will be present in "ClientState" object.
- "Player1Name" – string, name of the first player. Only present when State is "Playing".
- "Player2Name" – string, name of the second player. Only present when State is "Playing".
- "Winner" – type "Player" (enum). Describes the winner of the finished game. Only present when State is "GameOver".

The "StateEnum" can have following values:

- "None" – initial state when client has just connected to the server.
- "Matchmaking" – set when the client enters the mathmaking queue.
- "Playing" – set when there's an active game.
- "GameOver" – set after the game has completed.

6.8.Connection flow

The message flow from connection to the game start is shown on the Figure 20. In the scenario below the red player connects first, joins matchmaking and then waits for the blue player. When blue player joins, the game commences.

Upon connecting to the websocket endpoint, the red player receives "ClientStateUpdate" message with the state "None", which is the initial state for the newly connected as shown on the Figure 20. After connection, the client requests to join the matchmaking queue and also provides the display name of the player, which was previously entered by the user. The server updates the state and marks the player as waiting for the match. Corresponding “UpdateClientState” message is also sent with “Matchmaking” state. At this point a message is shown to the player that a server is looking for an opponent.

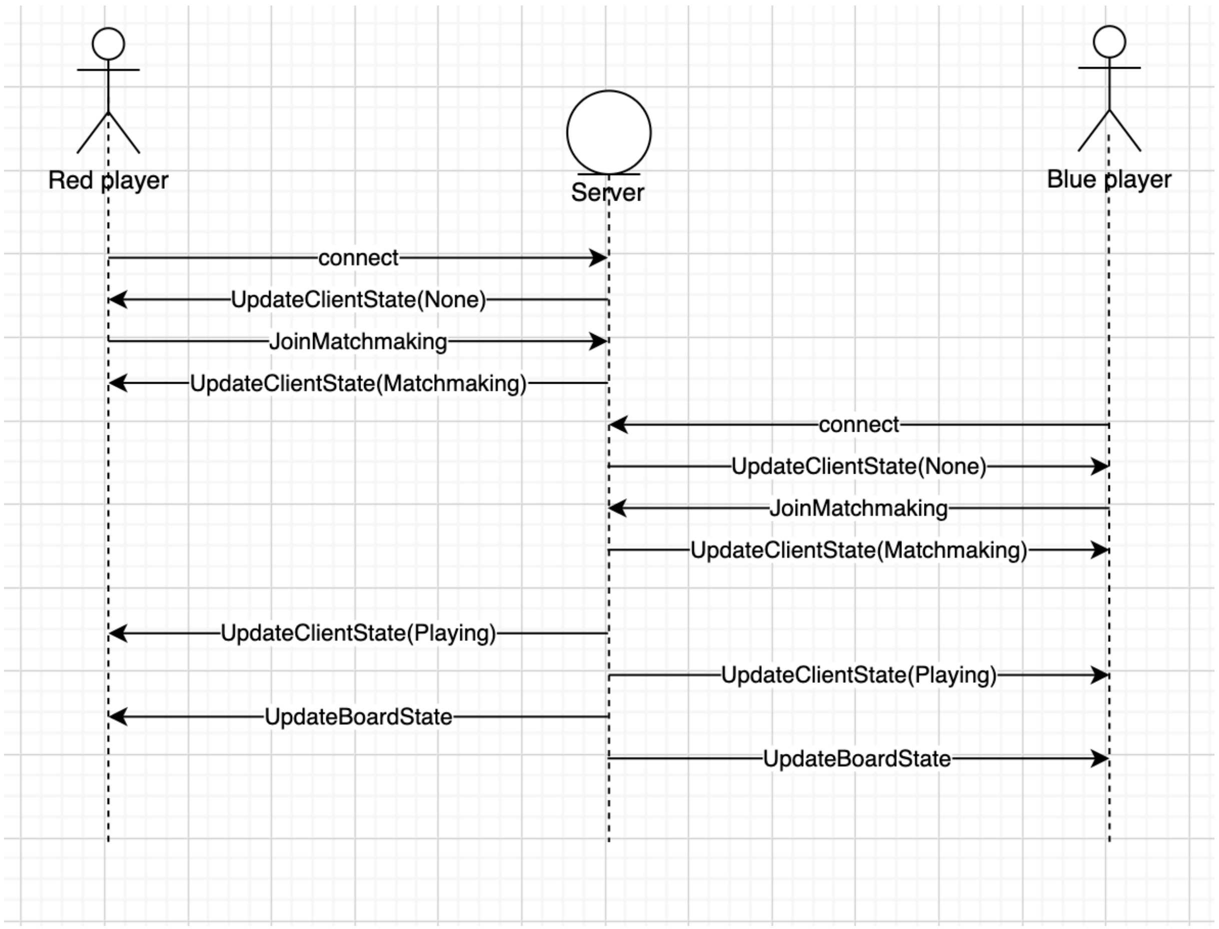


Figure 20. Connection and matchmaking sequence diagram.

Once the blue player goes through the same flow and joins matchmaking, the server can pair them together. A game entity is created on the server, linking both of the players. Players are also taken out of the matchmaking queue. Both players will receive “UpdateClientState” message with “Playing” state. This signals the game client to display the game board.

Server will also bootstrap an empty board state with initial configuration. Following that, “UpdateBoardState” message will be sent containing the full board state serialized in the JSON format. The clients will deserialize the message and update their local state correspondingly. Because the state itself is not particularly large, we can serialize it and send it over in a single message. This simplifies the code as well as there no need to deal with partial state updates: client just receives a new state from the server and replaces the previous one with it.

At this point both of the players are ready to play the game, the process of which is described, in-detail in the next section.

6.9. Game flow

Once the initial connection flow has finished, players can begin playing the game. The full sequence diagram of the flow is available on the Figure 21.

The main portion is players consecutively making moves on their turn. Red player goes first. As soon as the player clicks on the screen to make a move, an optimistic game update happens. Each player has a local clone of the game state, as well as the means to update it. This is important because if we'd delegate the update functionality entirely to the server, players would be exposed to network latency. That is, there would be a noticeable delay

