Implementing multiplayer core features for GameMaker Studio 2 using Node.js

Game area instances and Simulation layer

Eetu Aaltonen

Bachelor's thesis, AMK
May 2024
Bachelor of Engineering, Information and Communication Technology

jamk

Description

Aaltonen, Eetu

Implementing multiplayer core features for GameMaker Studio 2 using Node.js - Game area instances and simulation layer

Jyväskylä: Jamk University of Applied Sciences, May 2024, 116 pages.

Information and Communication Technologies. Bachelor's Degree Programme in Information and Communication Technology. Bachelor's thesis.

Permission for open access publication: Yes

Language of publication: English

Abstract

The objective of the thesis was to implement multiplayer core features for an existing game prototype developed on GameMaker Studio 2. The game prototype, which had a single-player mode, was targeted to support multiplayer and networking. Node.js offered a solid JavaScript runtime with technologies and modules that helped to design and build a network topology and a client-server integration to the current game prototype.

The project and development were executed by the author alone. The work utilized common project management conventions and methods, including game designing, software requirements specification, agile development, kanban boards like Trello, version control tools like Git, and continuous testing. The tools and languages used included GameMaker Studio 2 with GML, Node.js written in JavaScript and run on Visual Studio Code, a group of NPM third-party modules, and several popular free project management tools.

As a result of the thesis, an integration between GameMaker Studio 2 and Node.js server was achieved. The implementation utilized network elements and technologies such as UDP communication protocol, a hybrid server model, and a custom protocol layer. With added multiplayer mode support, the game prototype has the potential to scale and include more gameplay elements adapted to networking in future development. It was also proved that GameMaker Studio 2 has potential and tools for multiplayer development with the help of external technologies.

Keywords/tags (subjects)

Game development, Multiplayer, Networking, GameMaker Studio 2, Node.js, Communication protocol, UDP socket

jamk

Kuvailulehti

Aaltonen, Eetu

Implementing multiplayer core features for GameMaker Studio 2 using Node.js - Game area instances and simulation layer

Jyväskylä: Jyväskylän ammattikorkeakoulu, toukokuu 2024, 116 sivua.

Tietojenkäsittely ja tietoliikenne. Insinööri (AMK), tieto- ja viestintätekniikka. Opinnäytetyö AMK.

Julkaisulupa avoimessa verkossa: Kyllä

Julkaisun kieli: Englanti

Tiivistelmä

Opinnäytetyön tarkoituksena oli tuottaa olemassa olevaan GameMaker Studio 2:lla kehitettyyn peliprototyyppiin moninpelimuodon ydinominaisuuksia. Pelin prototyyppi, joka aikaisemmin sisälsi vain yksinpelattavaa sisältöä, oli tähtäimessä jatkojalostaa tukemaan myös moninpelattavuutta ja verkkoviestintää. GameMaker pelimoottorin ohella käytettiin Node.js runtime:a, joka tarjosi hyödyllisiä teknologioita ja moduuleja verkkotopologian sekä palvelin integraation suunnitteluun ja rakentamiseen nykyisen pelin prototyypin päälle.

Projekti toteutettiin itsenäisesti kirjoittavan opiskelijan toimesta. Työssä noudatettiin yleisiä projektinhallinnan menetelmiä ja yleissopimuksia, joihin lukeutui mm. pelisuunnittelua, ohjelmiston vaatimusmäärittelyn laatimista, Git-versionhallinnan hyödyntämistä, ketterien menetelmien mukailemista, kanban-taulun kuten Trellon käyttöä ja jatkuvia testauksen syklejä. Käytettyihin teknologioihin ja alustoihin lukeutui GameMaker Studio 2, Visual Studio Code ja Node.js sekä muutamia NPM:n tarjoamia kolmannen-osapuolen moduuleja. Työn tukena hyödynnettiin tarjolla olevia ilmaisia projektinhallinnan työkaluja.

Opinnäytetyön lopputuloksena syntyi toimiva integraatio GameMaker Studio 2:n ja Node.js:llä rakennetun palvelimen välille. Toteutuksessa hyödynnettiin UDP-verkkoprotokollaa, hybrid-palvelinmallia ja mukautettua protokollatasoa. Pelin prototyypin jatkokehityksen kannalta moninpelattavuuden tuomat ominaisuudet antavat pelille selvää potentiaalia laajentua ja sisältää kattavammin verkkoviestintää hyödyntäviä pelielementtejä. Tutkimuksella myös todistettiin, että GameMaker Studio 2:lla on mahdollista toteuttaa moninpelejä sen tarjoamia työkaluja hyödyntäen, ja ulkoisia teknologioita apuna käyttäen.

Avainsanat (asiasanat)

Pelikehitys, Moninpeli, Verkkoviestintä, GameMaker Studio 2, Node.js, Viestintäprotokolla, UDP socket

Contents

Figures

Tables

Terminology

Term	Description
%localappdata%	Windows environment variable pointing to a specific folder for applica-tions to access
ACK	Acknowledgment
AI	Artificial intelligence
Bit	The smallest unit of data in computer science
broadcast	Transmitting a packet to every host on the network
Buffer	A temporal storage in a memory to store data
Byte	A unit of digital information, contains 8-bit of data
DOM	Document Object Model, programming API for HTML and XML
DS	Data Structure to store values in GameMaker
GML	GameMaker Language
HTTP	Hypertext Transfer Protocol
I/O	Input/Output operation
ID	Identification
IDE	Integrated Development Environment
IP	Internet Protocol
JS	JavaScript
JSON	JavaScript Object Notation
localhost	A loopback address pointing to the current computer
MMORPG	Massively multiplayer online role-playing game
MSS	Maximum segment size
MTU	Maximum transmission unit
MVP	Minimum viable product
NAT	Network address translation
NPM	Package manager for JavaScript
OS	Operating System
OSI	Open systems interconnection
Ping	A measured time to send the smallest possible amount of data between two hosts and receive a response
Snapshot	A framed game state
Socket	Endpoint for sending and receiving data over the network
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User interface
UTF-8	An encoding standard
UUID	Universally unique identifier

1 Introduction

Multiplayer gaming has gained popularity since the 1980s as a socializing game genre previously known as multi-user dungeons. The latter growth of the internet and technological discoveries provided a worldwide platform for multiplayer games to evolve from local-area networking to online. Modern multiplayer games offer a variety of modes, from cooperation games and competitive tournaments to MMORPGs. (Balasubramanian 2022.)

Among numerous available game development tools and platforms, GameMaker Studio 2, with its GML coding language, caught the interest of the thesis author. GameMaker Studio 2 is an easy-to-use and full-featured development tool for newcomers and professionals to create simple or complex 2D games. However, the concept itself requires improvements in multiplayer game development. The lack of published information on building multiplayer games using GameMaker Studio 2 is evident, with only a handful of tutorials on the YoYo Games website discussing networking basics.

The thesis topic was selected based on the author's interests, experience, and knowledge of the platform gained from self-directed learning and hobby projects over five years. The author has developed their latest indie game prototype for the past year. The game has a playable single-player mode that is planned to adapt to networking to support a new multiplayer mode. The research requires seeking a solution outside the GameMaker Studio 2 because networking implemented onto built-in components and peer-to-peer topology is not the only or the most optimal option. The GameMaker Studio 2 has the potential in multiplayer networking, waiting to be revealed and shared publicly.

The thesis aims to seek answers from literature, studies, and the Internet combined with practical game development and applied research. It introduces and compares available communication protocols and network topologies for their benefits in multiplayer networking. The work follows typical game design, including coding with a client-side GameMaker engine and server-side backend. However, the research does not discuss the basics of GML or JavaScript. Instead, it focuses on client-server integration and game elements from a perspective such as game world zoning, instance management, and simulation layer.

2 Multiplayer game development

2.1 TCP and UDP performing on different gameplay flow

2.1.1 What is TCP/IP?

TCP stands for Transport Control Protocol and is an IETF Standards Track transport protocol on the transport layer of the Open Systems Interconnection (OSI) communications model. TCP is a connection-oriented protocol in which two hosts begin exchanging acknowledgment messages on a three-way handshake to initiate a connection. (Fairhurst, Trammell & Kuehlewind 2017, pages 5-6; What is OSI Model? – Layers of OSI Model 2023.)

TCP is in relationship with Internet Protocol (IP), and together, they increase the reliability and accuracy of data transmission over the Internet. The application byte stream is partitioned into TCP segments sent as an IP datagram. While Internet Protocol ensures the packets find their destination address, TCP maintains the persistent connection between two hosts. TCP combines incoming IP datagrams in the correct order and detects packet losses and errors. If one or more packets are lost or the datagram checksum does not match, TCP asks the sender to deliver the corresponding packets again. (Eddy 2022; What is TCP/IP? n.d.)

TCP header extends the IP header with a sequence number, checksum, and acknowledgment number to ensure reliability on the data transport. The sequence number indicates the amount of data sent during the session so far, which can be used to detect duplicate or dropped packets and sort incoming datagrams to the correct order. The checksum is used to verify that the content of the arriving packet is whole and not changed on the path between hosts, and the acknowledgment number holds the expected sequence number of the next incoming packet. (Eddy 2022.)

2.1.2 What is UDP/IP?

User Datagram Protocol (UDP) is a lightweight connectionless transport layer protocol. UDP is less reliable than TCP but is more suitable for time-sensitive data transformation in applications such as video playback, voice calls, live streams, and online video gaming. UDP gains its latency, bandwidth, and memory efficiency by omitting formal handshake, error control, flow control, packet order, and acknowledgment checks from the data transmission process. (Glazer & Madhav 2015, pages 41-42; What is UDP? n.d.)

UDP header is a small 8-byte header that contains a minimum amount of information to transmit data without a shared state between two hosts. It contains a source port, destination port, length of the UDP datagram including the data and header, and optional checksum. (Glazer & Madhav 2015, pages 41-42; User Datagram Protocol (UDP) 2023.)

2.1.3 Considering benefits and costs

In the early stages of multiplayer game development, an important decision needs to be made: Should the networking be built on TCP or UDP? (Glazer & Madhav 2015, pages 207-209). Does TCP offer reliability benefits for this particular multiplayer game, or does it negatively impact the gameplay flow? How much effort will it take to develop a custom reliability layer on top of UDP? To answer these questions, one must survey the nature of the gameplay flow.

Packet priority

Packet priority is usually associated with in-game actions in multiplayer games. Depending on the nature of the game, some actions may be prioritized over another by overwriting or delaying queued low-priority actions with highly prioritized ones.

For instance, there might be scenarios in fast-paced multiplayer games where TCP packets are received in a prioritized order that is less than ideal from the player's perspective. Such issues can occur because TCP relies on the exact send-receive packet ordering. In other cases, a high-priority action is yet to be executed due to a dropped low-priority packet being sent ahead of it. This can happen because the high-priority packet is waiting for a dropped packet to be retransmitted. Such packet delays can significantly impact the gameplay, with game elements updating on a delay or in a non-ideal priority order, potentially disadvantaging a player's chances of winning. (Glazer & Madhav 2015, pages 207-209.)

TCP utilizes retransmission on dropped packets by using their sending order, while UDP only assumes that a remote host successfully receives a high-priority packet as it did with a prior low-priority one. However, neither of them does any packet sorting, leaving the responsibility to developers.

Delay

Delays can be caused by many small factors and accumulate over time. In most cases, they only harm gameplay. For instance, delays in network communication can result from packet drops, a distance between hosts, or transmission processing times.

TCP can cause unpleasant delays even if the game has been coded to prioritize in-game actions equally. If packets happen to arrive in a group and one of them gets dropped, this halts the processing of the other packets. After a short period, the retransmitted data could have lost value and been classified as a stale world state in fast-paced games. During the delay period, the game state might have advanced in time, and objects and players might have already changed their speed, direction, and position or taken another action. (Glazer & Madhav 2015, pages 207-209.)

UDP attempts to solve the problem by offering the freedom to build custom packet management. However, it requires some effort to map and write sufficient test coverage for numerous possible scenarios where packet dropping has benefits rather than unpredictable harm. (Glazer & Madhav 2015, pages 207-209.)

If the Multiplayer game aims to maintain real-time network communication, the drawbacks of dropped packets are sometimes easier to tackle than dealing with an unpleasant delay. On the other hand, turn-based multiplayer games, like card games and tabletops, can benefit more from communication reliability than lower delays.

Reliability

On the other hand, UDP can drive action chains out of track with chaotic results or cause major desync if essential messages do not reach the destination host. Imagine a marketplace transaction where payment is lost on its path to the buyer or an unregistered perfectly landed hit that could have made the player win a match.

Because UDP does not have acknowledgment message exchanging or packet ordering mechanisms, these missing reliability features need to be solved on the higher-layer protocols. A custom reliability protocol is one potential option to ensure the integrity of network communication. However, it might come with increased engineering and testing time costs. (Glazer & Madhav 2015, pages 207-209; TCP and UDP in Transport Layer 2021.)

TCP includes several reliability mechanisms and features out of the box, leaving developers more room to focus and prioritize their resources on other things. TCP is not the only communication protocol option for every multiplayer game, taking into account being non-ideal for fast-phased real-time shooter action games. With a custom protocol layer built on UDP, developers can control and manipulate packet handling and priority, achieve lower transmission times, and eliminate delays in unnecessary retransmissions.

Flow Control

Deficient flow control can lead to increased packet drops in numbers. Technical or physical limits exist to how much data a receiving host can handle and process in a small time window. The data sent rate can vary depending on the number of players in a single server or session, the size of the multiplayer game and its in-game world, and the gameplay flow. The sent rate can vary and peak unexpectedly, posing a challenge if not adequately managed.

If network packets are sent to the endpoint at a significantly high rate and the receiving host is incapable of processing them rapidly enough, packets get dropped. TCP provides an end-to-end flow control, where hosts communicate with acknowledgment messages to keep the send rate within appropriate limits. UDP, on the other hand, lacks such a built-in mechanism. (Glazer & Madhav 2015, pages 207-209; TCP and UDP in Transport Layer 2021.)

Developers are responsible for managing the packet send rates or creating custom controllers if necessary.

Fragmentation

The Maximum Transmission Unit (MTU) defines the maximum size of the payload – which includes multiple headers, payload data, and packet wrappers – carried in a single segment. When a payload size exceeds the MTU, the packet gets fragmented. This mechanism takes effect on a bridge between two connected Link Layers with different MTU limits. Such a phenomenon is common in cases where a large packet enters the Ethernet level with a limitation of 1500 bytes MTU. However, many modern Ethernet network interface cards support MTU sizes of up to 9000 bytes. (Glazer & Madhav 2015, pages 22, 35 and 80; TCP and UDP in Transport Layer 2021.)

With network communication built on UDP, a good practice is to avoid sending a segment size larger than the Ethernet can carry in a single segment. For that reason, developers must decide how network packets are segmented to avoid exceeding the MTU. (Glazer & Madhav 2015, page 80.)

TCP supports a Maximum Segment Size (MSS) and Path MTU discovery. Therefore, these fragmentation steps can more or less be ignored – with MSS set to a small enough value – because TCP is designed to transmit larger data streams than UDP. (Glazer & Madhav 2015, page 42; TCP and UDP in Transport Layer 2021.)

Resource usage

Without precise custom data control and memory management, applications built on UDP may face a high risk of jamming and stalling network traffic. In the worst scenario, they can allocate an unnecessarily large amount of memory. The issue can result from an unrestricted packed send rate and poor data grouping. (Glazer & Madhav 2015, pages 107 and 207-209; TCP and UDP in Transport Layer 2021.)

All connections and outgoing data are tracked on TCP communication. Maintaining such a reliable connection comes with a considerably larger resource allocation cost. Sent packets are stored in memory for the time being until acknowledgments about successful delivery are received. TCP makes it challenging to build custom data tracking and routing because the Operating System manages resource allocation for the network traffic. (Glazer & Madhav 2015, pages 207-209.)

Finally, TCP and UDP headers differ in their structure. UDP is a simple connectionless communication protocol to send data over the Internet, though its header contains minimal information about the packet's target destination for delivery. To ensure that all sent data is received – whole and in the correct order – TCP requires a larger header than UDP. (Glazer & Madhav 2015, pages 207-209; TCP and UDP in Transport Layer 2021.)

TCP shines with its reliability in demanding, secure, and connection-oriented applications. However, in some cases, it can use more bandwidth on data delivery and retransmitting unnecessary packets than communication on flexible and lightweight UDP.

2.2 Network topology

2.2.1 General

The second important decision in multiplayer game development is to solve the problem of how players can communicate with each other. The decision will direct how the network code and networking are implemented.

This concept is called a network topology. It defines how clients can find and communicate with each other, how processing is balanced between client and server, who has the authority to make the final decisions in conflicts, and how to keep every client game state synchronized and up-to-date in a shared virtual world. (An introduction to multiplayer network and server models n.d.; Glazer & Madhav 2015, pages 166.)

Developers must consider which network topology would best fit their multiplayer networking. Influencing factors may include the available hosting budget, target user base, scope, game size and complexity, and security concerns. Therefore, one must understand the benefits and drawbacks of different network topologies.

2.2.2 Dedicated server

In a dedicated server model, one host – a server – is a central point that connects other participants – clients – to play a game together in a shared virtual world. This model follows a client-server architecture, in which clients establish connections and exclusively communicate with the server. A dedicated server is typically a "headless" version of the game instance that does not draw graphics. Instead, it runs game world simulations, sends rules and information that clients must follow accordingly, and ensures that the shared game state is synchronized between endpoints. The model allows running multiple server instances on a single powerful machine. (Client-Server Network: Definition, Advantages, and Disadvantages 2023; Glazer & Madhav 2015, pages 166-168.)

In most scenarios, the server has full authority. In other words, this means that the server-side simulations are considered correct and result in the final game state that clients must replicate. If a client notices a troubling discrepancy with the server's game state, the client must correct its local game state. To execute an action, a client must inform the server about their intentions; the server then decides if the action is processed or forbidden. The result is then broadcast to clients, who update their local game state to match the outcome. (Glazer & Madhav 2015, page 167.)

A dedicated server can be hosted on a separate – "dedicated" – machine or locally on a client side. Several options are available for server hosting on separate hardware: on-premise data centers, cloud platforms, and different hosting service providers. This hosting model is ideal for multiplayer games where dedicated servers are geographically distributed, and the game is made available globally. However, if the closest dedicated hosting point is located geographically far from the client, it can cause unpleasant latency. A high ping always has adverse effects and disadvantages in latency-sensitive games. On the other hand, a separate server protects against cheaters from gaining access to sensitive data or IP addresses. However, hosting costs may vary depending on the hosting platform and service fees and quickly grow greater than initially planned. (An introduction to multiplayer network and server models n.d.; Porting from client-hosted to DGS - Client-hosted vs DGS-hosted 2023.)

In a client-hosted model, the dedicated server process is located on one of the client's machines. This model is less reliable and comes with performance and security issues. Because the game client and server are running locally, the hosting machine must be powerful enough to maintain stable and smooth performance. The user who physically owns the hosting machine can easily access the server terminal and stored data to gain advantages in the game. Client-hosted games may also scale poorly because consumer machines are more likely designed for gaming than concurrent server hosting. Data centers or cloud platforms can instead handle hundreds of connections. (An introduction to multiplayer network and server models n.d.; Porting from client-hosted to DGS - Client-hosted vs DGS-hosted 2023.)

In addition, client-hosted setup usually requires advanced knowledge about router and firewall configurations to allow incoming network traffic and make the server accessible outside the local network. On the other hand, dedicated servers are easy to set up on a cloud hosting platform but may come with high rent costs.

2.2.3 Peer-to-peer

In a peer-to-peer model, all clients are directly connected and communicate with each other. This model does not require a server to determine the final game state and rule everyone. Instead, peers share equal authority and responsibility for running the game logic. Every peer sends actions outward and simulates the local game state based on its own and incoming inputs. The game state must remain consistent between all peers. Therefore, every simulated turn or cycle must yield the same output state on every client to ensure synchronization. (An introduction to multiplayer network and server models n.d.; Glazer & Madhav 2015, pages 168-169.)

Every client must share their IP addresses to establish a connection with the participants while still being able to join the game through a single IP address. In this scenario, groups and games are usually formed via matchmaking services. At first, one peer is a named master who invites other clients and opens a lobby that will be available and listed on a matchmaking service. There, clients are gathered together to start the communication. (Glazer & Madhav 2015, pages 168-169.)

The peer-to-peer network model does not usually require any distributed hosting platforms for the server, making it a relatively cost-efficient way to implement a multiplayer game. However, because clients typically maintain the synchronization within the network by sharing information about in-game actions and changes directly with others, it is not self-explanatory who has the decisive word and authority.

2.2.4 Relay server

A relay server acts like a dedicated server but does not run or simulate any game logic. Instead, it is a public linking point to establish connections and provides low-latency datagram exchange between clients. A relay server is responsible for greeting the joining clients, delivering messages and data that clients send, and maintaining the connection during the game session. This model can solve common connection issues where clients cannot reach each other. A relay server model is suitable for scenarios where one client hosts a game session that other participants may join. Because clients communicate indirectionally and without sharing their IP addresses with other participants, this model offers more security and privacy than the peer-to-peer model. (Relay servers. n.d.)

The relay server is a moderately cost-efficient way to implement a linking server to connect clients to a multiplayer session. It usually requires fewer backend coding and hosting resources than a fully dedicated server. However, because the relay server only cares about transmitting data from a sending client to others, it does not provide any server-side authority or game engine functionalities.

2.3 Game world

2.3.1 Zoning

Game world zoning comes hand-in-hand with object scope and relevancy. In multiplayer games with small game arenas, rooms, or tabletops, there will be no concern about bandwidth and processing time when every object, character, and player is replicated between each participating client. However, when a game world, a map, grows in scale and the game supports tens or hundreds of players, technical limitations must be taken under scope more precisely. (Glazer & Madhav 2015, pages 254-255.)

How relevant is it for a player to know about others moving far on the horizon? Or how relevant is it for a player to be aware of objects behind a wall? On a larger scale, developers must build boundaries and scopes to define how much information should be shared with clients about the surrounding environment and pawns. Otherwise, recklessly sending data around would use an unnecessary number of resources. (Glazer & Madhav 2015, pages 254-255.)

Several approaches are used in multiplayer game development to tackle relevancy issues, from rendering optimizations – like visibility culling – to zone instances.

2.3.2 Static zones

Static zones are divided areas of the game world. They have boundaries and scopes that define which objects and players share replicated information within the zone. Like in beloved MMORPGs, players can travel freely between world locations such as towns and forests. However, it is essential to consider that sending replicated data to a client about a player beyond reach and sight can have a noticeable impact on network traffic. (Glazer & Madhav 2015, pages 255-256.)

In theory, when players are distributed into static zones, it should help balance player groups evenly between zones and decrease the number of relevant objects in a client's surroundings. However, players tend to interact socially, gather in meetings, and do business in central towns. Such behavior makes some static zones gain more popularity than others, making them crowded by players. (Glazer & Madhav 2015, pages 255-256.)

Players commonly gather around points of interest and popular locations. Games that feature guilds and groupings, public events, and racing increase the chance of crowding individual areas. From the game design perspective, it is recommended to scatter game activities around the in-game world and utilize the available space to control players' behavior and movements.

Before a player enters a static zone, in some cases, they are locked into a loading screen between the zone transition. A loading screen provides a time frame for a client and the server to exchange replication data so the client can receive the whole picture of the world state of the destination. Some games have achieved a smooth and seamless zone traversal without such loading screens by object fading. Additionally, if the zone terrain stays static, storing the terrain locally on a client enables the game engine to render distant lands, even over the zone boundaries. (Glazer & Madhav 2015, pages 255-256.)

Depending on the game and setting, delays and continuous zone traversal can significantly impact gameplay, notably in action games. In these games, the intensity of fighting and battles may concentrate in much smaller areas than in MMO games, leaving static zones irrelevant. Additionally, in fast-paced action games, a delay during zone traversal can lead to players lagging behind others. (Glazer & Madhav 2015, pages 255-256.)