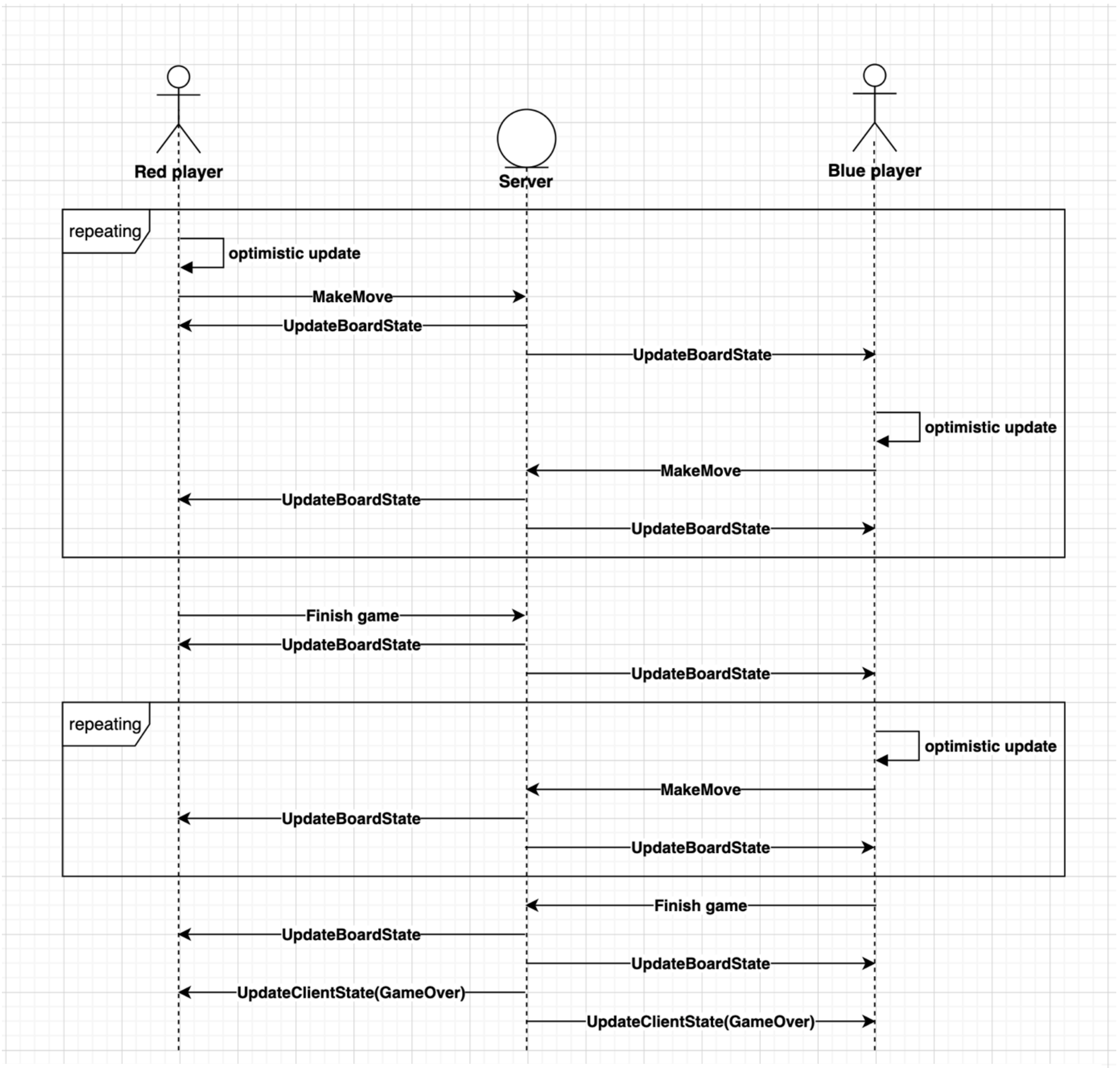


Figure 21. Game sequence diagram.

between player input and the events being reflected on the screen. This is solved by the game client making a local update first optimistically.

In general this can be tricky in fast-paced games as the client has to guess how other players and entities on the scene would react, and then also reconcile the local guessed state with the update coming from the server. Achieving this in a way that doesn't adversely impact the gameplay can be difficult. Luckily, this problem becomes easy in a turn-based game. Since only one player can move at a time, the client knows that any changes to the game state can only occur because of its own actions. Provided that the client follows the same state update algorithm as the server, it should arrive at the same replica of the state, as the server processing the update.

The client will serialize the move and send it over to the server in the "MakeMove" call. Using the serialized move, the server will update its own local copy of the game state and then notify both players that the state has updated via "UpdateBoardState" call. Clients assume that the server is the source-of-truth for the state and will prefer the game state from the server over their own. Effectively, upon receiving "UpdateBoardState" call, clients will deserialize the state and use it as the new current state.

Red and Blue players alternate making moves until one of the players decides that no more moves can improve their position. That player (let's assume this is the Red player in this example) will press the "Finish game" button, firing the "FinishGame" RPC call. This will set a flag in the state and now it will be the other player's turn to do any number on consecutive moves to finish the game. The moves are made in the same manner as during the normal gameplay, with the exception that players don't alternate making moves. After the Blue player is done, they also can click the "Finish game" button. When both players finish, a final state update is sent out that will be used to display the score. After that, an "UpdateClientState" RPC is sent setting the state to "GameOver". This signals the client to display the game over screen showing the winner and the score.

7. Conclusion

As a result of this work, a turn-based game was created in the Unity engine with multiplayer support. A custom networking protocol was designed and engineered as an alternative to existing solutions. The implementation spanned from low level transport handling and session management to high level game logic. An abstraction was built around components not specific to the game displayed in this work. This allows them to be easily reused in different games if necessary.

The network protocol is based on WebSockets – one of the only real-time transport methods available on desktop platforms, phones, and web-browsers. Since WebSockets only provide a way to send and receive binary messages, an RPC and serialization components were developed. The resulting system, a WebSocket transport coupled with the RPC mechanism, provides simple, yet powerful abstraction for implementing multiplayer in video games.

The network protocol implemented in this work is best used for slower paced turn-based games. Since it doesn't integrate with the game-engine the way other solutions do, developers would be required to do extra work to connect the networking logic with the game-objects rendered on the screen. At the same time, lack of integration with the game engine comes at a benefit of the protocol being engine-agnostic. Meaning that the backend server can be built independent of Unity. This allows developers to be much more flexible in the backend architecture they choose, as well as allows them to build services that can handle much greater number of concurrent games on a single instance.

A special care was put to make the protocol work in Unity games compiled to WebGL. Even though code for the Unity engine is written in C# for the .NET platform, many practices and libraries that are used for native applications don't work in Unity. One of those is package management, which is normally done using the NuGet package manager. This work shows that it is possible to use the same methods in Unity, although not trivially and one should not expect the same packages that work for native platforms to work in Unity. Finding substitutes for libraries that work in Unity and WebGL was a trial-and-error process. This work successfully tested WebSocket and JSON serialization libraries that work in both .NET Core environment and Unity for desktop (Mono) and WebGL builds.

Code sharing was another difficult aspect. Reusing code between the client and the server drastically decreases the amount of effort needed to develop an application, while eliminating the possibility of client and server logic diverging (since they are running the same code). Unfortunately Unity does not provide a way to be easily integrated into a MSBuild based pipeline that's common for .NET ecosystem. This work shows a possible solution for sharing a common module between Unity game and a tradition MSBuild solution. The method is not ideal since it works by compiling a subset of source files by two unrelated build toolchains. This forces programmer to write code in such a way which is compatible for both toolchains. Sometimes this is impossible and conditional compilation directives may be required.

There are couple of further improvements that could be made to the network protocol. Improving the RPC API as described in section 6.5 would benefit developer productivity as well as eliminate one of the sources for potential bugs. Also, JSON-RPC could be swapped for a different protocol. While it advantage of being simple and easily introspectable because of its human-readable JSON encoding, it also adds a significant overhead. JSON encoding of data is not optimal when optimizing for the size of messages transmitted over the wire. A binary protocol would do a better job in such case.

8. Bibliography

1. Semantic Versioning 2.0.0. [Online] [Cited: 23 January 2022.] https://semver.org/.

2. NuGet package manager. [Online] [Cited: 23 January 2022.] https://www.nuget.org/.

3. Information technology — The JSON data interchange syntax. [Online] [Cited: 23 January 2022.] https://www.iso.org/standard/71616.html.

4. GlitchEnzo. NuGetForUnity. [Online] [Cited: 23 January 2022.] https://github.com/GlitchEnzo/NuGetForUnity.

5. Avi. An Introduction to AWS Lambda for Beginners. [Online] [Cited: 23 January 2022.] https://geekflare.com/aws-lambda-for-beginners/.

6. University, . Union Find Algorithm. [Online] [Cited: 23 January 2022.] https://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf.

7. Levoy, . *Area Flooding Algorithms.* 1982.

8. Microsoft. Iterators. *C# language documentation.* [Online] [Cited: 23 January 2022.] https://docs.microsoft.com/en-us/dotnet/csharp/iterators.

9. Hansen, . DISTRIBUTED PROCESSES: A CONCURRENT PROGRAMMING CONCEPT. [Online] [Cited: 23 January 2022.] http://brinch-hansen.net/papers/1978a.pdf.

10. @F-R-O-S-T-Y, . Unity Networking Frameworks - Feature List. [Online] [Cited: 23 January 2022.] https://docs.google.com/spreadsheets/d/100vNy3grUgLV5M7rIUKUtRqO4cmTewQI8P5uwf-Qb9k/edit#gid=1012362019.

11. Task Asynchronous Programming Model. [Online] [Cited: 23 January 2022.] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/task-asynchronous-programming-model.

12. I. Fette, A. Melnikov. The WebSocket Protocol. [Online] [Cited: 23 January 2022.] https://datatracker.ietf.org/doc/html/rfc6455.

13. etheaven. NETCoreServer. [Online] [Cited: 23 January 2022.] https://github.com/etheaven/NETCoreServer.

14. *What is a GUID?* [Online] [Cited: 23 January 2022.] http://guid.one/guid.