Static zones are essential elements in game world-building. They are great framing tools used to draw lines onto the terrain to separate different in-game areas on the world map. Static zones aim to balance players around the available space and environment. However, technical limitations may sometimes make using them impractical or inefficient – fortunately, an alternative way is to cast popular zones and areas into instances.

2.3.3 Instancing

Instancing is usually referenced to multiplayer games with a shared game world. Two players may live in separate instances and stand in the exact location without knowing the other’s existence. Instancing is a technique that can be used for two purposes. The first creates a capsulated version of a game area that lives as an independent story. The second aims to balance the player count into separate instances in overcrowded zones. (Glazer & Madhav 2015, pages 262.)

For example, in popular MMORPGs, dungeons are independent instances separated from the shared game world and designed to be explored by a group of players. They provide single-player-like storytelling and scripted content for smaller groups that raid the dungeon area encapsulated from the outer world. Furthermore, instancing can cover much larger areas and zones of the shared game world. In storytelling, individual areas can be personalized for each player depending on their progress on a specific quest or storyline by creating instanced scenarios. (Glazer & Madhav 2015, pages 262.)

Instancing is a great tool for game designers and scriptwriters to create distributed timelines, scenarios, and eras in the game world. It opens possibilities for storytelling that can be tailored for individual players based on the player's earlier choices and progress in the storyline. With instancing, game developers can create difficulty levels for dungeons to offer simultaneously an entry mode for newcomers and challenges for veterans.

From a technical perspective, instancing can potentially solve performance issues in overcrowded and popular zones. Game zones can have a preset cap to define how many players they can simultaneously hold. When the player count reaches the limit, a second instance of that zone is created, where the following players are then directed. This way, developers can optimize and balance the client and server-side processing without restricting the player movement in the shared game world. (Glazer & Madhav 2015, pages 262.)

However, instancing may cause the player base to feel that they do not fully share the multiplayer experience or are forced to play in compartmentalized worlds. It might also require extra steps – like setting up a group and teleport invitations – for players to meet each other if they happen to end up in different instances, even if all of them are gathered at the exact geometrical location. (Glazer & Madhav 2015, pages 262.)

When properly used, instancing can add new layers and depth to game mechanics and gameplay. It can provide solutions to games to scale and support more simultaneously playing users and help balance server workload. However, instancing comes with drawbacks and challenges when the logic grows in complexity or multilayer. It may require developers to implement an instance management system with controllers and components besides precise testing and memory management.

2.3.4 Simulations

Living on a mesmerizing planet called Earth has its role in the consciousness of the human mind. How one can sense the environment – the day turning into night and the weather shifting from sunny to stormy – fascinates game developers and artists. Video games, fictional or realistic, attempt to immerse the player into the virtual world and channel feelings through the game characters. “World simulation (things like lighting and weather systems) create a concrete sense of place via building a world that moves and breathes all on its own.” (Tremblay 2023, page 86).

Depending on the setting, world simulations allow artists and programmers to create visually stunning effects and a believable game world that is easy to engage. The world can contain dynamic elements integrated into a puzzle or survival scenario where players must take specific actions or change their behavior to adapt and overcome various obstacles. Something that requires knowledge and observation from the players to sense and respond to the dynamic shifts of the game world. (Tremblay 2023, pages 86-88.)

Structurally, the world simulation always follows internal logic and programmed rules. Various tools and methods are used to create world simulation elements. One of them is shaders. Shaders are practical tools for artists and programmers to produce in-game effects by manipulating the rendering behavior of existing pixels. They can be used to cast shadows and adjust lighting and color pallets. A day-night cycle is a typical combination of these elements. (Tremblay 2023, pages 86-88.)

> *A switch in palette is an effective way to indicate a shift in environment. In games, light blues and darker palettes are an obvious indicator of night versus day (particularly in games like Sable where the ability to see and understand the world is changed significantly in the day or nighttime). (Tremblay 2023, page 74.)*

Shaders and dynamic elements can change the visuality of an environment to create a convincing illusion of the passage of time, like a shift from daylight to midnight, which makes the game world more immersive and believable.

The weather system and world events are alternative tools to make the game world dynamic. Ever-changing weather can noticeably affect the atmosphere. The weather can form fog walls, giving players a hiding place, or create untraversable obstacles such as hurricanes and tremendous waves, making adventuring – like sailing at sea – more dangerous. (Tremblay 2023, pages 86-88.)

World events can signal players to avoid certain parts of a map or offer different and rewarding activities to complete. For instance, in Sea of Thieves, world events can indicate an upcoming danger or soothing clear seas. The signals can lure players to raid valuable resources and treasures or fight mighty enemies and bosses in specific areas. (Tremblay 2023, pages 86-88.)

3 GameMaker Studio 2

3.1 General

GameMaker was founded in 1999 by Mark Overmars and introduced as a graphics tool named "Animo" on November 15. Since 2007, it has been developed and supported by a British software development company, YoYo Games (GameMaker 2023; YoYo Games 2023). GameMaker aims to provide a user-friendly and full-featured game development platform with advanced tools for building simple and complex video games. (Ford 2009, pages 1-10; Minor 2022.)

GameMaker Studio is an excellent development environment for newcomers and professionals. It offers a steady landing and adaptive learning curve for beginners to get their first touch with game programming. GameMaker Studio provides powerful and user-friendly tools to create games with 2D and limited 3D graphics. (GameMaker Manual 2023; Minor 2022.)

3.2 IDE and tools

GameMaker Studio features a clean and customizable Integrated Development Environment (IDE) layout with well-organized tools and widgets. The Sprite Editor allows users to draw and edit sprites and images for characters, tile sets, UI, and other game assets. The tool also allows importing images from external editors into a GameMaker project. The Object Editor is a tool for creating templates for different objects and characters. Users can modify various object properties, assign sprites, and program events and logic to control how objects act and behave once they are instantiated. (GameMaker Manual 2023.)

Levels are designed and built with the Room Editor. The editor provides tools to import objects and assets into rooms and modify level properties and behavior. Viewports, cameras, layers, and backgrounds are all customizable. The Sound Editor is used to add and edit sounds. Users can adjust sounds' volume, playback time, sample and bit rates, quality, and compression attributes. Currently, the Sound Editor supports file formats such as WAV, MP3, and OGG. Sounds can be triggered by in-game events or played continuously in the background. (GameMaker Manual 2023.)

Furthermore, GameMaker features tools such as Paths, Sequences, and Shaders for those seeking advanced programming and complexity. The engine also offers different visual effects like particles, filters, and shaders, allowing developers to embellish their games with uniqueness and personality. (GameMaker Manual 2023.)

The IDE features two game programming style options. Users can write code using GML scripting language or GML Visual interface, whether they like programming with drag-and-drop building blocks or traditional coding. However, GameMaker Studio does not exclude the possibility of combining them into the same workflow. It gives developers flexibility and a unique touch to program their game, whichever they feel comfortable and efficient. GameMaker Studio IDE and editor elements are broadly customizable through Preference settings. Users can customize their IDE, including layouts, fonts, theme, colors, and key bindings. (GameMaker Manual 2023.)

GameMaker engine supports several devices and platforms to export a built game. On desktop, games can be published on Steam, Itch.io, GM.games, and HTML5 for Windows, Mac, and Ubuntu. Games can have cross-platform features or be built individually for mobile devices – like Android, IOS, and Amazon Fire – or for the latest consoles, such as Nintendo Switch, PlayStation 5 and 4, Xbox One, and Series X|S. Account licenses and export platforms are categorized into tiers: Free, Creator, Indie, and Enterprise. (Bramble 2023.)

GameMaker Studio also has an online Marketplace where registered users can publish and sell their asset packages. The Marketplace lists plenty of pre-made sprites, shaders, scripts, sound effects, frameworks, and demo projects. These assets can help users get started and focus on something they want to learn the most, allowing publishers to earn money by selling crafted game assets. (GameMaker Manual 2023.)

GameMaker developers and the community have published helpful tutorials, from beginner-level to advanced, on the https://gamemaker.io/en/tutorials.

3.3 Objects and instances

Objects are versatile assets and templates for in-game elements. Characters, walls, and enemies are instances – copies of objects – that appear inside a game level. Users can design and program instances via object assets. These assets control how inherited instances are drawn, how instances move and scale, and how they interact with other instances. If objects are associated with sprites, their instances can be rendered on the screen with different animations and movements. Controllers are types of objects that act behind the scenes. They are usually left out of the rendering pipeline by unsetting the sprite property. Controllers' typical roles include controlling other game elements and timers or triggering different events. (GameMaker Manual 2023.)

GameMaker engine identifies objects by unique names. Objects also have modifiable properties that directly impact their instances. Some properties make instances invisible, while others expose them to collisions or define their gravitational and physical features. (GameMaker Manual 2023.)

Instances inherited from "solid" objects follow built-in physics, ensuring they don't overlap with other solid instances. When two solid instances collide, the engine automatically reverts them to their previous positions, keeping them apart. A collision mask is associated with collision events and bound, by default, to an object's sprite. The collision mask is always based on an existing sprite image, and its role is to define the physical shape of an object. The object's physical outline is also known as the boundary box and is fully modifiable at runtime. (GameMaker Manual 2023.)

Every instance-related logic is written and programmed into events via Object Editor. These events are executed one-by-one – in a predefined order – on every game cycle called frame. An instance lifecycle begins after its Creation Event gets executed. The Creation Event mainly contains object-related variables and properties and is called only once, by default, after a room is loaded. Step Events, Alarms, and Draw Events are executed on the following frame. The instance lifecycle continues until the Destroy Event is called – manually or at the end of a room – or during the final clean-up when the game closes. The predefined event order is illustrated in Figure 1.

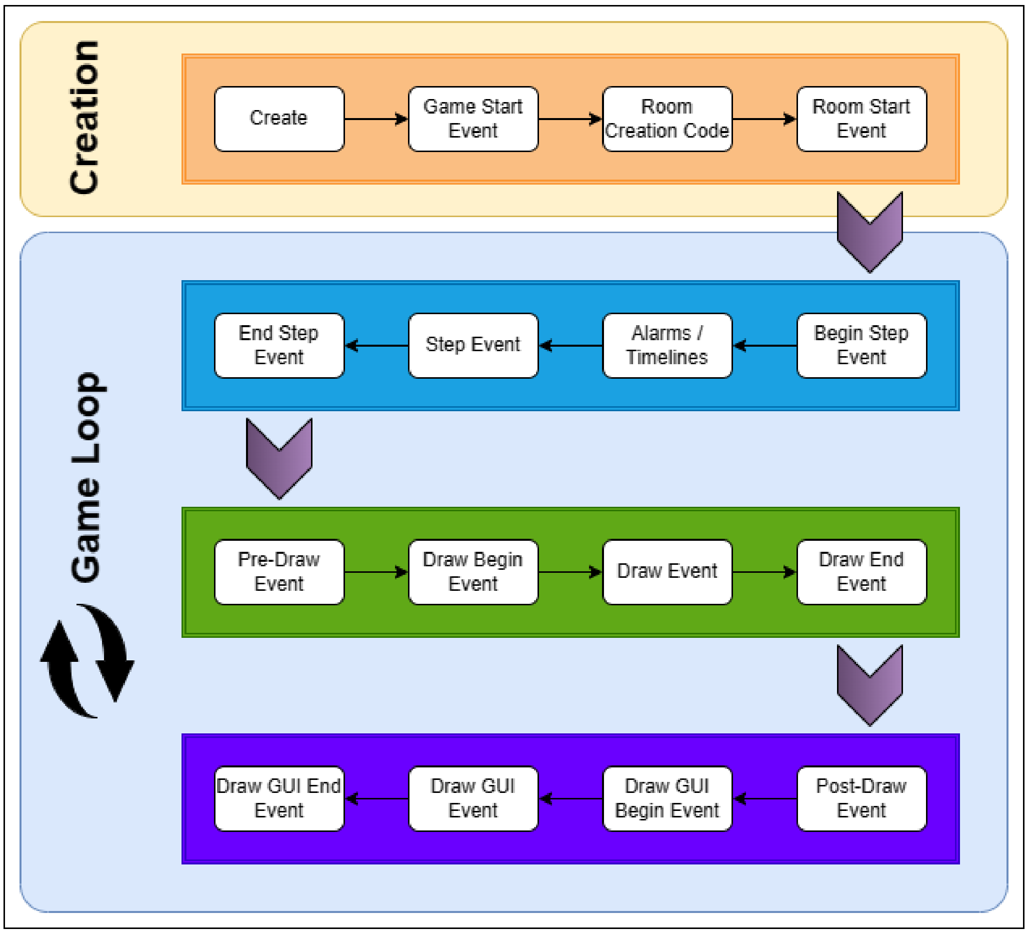


Figure 1. Event order

GameMaker utilizes parent-child hierarchy. Objects can share events, actions, and logic with others through inheritance. Objects with parents are called children, who can inherit or override different events and properties of choice from their parents. Object parenting helps keep the game structure noticeably cleaner and organized. It is a powerful tool for object grouping, creating object bases, and identifying objects by inheritance. (GameMaker Manual 2023.)

Parent-child hierarchy delivers noticeable benefits in large-scale and complex projects. It can eliminate duplicate parts from the code base, especially when dealing with different object variants that, on closer observation, share mutual behavior.

3.4 Networking

In GameMaker networking, game instances – clients – can connect and communicate via sockets. GameMaker Studio 2 features several built-in networking functions that provide simple ways to set up and configure sockets, manage connections, and transmit data packets. (Alexander 2019; GameMaker Manual 2023.)

3.4.1 Sockets

Sockets are categorized by type to TCP, UDP, and regular or secure WebSocket, based on the network transport protocol used. The GameMaker server socket is created with a *network_create_server()* function. The function takes type, port, and maximum number of connected clients as arguments. A GameMaker client socket, in turn, is created with a *network_create_socket()* function by supplying a single "socket type" parameter. Both functions return a unique ID that gives access to further socket calls and an ID value of '0' if the process fails. (GameMaker Manual 2023.)

The way sockets are represented on the operating system level varies depending on the platform. On Windows, sockets' states and data are read via UINT_PTR type memory pointers. On the contrary, POSIX-based platforms like Linux, Mac OS X, and PlayStation, sockets are represented by integer type indices in operation system list of open files and sockets. (Glazer & Madhav 2015, page 68.)

GameMaker Studio 2 offers easy-to-use built-in functions with multi-platform network socket connection support out of the box.

3.4.2 Networking functions

Multiplayer game development with GameMaker Studio 2 requires time and persistence. Fundamental understanding and knowledge of networking, communication protocols, and network packets remarkably ease development. Newcomers can get started with networking by reading GameMaker documentation to seek answers to essential questions like how GameMaker wraps and sends outgoing data packets and which network socket type is compatible with the game project.

GameMaker features several built-in networking functions for data transmission. A *network_send_packet()* function is designed to send data via TCP and, correspondingly, *network_send_udp()* via UDP sockets. Both functions take socket ID, buffer ID, and data size (in bytes) as arguments. Under the hood, these functions format the data so GameMaker can "split" incoming data from received packets correctly. Additionally, the engine includes a 12-byte GameMaker information header for each outgoing packet as part of its networking protocol. However, because the *network_send_udp()* function utilizes a "connectionless" transport protocol, UDP, it requires data about the destination address and port as additional arguments. It is essential to understand that different functions cannot be mixed and matched with incompatible socket types. (GameMaker Manual 2023.)

Clients can also connect to and exchange data with external servers by alternative function calls such as *network_connect_raw()* and *network_send_raw()*. These functions exclude the additional 12 bytes of the GameMaker information header and ignore built-in data formatting. Therefore, they are ideal for communicating with servers built with external technologies and languages, such as Node.js, JavaScript, and PHP. However, this setup requires more development time and coding to implement custom protocols to establish client-server connections reliably because data streams are received in unformatted form. (GameMaker Manual 2023.)

3.4.3 Asynchronous network event

When GameMaker detects an incoming network packet, the Async Network Event is triggered, and the packet is wrapped into a DS map. The packet content is then readable via the global *async_load* variable. The variable contains key-value pairs, including keys such as socket ID, packet type, socket IP address, and socket port number. Afterward, the global *async_load* variable is erased and set to the default "-1" value and is no longer acceptable outside the Async Network Event. (Alexander 2019; GameMaker Manual 2023.)

3.4.4 Buffers

Before GameMaker can send outgoing or read incoming payloads, they must be written into Buffers. Buffers are designated areas in the system memory explicitly reserved for the game. They function as temporal storage where data is acceptable and manipulatable via high-speed read-

and-write operations. These operations are executed sequentially to split data into “chunks,” separating the data into blocks by type. Values are then readable and writeable one by one. The GameMaker engine offers numerous built-in functions to create, write, read, fill, and resize buffers. (GameMaker Manual 2023; Guide To Using Buffers 2023.)

Buffers are created with a *buffer_create()* function. The function takes size, type, and alignment as arguments. The size value defines how much fixed or initial memory will be allocated in bytes for the buffer, and the following arguments supply the buffer type and byte alignment. They go hand in hand and should always be selected consciously to suit the target usage to ensure that the game will function properly. The function returns a unique buffer ID value that gives the game access to newly allocated memory for further operations. (Guide To Using Buffers 2023.)

The engine has four available constant buffer types, the most common ones illustrated in Figure 2. The *buffer_fixed* type is a fixed-size buffer that, once allocated, cannot shrink or grow on runtime. For use cases where data is desired to resize on write operations, the *buffer_grow* type is the best choice. The *buffer_wrap*, in turn, functions as a fixed-size buffer. When the data writing is about to pass the buffer size limit, the pointer will return to the starting point and start overwriting the memory. The final, *buffer_fast* type, is designed for high-speed read and write operations but has compatibility only with unsigned 8-bit integer values and 1-byte alignment. (GameMaker Manual 2023; Guide To Using Buffers 2023.)

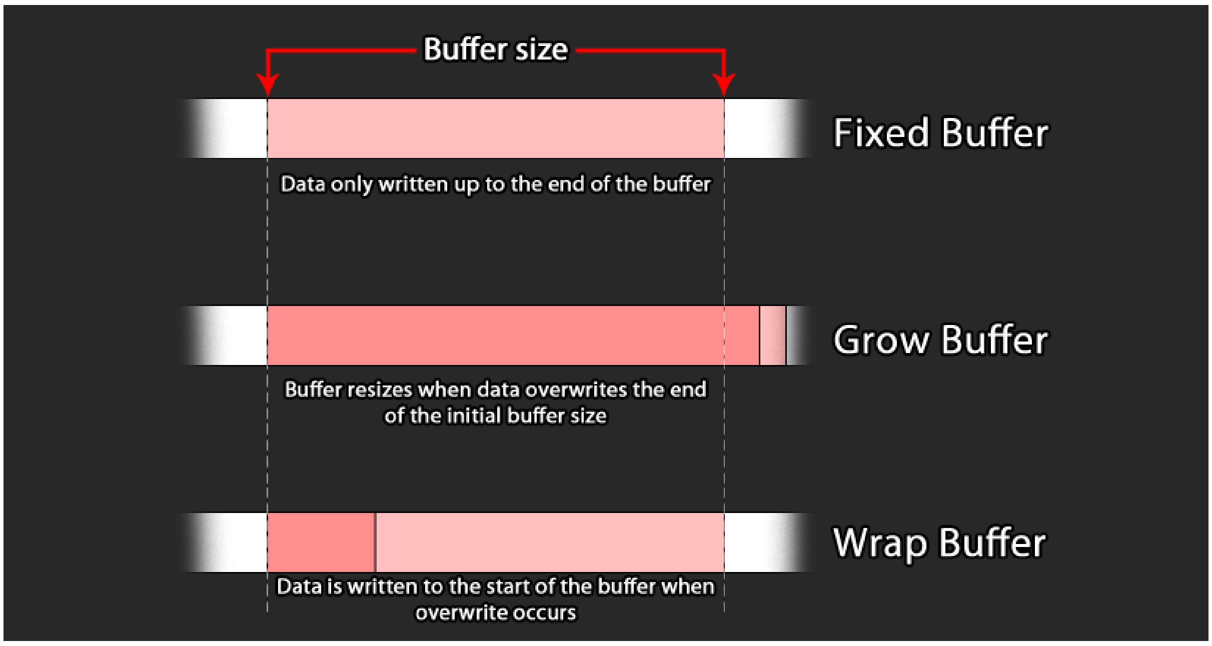

Buffer size
Fixed Buffer
Data only written up to the end of the buffer
Grow Buffer
Buffer resizes when data overwrites the end of the initial buffer size
Wrap Buffer
Data is written to the start of the buffer when overwrite occurs

Figure 2. Buffer types (Guide To Using Buffers 2023)

The alignment value represents an offset or padding in memory. For instance, two 1-byte values written sequentially with 4-byte alignment will sum up to 5 bytes of used memory. Similar examples of buffer alignments are illustrated in Figure 3. The alignment value should always stay within appropriate limits and make sense. However, if there is an ambiguity in choosing a proper value for the offset, the default value “1” is the safest choice. (GameMaker Manual 2023; Guide To Using Buffers 2023.)

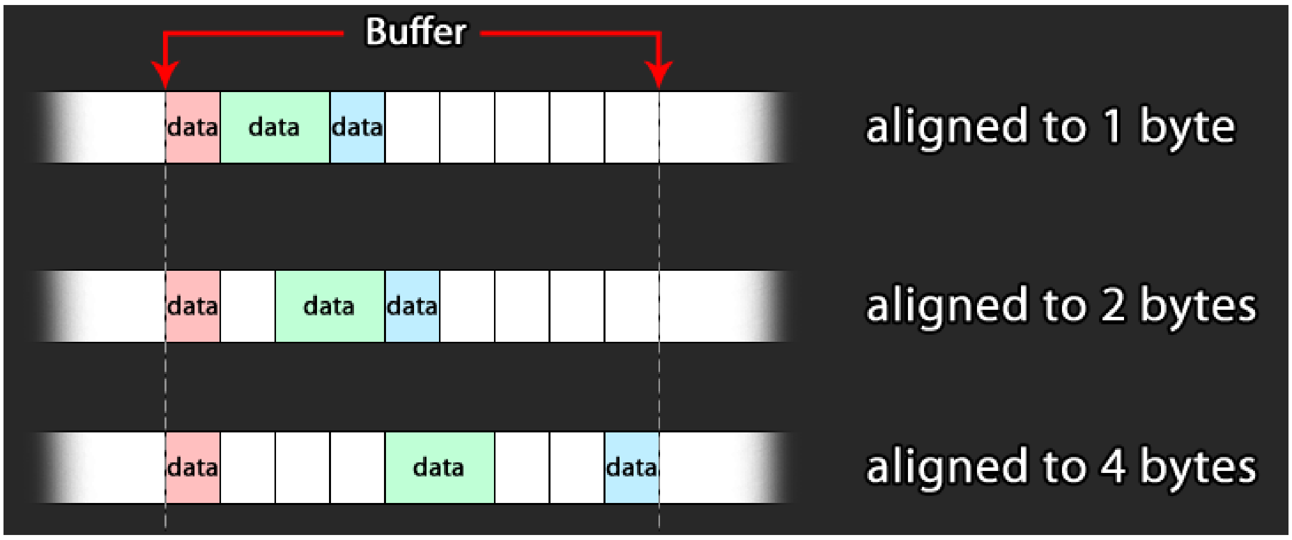


Figure 3. Buffer alignments (Guide To Using Buffers 2023)

Data can be written into memory with a *buffer_write()* function. The function takes buffer ID, data type, and value as arguments. The first argument identifies the buffer to be accessed, and the data type tells the program how many bytes are needed to write the given value. (GameMaker Manual 2023.)

The seek position is a pointer defining starting points for each operation. A *buffer_seek()* function is used to reset the pointer or seek certain positions in memory for buffer operations. The pointer can be set to start, to the end, or to the relative point where the previous buffer operation stopped. (GameMaker Manual 2023.)

When accessing buffers, read operations must be executed on existing buffers with the exact data type arguments as the buffer was written. Otherwise, if the data type does not match the target "chunk" size or the seek position is incorrect, the read operation may output unexpected values or drive the game into an error state. (GameMaker Manual 2023.)

It is important to remember to delete created buffers and release the reserved system memory afterward with a *buffer_delete()* function. Understanding the buffer behavior is essential to avoid undesired outcomes, where access to the buffers is lost on every game reboot when the engine's buffer ID handler gets flushed, but memory regions remain reserved. Therefore, as instructed, existing buffers should be deleted when the game closes. (Guide To Using Buffers 2023.)

An allocated buffer can differ in size from the final payload on data transmission. GameMaker may sometimes reserve more memory than the initialized buffer size if the data is written into a dynamic-type buffer. The engine can modify the reserved memory buffer based on built-in memory management that controls how it prefers to fill and arrange the memory during write operations. As a note, an outgoing payload contains only written data, meaning that the size of the sent data can differ from the original buffer size. One can check the sent payload size from the return value of functions like *network_send_packet()*, presented in bytes.

4 Node.js

In the past, JavaScript was primarily used to program web applications on browsers. The language was suitable for implementing web pages, manipulating DOM trees and URLs, triggering user input events, and building interactive websites. However, use cases for JavaScript were quite limited. At that time, alternative languages such as Java, Python, and PHP provided comprehensive features to read and write files, create and remove folders, create database queries directly, and build web servers. Because of such competition, JavaScript capabilities required urgent improvements, and Node.js was founded in 2009. (Mead 2018, pages 15-16; Young, Meck, Cantelon & Oxley 2017.)

Node.js is an open-source JavaScript runtime used to build scalable network applications. Node.js is an asynchronous and lightweight solution with minimal resource allocation costs. It is built on an open-source JavaScript and WebAssembly engine called 'V8' developed by Google. JavaScript code is converted to lower-level native code, C++, by the engine, which makes code fast to compile and run. (About Node.js n.d.; Mead 2018.)

4.1 Event loop

Influenced by Ruby's Event Machine and Python's Twisted, Node.js uses a similar event loop model, illustrated in Figure 4. Instead of presenting an external library, the model is included in the Node.js standard structure. On application launch, the index file gets executed, and JavaScript code is compiled and preloaded into RAM. Imported modules and project files are available to the application, including code blocks and functions that are only executed when needed. Node.js differs from the other systems, where an initial method starts the event loop, blocking other function calls in the meantime. Node.js does not necessarily require any callback call to enter the event loop. The program keeps running until the end of the last callback method. (About Node.js n.d.; Wexler & Simpson 2019.)

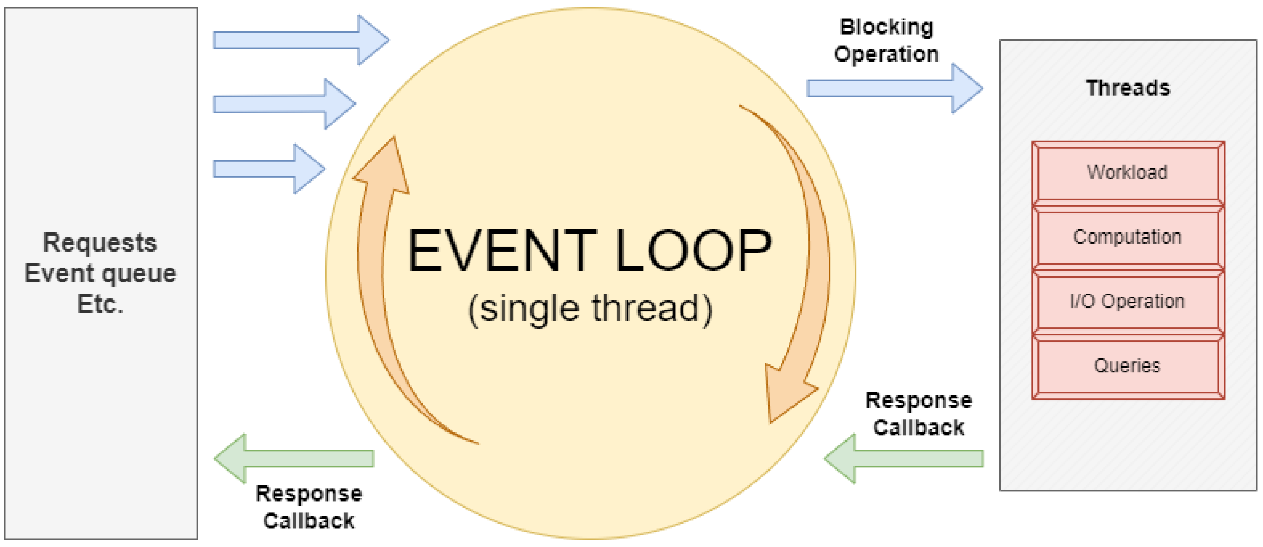


Figure 4. Node.js Event loop

The Node.js application runs single-threaded by default. The event loop handles requests and tasks in the main thread until the workload is too heavy to process, from where it allocates more threads and resources. Asynchronous tasks, like database queries and API requests, are queued and later executed immediately once the target resource becomes available. It is essential to set a reasonable timeout or interval time for callbacks. The main thread keeps processing incoming requests until an event listener gets called about the completed task. The threading logic is hidden under the hood but can be accessed through Node.js API for specific use cases. (Wexler & Simpson 2019; Young et al. 2017.)

4.2 Non-blocking I/O

Applications without event loops execute I/O operations one at a time. This logic model is known as blocking I/O code, where the client is left waiting for the service to respond, causing significant time wastage. Time-consuming processes like disk read-write operations, database queries, and network access can negatively impact the user experience with long wait times. Non-blocking code, in turn, offers significant benefits by utilizing asynchronous task processing and the event loop hierarchy. While one event listener, for instance, waits for the database update task, the program can initiate a second task to send an HTTP response to a client. This approach can handle thousands of simultaneous ongoing events and tasks, making it a more efficient and scalable solution. (Mead 2018, page 33; Young et al. 2017.)

Node.js can effectively handle most workloads, but exploring alternative technologies, like Amazon lambda functions, might yield better solutions for demanding and heavy processing.

4.3 Node package manager

Node.js features several built-in APIs and modules that offers essential functionalities for applications. Node PackageManager (NPM) is the world's most extensive open-source third-party package and module registry. The innovation behind the registry is the ability to share and borrow code. Installing available packages and modules into projects can cut development time significantly and eliminate the need to solve identical problems continually. (About npm n.d.; Mead 2018, pages 63-64.)

Modules are JavaScript libraries and standalone tools, building blocks in applications. They can be installed via Node.js packages into the separate "node_modules" folder or imported into the project base. With proper design, modules can function independently as part of the business logic in one application and be reusable throughout other projects. (About npm n.d.; About packages and modules n.d.; Mead 2018, page 83.)

Node packages are collections of files and modules shared on a public or private registry listing. Public packages must include the mandatory "package.json" file that contains essential metadata about the package, like name, version, author, license, and index file path. Modules can be resolved and loaded into the application using a *require()* function from CommonJS or a "import" statement from ECMAScript modules. (About packages and modules n.d.; Node.js v20.8.0 documentation n.d.)

4.4 node:dgram module

The *node:dgram* is a built-in module in Node.js. It contains an implementation of UDP datagram sockets, including a Socket class and a *createSocket()* function. The *Socket* class is extended from *EventEmitter*. It encapsulates several datagram functionalities, including events emitters like close, connect, error, listening, and message. A UDP socket is bound to listen for datagram messages on a specific port(s) and address on creation. Both initial arguments are optional. If the target port is left blank or set to value 0, the OS automatically binds the socket to a randomly selected available port, usually on the dynamic port range. Missing address value, in turn, will let OS bind the socket to listen to incoming messages on every address endpoint. The socket binding can either finish successfully or trigger an error event that leads the application to a thrown *Error*. (Glazer & Madhav 2015, page 77; Node.js v20.8.0 documentation n.d.)

5 Research design

5.1 Research questions

The research aims to assess and compare available network protocols for their benefits in real-time multiplayer games. Furthermore, an existing game prototype will be adapted with networking to support multiplayer mode, yielding a robust foundation for future development. The research utilizes the applied research method combined with available documentation, published guidelines, and the author's prior knowledge of the development tools and languages to build a client-server integration.

The research and the problem circulate around the following three questions.

1. How to build network communication between the GameMaker client and the Node.js server?
2. How to manage static zones and instances in a real-time multiplayer game?
3. How to implement a simulation layer into a real-time multiplayer game?

The research target is to bring the hiding potential of GameMaker Studio 2 as a multiplayer development platform to the daylight through a demonstration of a game prototype. The developed prototype must meet all the software requirements, see Appendix 1. The new multiplayer mode must be playable on distributed computers while the server can be located outside the local network. The server must support several concurrent players that scavenge and travel in zoned in-game world locations and maintain the base-level synchronization between clients. The server must also feature a simple game area instancing and simulation layer.

5.2 Scope

The topic itself can – without a doubt – delve deep into complexity like client-side prediction, server reconciliation, and synchronization techniques that aim to negate unwanted effects of high latency in network communication. Therefore, the scope of the thesis was selected to cover the integration between the GameMaker client and the Node.js server with a perspective that discusses the basics of the game area instancing and simulation layer. Additionally, the thesis includes several networking topics like communication reliability, network packet handling, and data serialization and formatting. In simplicity, the work expands the single-player game with networking.

5.3 Applied research

Applied research is a modern research method that approaches solving problems in applications and environments by developing and producing practical solutions for working-life sectors. It is highly modular and can combine qualitative and quantitative methods or be directly qualitative. (Kananen 2015, pages 29 and 76; Pernaa 2013.)

Applied research integrates published theory and prior research discoveries with hands-on demonstrations and piloting. Usually, the process utilizes iteration cycles, with continuous development and testing as its major cornerstones (Pernaa 2013). The reliability of the solution and results is often evaluated by practical testing in a real-life environment and comparing them with the predefined requirements (Kananen 2015, pages 29).

5.4 Methods

The literature review presents a valuable theory about the thesis's key topics. The research utilizes a systematic literature review process. The process involves identifying and evaluating collected materials by comparing their relevance with the identified research problems and the application. The literature review aims to capsulize in-depth knowledge from various sources with concrete examples about network topologies and protocols, game design, and tools' architecture.

Records were collected into a pool by progressing from internal sources to external. Jyväskylä University of Applied Sciences provided access to licensed material via an online library containing books, databases, public theses, and e-books. The search was further expanded to include sources such as Finna(.fi), Google Scholar, CORE (research service), IEEE Xplore (digital library), local libraries, online bookstores like Google Books and Amazon Books, and a variety of online articles and websites, ensuring a comprehensive search.

The screening process followed the search phase. Predefined inclusion and exclusion criteria were used to identify potential records from the pool. The most notable factors required that the material was published within a specific time frame and by trusted authors. Materials, specifically ones from less-known authors, were also filtered by their peer-review references.

Licensed materials with restricted access or paywalls were carefully evaluated, selected, or excluded from the pool case-by-case basis.

5.4.1 Development

With proper project management, the development produces a practical application based on the predefined requirements, game design, and theoretical models. In the process, theories and research problems evolve into project planning, documentation, and modeling. The following phase includes prototyping and actual development, and the final phase is dedicated to game and integration testing and result analysis.

The test results evaluate the project success rate, and the product is compared to the Software Requirements Specification, ensuring that all desired multiplayer game features are met. The research and testing coverage was planned to focus more on the system and integration level than an end-point user experience.

5.4.2 Monitoring and observation

Monitoring is an essential part of software development. The method includes collecting and analyzing data and focuses primarily on metrics. Monitoring eases tracking particular events, data transmissions and packet deliveries, concurrent actions, and synchronization in the multiplayer game developed.

Observation, in turn, provides a deeper understanding of the system’s internal state. This method involves analyzing data generated during the program’s runtime. For example, logs and metrics can be analyzed, offering detailed tracking of how a host handles incoming network packets, processes the data, and prepares outgoing packets for transmission.

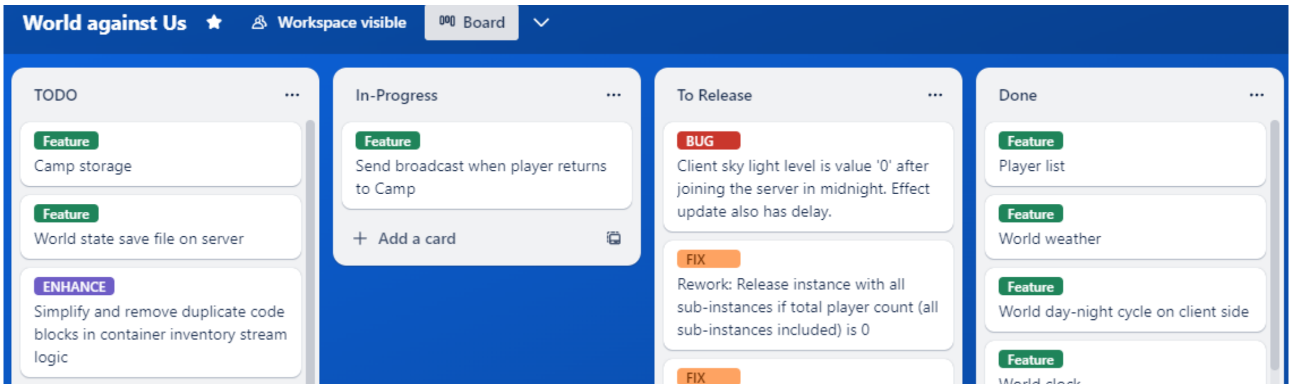
6 Planning and design

6.1 Project management

Proper project management will keep the development work on track from design to implementation. The project will require self-directed project management and execution as it relies on self-sufficient indie game development. Luckily, the Internet offers numerous free tools and applications for project management and visual design. Trello, Clockify, Lucidchart, draw.io, Git, and Sourcetree were selected for the project toolkit.

The project will follow agile project management practices. With agile project cycles, the development process will advance the game toward the desired state while keeping the development time and costs reasonably low.

Each feature, issue, and bug – noted or under development – will be written into a ticket and placed onto the Trello Kanban board, as shown in Figure 5. The Kanban board will be split into categories like Todo, In-progress, To Release, and Done. This will help maintain task tracking and ensure that features or fixed bugs are tested and reviewed before the project enters the next cycle.



World against Us ★ Workspace visible | Board

TODO

- Feature — Camp storage
- Feature — World state save file on server
- ENHANCE — Simplify and remove duplicate code blocks in container inventory stream logic

In-Progress

- Feature — Send broadcast when player returns to Camp
- + Add a card

To Release

- BUG — Client sky light level is value '0' after joining the server in midnight. Effect update also has delay.
- FIX — Rework: Release instance with all sub-instances if total player count (all sub-instances included) is 0
- FIX

Done

- Feature — Player list
- Feature — World weather
- Feature — World day-night cycle on client side
- Feature

Figure 5. Trello Kanban

Architecture and data structures, game logic, and game design – complex or simple – will be illustrated in visualized form using visual diagram tools such as Lucidchart and draw.io. Visualization eases development by splitting the game structure into understandable and logical figures and models.

6.2 Platform and tools

New multiplayer core features for the game client will be built on the Desktop-licensed version of GameMaker Studio 2. JavaScript coding and the Node.js server will be implemented using Visual Studio Code, which the author prefers as the most suitable IDE for the job. Because of license pricing, the multiplayer game developed will only have the Windows Desktop platform support.

As the original engine for the game project, GameMaker Studio 2 was a clear platform choice because it offers powerful tools to build 2D games with a low development workload. The latest stable game version has a solid foundation for further prototyping, including existing game logic, data structures, and UI frameworks.

Visual Studio was a suitable IDE choice because its free licensing, built-in terminal, and extension library brought considerable valuable benefits to the project. Node.js has a npm package manager that offers a variety of convenient third-party modules and libraries for the multiplayer game server. JavaScript and Node.js are comprehensive and efficient technologies for prototyping, and the runtime is easy to install and set up. The most notable installed npm modules and packages are listed below.

- node:dgram (built-in)
- node:zlib (built-in)
- dotenv (v^16.3.1)
- moment (v^2.29.4)
- uuid (v^8.3.2)

Versions

The used software and runtime versions are GameMaker Studio 2 (IDE 2023.8.2.108 and Runtime 2023.8.2.152) and Node.js v20.10.0 LTS.

6.3 Software requirements specification

In software development, it is vital to have a clear understanding of the application's structure and functionalities. Without this understanding, the project can lead to confusion and endless questions about the product's primary objectives and desired quality. A software requirements specification documentation (see Appendix 1) is included in the project plan to avoid such situations. The documentation outlines the new features and defines the MVP (minimum viable product), including the requirements for the target prototype iteration. These practices help improve the planning, game design, and overall project quality.

The software requirements specification also helps to map the problem into smaller, manageable elements. For instance, the server should have a client registry and identify a connecting client using UUID to send network packets. Furthermore, to implement a custom application layer protocol, the network packet header should extend the UDP header with a message type, client ID, and sequence number. These were brief examples of the elements in question.

6.4 Game prototype

World Against Us is a prototype for a real-time 2D action game with base-building and survival elements. Players find themself in a post-apocalyptic world where competition over remaining supplies and resources drives humankind against each other into chaos. The key to survival is scavenging local towns and forests by running expeditions to gather anything – what is left – while building a safe and hidden Camp to stay alive for as long as possible. Players must adapt to cruel nature and weather conditions, avoid dangerous bandit patrols, and wisely use limited daylight hours. In this disintegrated world, there are no true winners.

6.4.1 Game architecture

The target prototype iteration will have new embedded networking components, extending the game logic with multiplayer mode support. Adding such support in as early development states as possible leaves breathing room for the code base modification work before the planned changes become too expensive with radical reconstruction of the game base. The game architecture is illustrated in Figure 6.

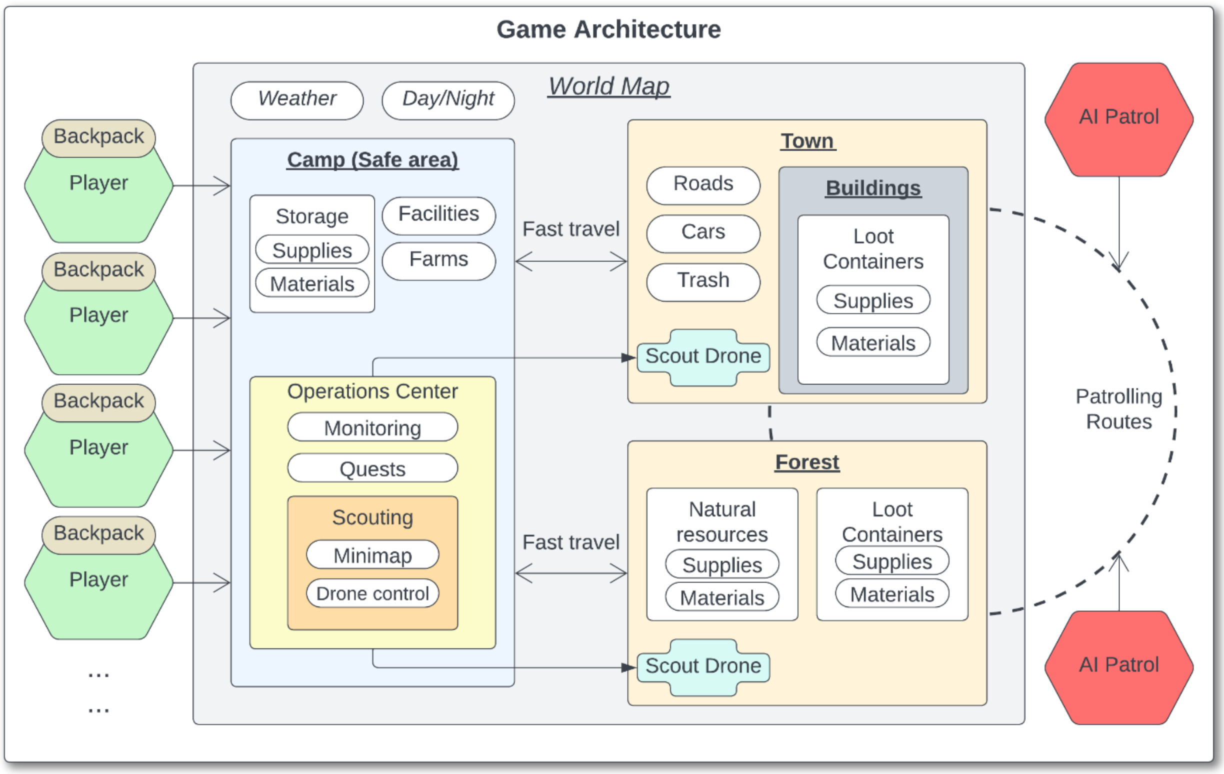


Figure 6. Game Architecture

6.4.2 Gameplay loop and elements

One fundamental target in video game design is a rewarding gameplay experience. Ideal gameplay experience arouses a believable feeling in the player base that they have primary goals and objectives to complete – something they have the will to fight for. (Tremblay 2023, page 60.)

Survival games and popular titles like *Subnautica* are accomplished to create never-ending scenarios where players must prevent their characters from dying of hunger and dehydration. However, satisfying the character's needs is often one of the minor threads in video games. Typical enemies like bandits often represent the evil source of disorder and destruction that infects the atmosphere with fright and insecurity. (Tremblay 2023, pages 59-60.)

Surviving

In the game prototype developed, the survival aspect is the primary element of the gameplay loop. Survival brings satisfyingly tough challenges, including players having to sustain their supplies to break even with the drain. It leads players on expeditions and scavenging trips into local world map locations. Players must develop skills in several sectors, from sneaking and avoiding dangerous opponents to navigating through dark and misty environments with limited vision. They must also manage their finite carrying capacity and selectively haul the most vital resources from trash piles and caches. Players can transport retrieved supplies back to the camp and store them securely into storage for future use.

Bandit patrols

Bandits are enemies, humans, that players should avoid at any cost in the wasteland. Bandits are dangerous competitors with the mutual motivation of surviving. They own overwhelming strengths and mobility to catch and rob players, forcing their competitors to flee the area empty-handed. Bandits travel lands along the patrolling routes, but their schedule remains unpredictable.

Scouting

The operations center in the game prototype provides virtual monitoring and a remotely controlled unmanned flying drone to scout areas and observe the patrolling bandits. Unfortunately, the drone has a limited connection range, restricting scouting coverage in the world map locations to outdoor areas. Nonetheless, it is essential equipment for gathering information. Players can benefit from the collected intel that helps plan new expedition routes, schedule the operation execution, and get a clear picture of targeted locations.

6.4.3 Level desing and world zoning

Level design and world-building together form an artistic craft of creating an emotional connection between players and the virtual world. The level design creates an in-game environment where players can wander and interact. Residential areas, nature, and unique locations around people and animals construct a world where every aspect, from geographical features to materials and props, fits into their typical environment and century. (Tremblay 2023, pages 53-54.)

The world map in the game prototype has a sanctuary called a camp. It is hidden from outward dangers, giving players room to rest and be safe. The camp is located on the western side of the local river, where anyone else has yet to reach. It is a small outdoor area with a storage unit and construction sites.

The game prototype has two locations, a town and a forest, for players to visit. The town is an unpredictably dangerous place with silent streets and abandoned buildings. Local offices, libraries, and shops contain caches with potential piles of materials and resources for players to scavenge. Indoors and houses function like temporary safe rooms that are unreachable from bandits. However, they are not suitable spots to settle down for a permanent stay. A typical office building is illustrated in Figure 7. The known fact that the town has a limited variety of resources forces players to extend their map coverage into the forest. The forest – middle of nowhere – seems a safe area in the first place, but unfortunately, it is also along the bandits’ patrolling route.

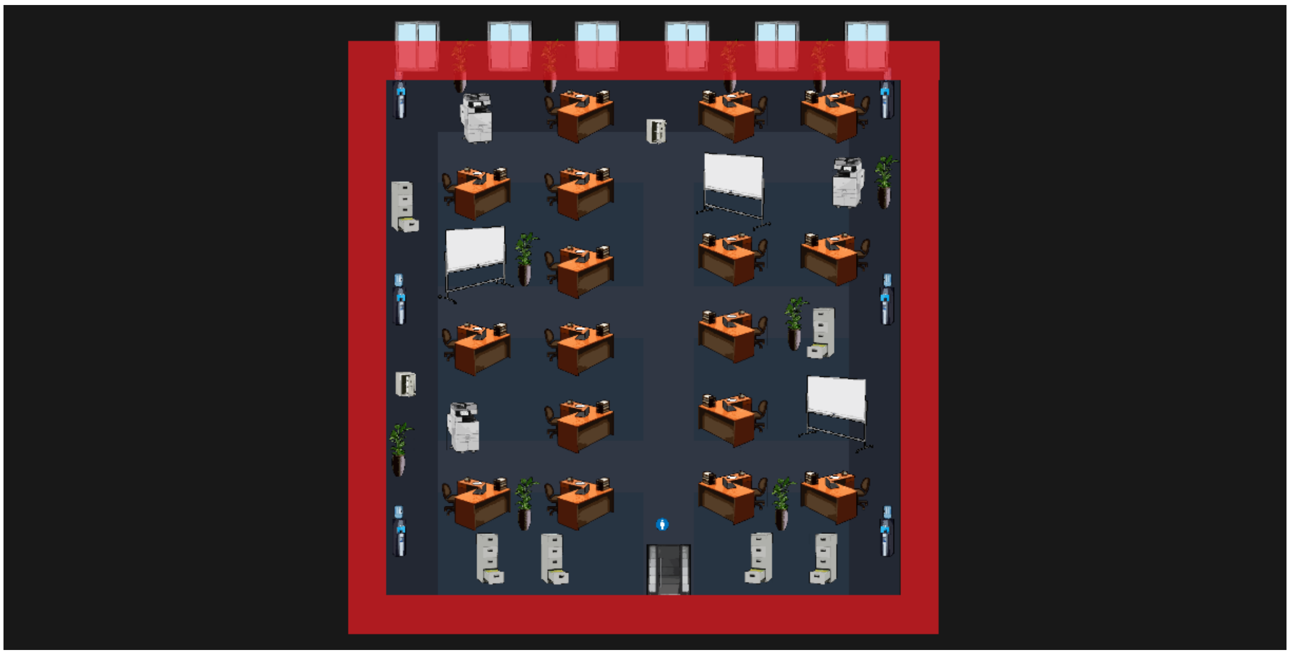


Figure 7. Level design office

From the technical perspective, because the effectiveness of the room editor in GameMaker Studio 2 focuses on building average small-sized game areas and rooms, world zoning has come naturally during the prior development process. Therefore, outdoor and indoor areas represent static zones with boundaries in the game world, distributing players into individual rooms. These room boundaries with the predefined static zones will ease the workload on multiplayer development.

World map

Maps are categorized into two categories in video games: diegetic and non-diegetic maps. A diegetic map is a physical object in the game world that characters can hold in their hands or observe on a table or a mounted frame. The non-diegetic map, in turn, is a game UI element often accessed through a game menu or a bound key and shown on the screen. It can also be a small framed minimap at a corner of the game UI. There are endless ways to present and style world maps. For instance, maps can be illustrated with a pen and paper or, like in a video game called *Skyrim*, as an interactive map. A map can feature built-in zoom and scroll controls, utilize effects to highlight points of interest, and have fog to cover undiscovered areas. (Tremblay 2023, pages 79-81.)

A world map is an excellent tool to visualize in-game lands and areas. In the game prototype, the map UI element also serves as a fast-travel interface. When players want to leave the camp by interacting with the exit point, a world map window opens on the screen, as illustrated in Figure 8. There are two selectable destinations: a town and a forest. The target prototype iteration will extend the world map window with a list view of available instances for players to travel.

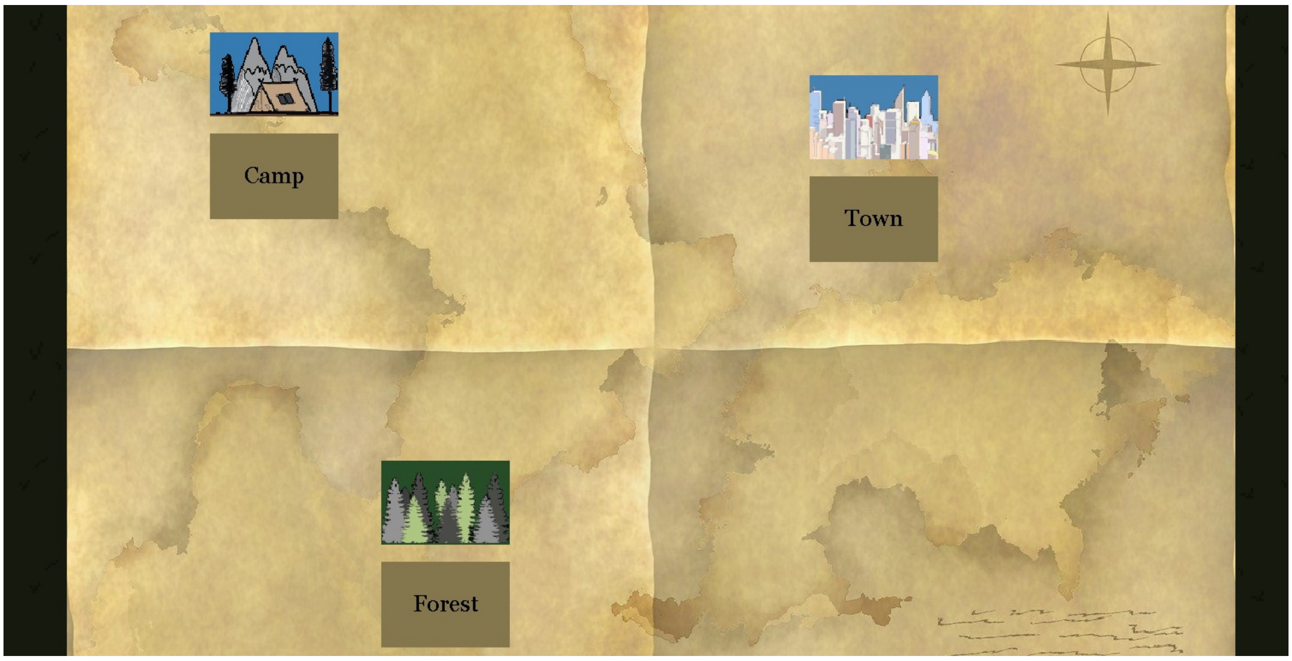


Figure 8. World map template

6.5 Multiplayer core features

The prior World Against Us versions were initially developed to support only the single-player mode. However, it is noticeable that modern games are more likely to support multiplayer or co-op mode sooner or later. The changeover has already been seen in popular game titles like *Stardew Valley*. The multiplayer mode will offer a new social aspect to the gameplay experience that brings value to the game. With new multiplayer core features, the target prototype iteration will have base-level networking support for further multiplayer development.

6.5.1 Network communication

The technical implementation is planned to begin with establishing network communication and a base-level synchronization between the client and the server. Networking will be the central point of the target prototype iteration. The network communication will utilize the UDP transport layer protocol for real-time data transmission. The project will briefly delve deeper into reliability issues, excluding more advanced techniques and theories. Nonetheless, the quality requirements will ensure a smooth and functional gameplay experience.

Furthermore, the network metrics and monitoring will not cover in-depth sampling. However, the target prototype will have a crude foundation and functional integration within the software requirements specification. The fact that time and available resources are limited leaves the implemented game in a prototype state for now. But one step closer to the release.

6.5.2 Game area instances

Because the planned multiplayer mode will allow players to join the public game sessions, the game world will be significantly more populated than in single-player mode. Therefore, the target game prototype iteration will utilize game area instancing in numerous ways. It will offer scalability and, most importantly, give a unique touch to the gameplay. When players wish to visit a world map location, they should be able to fast travel to existing instances available or request a new one from the server. When the last remaining player leaves an area, the instance – excluding the camp – should be deleted and no longer appear available.

One notable challenge during the upcoming development will be instance management. Maintaining synchronization and integrity will require cautious planning and comprehensive testing. Players can perform unpredictable actions that may deviate from the assumed chain, which emphasizes the importance of testing. Because instances may contain numerous objects to be synchronized on each participating host, the networking must utilize, for instance, data replication, fragmenting, and streaming in the name of bandwidth optimization. However, the primary focus for the target prototype iteration is not to build a perfectly polished management system – the research rather aims to prove the compatibility of the client-server integration.

6.5.3 Simulation layer

In the target prototype iteration, world state elements such as date, time, dynamic weather, and other simulations will be adapted with networking. Network communication ensures the world state synchronization between the server and participating clients. Furthermore, each element on the simulation layer will be separated between clients and the server as part of a planned authority distribution. The simulation layer will also serve as a framework, with a variety of controllers and components, for AI behavior and operations center features.

The target prototype iteration will introduce AI behavior and bandit patrols to the multiplayer mode. A new extended logic will run AI simulations simultaneously on the global and instance levels. With network communication, the server can ensure that every action or change in AI behavior is synchronized between the participants. For instance, when an enemy bandit detects a player passing by and desires to leave their usual patrolling to rob the target, the AI behavior switches from one state to another. The new state is broadcast to clients, who can synchronize their local game state with the server. The state should remain unmodified and equal on hosts until the enemy bandit changes their behavior again and returns to patrolling. That was a brief example among all possible cases.

The multiplayer mode will introduce new operating aspects. The operations center will be adapted to the simulation layer and networking, allowing players to monitor and scout instances remotely. With such equipment, players can organize their actions and processes in groups. One player can operate remotely via a terminal from the camp while other players scavenge the wasteland.

The terminal element will utilize bidirectional network communication over the instances with data streaming and virtual simulations. The terminal tracks the player's and enemies' movements on the operating client and renders character figures in a miniature-like map view. The map view will be rendered based on incoming data streams of dynamic objects and loading the remaining static map data from the locally generated files. In addition, the remotely controlled drone will be synchronized with participant clients to render an actual physical flying drone on their screen.

6.6 Hybrid server model

Synchronizing every player’s movement and actions between clients on a peer-to-peer server model would be complicated in a game where players live in separate instances. In GameMaker Studio 2, a room represents an instance, and the player’s actions are bound inside it. Because the incoming and outgoing network traffic scales linearly with the number of connected clients, the bandwidth usage will grow simultaneously at the same rate. Furthermore, if the game is meant to perform different events requiring authority-based checks and consensus among the hosts, clients must broadcast their local state snapshots to every other participant. These events can include collisions between the game objects, for instance. In both cases, the bandwidth usage can be challenging to reduce and optimize in some applications.

On the other hand, programming a dedicated server with GameMaker's built-in networking system would provide full server-side authorization and allow engine code to run on the backend. Nevertheless, there are some known issues. GameMaker runs single-threaded and does not support multi-threading or hyperthreading. Its networking system has a limit of 1000 connected clients, though it does not impact small-scale multiplayer servers. GameMaker Studio supports common runtime output types for Windows, Linux, and MacOS platforms, but the game engine has dependencies with graphical user interface APIs like *DirectX* and *X11 Display*. That would restrict the possibility of running a server on systems and platforms that lack such compatibility. Additionally, implementing a fully dedicated server on external languages and frameworks and writing replicated GameMaker engine code from scratch would cost unnecessary development time, even with the help of third-party modules and libraries.

Because none of the presented architecture fits into the project perfectly, the problem is attempted to be solved with a custom hybrid server model. This hybrid implementation will utilize several relay and dedicated server model features to solve significant networking system shortages in GameMaker Studio 2. The scope is not to invent the best or the most secure solution for multiplayer servers but to create a crude version for prototyping and research purposes. Nevertheless, the work still follows the software requirements specification document accurately. It ensures the game meets the minimum viable product classification with a solid foundation for the alpha version.

The idea behind the hybrid model is to allow multiplayer logic to run partially on the client-side engine, which makes the model highly modular in scenarios where the server and the client do not share the same engine base. The developed Node.js server will function like a relay server and maintain the synchronization between participant clients. It does not try to mimic the GameMaker engine logic completely. The implementation has mutual elements compared to the asynchronous lockstep protocol. Clients can advance in time without negotiations with other hosts until the next interaction triggers the synchronization call between them. The Node.js server will have control and authority over the global game state. The clients, in turn, can focus on rendering the game and, occasionally, perform minor tasks at the server's request. This method helps balance the workload between the server and clients. The hybrid server model is illustrated in Figure 9.

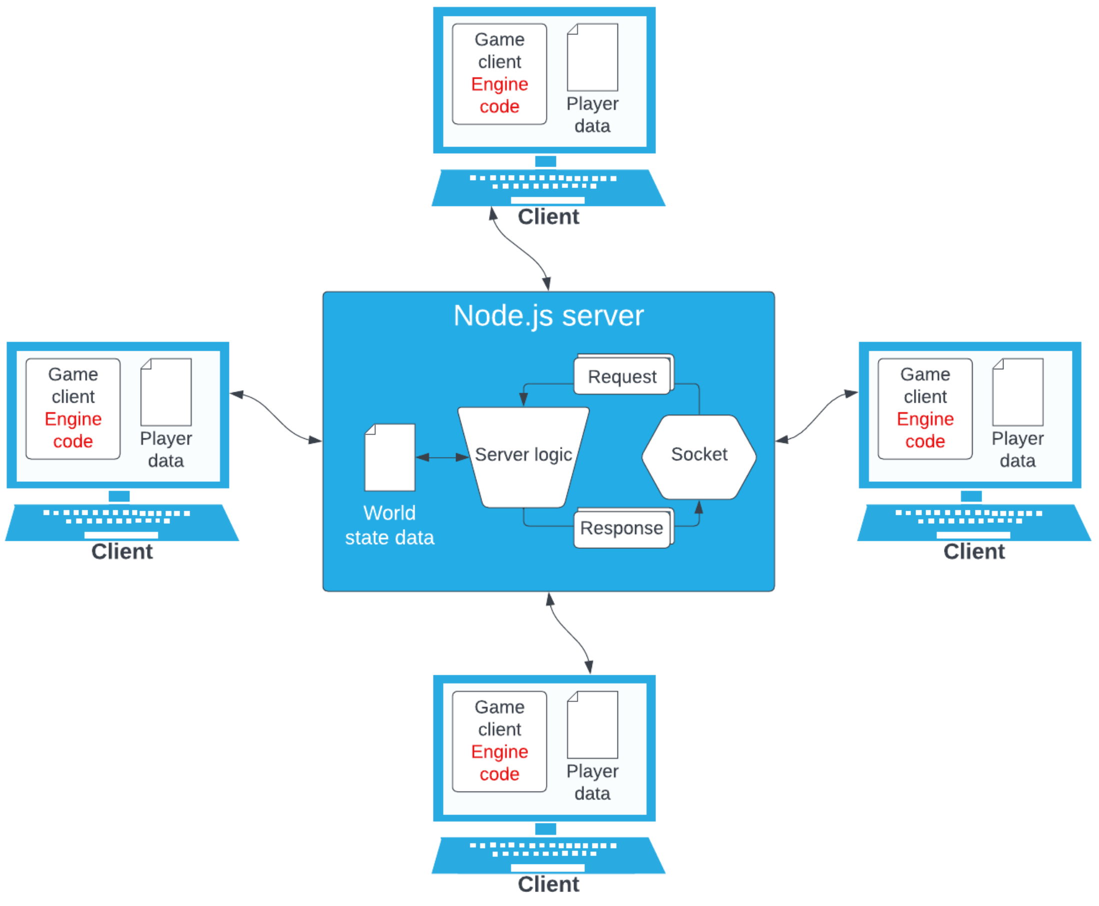


Figure 9. Hybrid server model

The hybrid model will be flexible and support the most common server hosting setups. If the host user faces problems with port forwarding, firewall rules, or NAT, they can host the server on any cloud platform that supports the latest Node.js version and “headless” programs. Because the Node.js server will be a separate instance from the game engine and client, there is no need for a named game client to host the server locally. Instead, the server can be available and online around the clock. Furthermore, clients can freely join and leave the game whenever they desire without causing the server to shut down.

6.6.1 Authority

The Node.js server will have full authority over the client registry and connections. It can disconnect clients without changing acknowledgments with them. Clients, in turn, must complete the related acknowledgment steps on every disconnect attempt.

Because the Node.js server will not simulate player movement or interactions, the game client will have the authority and responsibility to provide player-related updates to the server. Local player parameters, data, and actions will be relayed and broadcast through the Node.js server to keep clients synchronized in predefined intervals and triggers. Clients can then draw and simulate the movement of remote players based on received data. There will be local game client data that would be non-relevant to synchronize with other participants. That includes local sound and particle effects, player inventory, and collision events. The server will still authorize and update interactions and events that impact the world state. The described authority distribution is illustrated in Table 1 with the following text-markings: **Authority** and *related action or data*.

Table 1. Authority roles on gameplay

Server	Client
Client registry and authentication	*Login and disconnect requests*
Network packet content/format validation	*Network packets: request/update*
PING clients	**PING the server**
PING timeout and client disconnect	*PING timeout and disconnect*
Player data request	**Local player save files**
World state save files	*World state data request*
World state change validation	*World state change request*
Broadcast: World state update	*Update locally cached world state*
Broadcast: player position and input	**Local player position and input**
No tracking or broadcasting	**Local player collision events**
Broadcast: player movement and speed	**Local player movement and speed**
No tracking or broadcasting	**Local player inventory**
Broadcast: player interactions	**Local player interactions**

The Node.js server will control the game area instance registry and authorize incoming fast-travel requests. At first, it will create a new game area instance or fetch an existing one from the registry before allowing a client to travel to the requested instance. Because the server cannot run the GameMaker engine code for 2D physic simulations and most of the AI logic, the instance must have a named region owner-client. The owner role will typically be assigned to a client who arrives at the location first. From now on, the owner-client will perform minor tasks and calculations and provide game logic for the server.

Tracking networked objects in a shared game world will require server-side registries to store network ID identifiers for each object. Region owners will have essential duties to inform the server about dynamically networked objects' data and updates within their current instances. Other clients can then request networked object data from the server-side registry to synchronize their local game state when they enter an instance.

The described authority distribution on game area instances is illustrated in Table 2 with the following text-markings: **Authority** and *related action or data*.

Table 2. Authority roles in game area instances

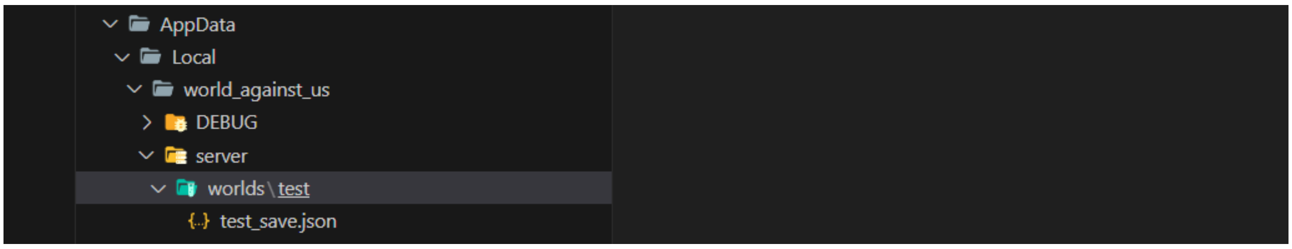
Server	Client: Instance owner
Game area instance registry	*Instance detail request*
Game area instance creation and management	*Fast travel request*
Region owner registry and role assigning	*Region owner request*
Network ID registry and ID assigning	*Network ID request*
Container content registry	*Container content request*
Random loot request	**Random loot generation**
Container content change validation	*Container content change request*
Patrol registry	*Patrol data request*
Wandering patrol registry	*Wandering patrol inspecting on the world map*
Broadcast: patrol updates in an instance	**Wandering patrol in local instance**
Broadcast: AI pathfinding state in an instance	**Local AI pathfinding calculations**
Broadcast: AI behavior changes in an instance	**Local AI behavior changes**
Interaction consequences	**Player interactions with wandering patrols**

6.6.2 Save files

The game data will be split into separate save files and stored on distributed hosts. Only the online world state data will be stored on the server side, while player data will be located on the client side. This multi-save file model allows users to progress their player character in the multiplayer mode or continue playing in the single-player mode and use locally stored save files. Simply put, a game character can jump between multiple worlds. This implementation is partially influenced by popular game titles like *Terraria* and *Valheim*. Save files will be stored in the JSON format. The game will utilize autosaving feature in both online and offline modes but restrict manual saving during multiplayer sessions.

The file system in GameMaker has limited access to perform read-write operations to folders and files on the local operating system. However, a security mechanism known as a sandbox includes two accessible file locations called the *File Bundle* and the *Save Area*. All packaged files – the game executable and included project files and assets – are stored in the *File Bundle*. This location contains files and folders that the GameMaker engine can only read and search. The *Save Area*, in turn, allows GameMaker to read and write save files freely. (GameMaker Manual 2023.)

World data will be saved under the game's %localappdata% folder, which GameMaker utilizes natively. The server-side world data will be saved in an identical location under a "server" subfolder to ensure consistency in the designed file system. The described directory structure is illustrated in Figure 10.



```
AppData
  Local
    world_against_us
      DEBUG
      server
        worlds\test
          test_save.json
```

Figure 10. Local Appdata directory

Player data

The player data is separated from the world state data to accomplish mechanical flexibility, allowing users to choose their playable character for each playthrough freely. The player data save file contains basic data like the last known player location, stats such as health, stamina, hunger, hydration, and energy, character data like name, type, and race, and the player's equipped backpack with items.

World state data

The world state data save file contains a record of the latest world state snapshot. The snapshot holds basic information like the last known time and date, time scale, weather conditions, and camp storage content. However, dynamically generated instances and related data will be excluded from the save file in multiplayer mode. That is because the data will, sooner or later, lose its value after the related instance gets deleted by the server or the next server reboot, making storing such data a waste of disk space.

7 Technical implementation

7.1 Network sockets

The first step to establish a connection between GameMaker Studio 2 and Node.js was to create UDP sockets and bind them to listen to a specific port.

The server-side socket was created in the executable “server.js” index file with the following node:dgram module import and function call.

```
import Dgram from "node:dgram";
const server = Dgram.createSocket("udp4");
```

The socket was configured, as shown below, with emitters and event handler functions to listen to messages and handle socket errors. When the socket receives a network packet, the packet is passed with remote address info to a head network controller called *NetworkHandler*. If the socket detects an error, the server stops and closes the socket.

```
server.on("error", (error) => {
  ConsoleHandler.Log(`Server error:\n${error.stack}`);
  networkHandler.onServerClose();
});

server.on("message", (msg, rinfo) => {
  try {
    networkHandler.handleMessage(msg, rinfo);
  } catch (error) {
    networkHandler.onError(error);
    setTimeout(() => {
      ConsoleHandler.Log(`Server error:\n${error.stack}`);
      networkHandler.onServerClose();
    }, 2000);
  }
});

server.on("listening", () => {
  const address = server.address();
  ConsoleHandler.Log(`Server listening ${address.address}:${address.port}`);
});
```

Finally, the socket was bound to a specified port number and IP address read from environment variables. By default, the socket was set to be bound to port 8080 and localhost address 127.0.0.1 if these variables were not presented.

```
server.bind(process.env.PORT || 8080, process.env.ADDRESS || "127.0.0.1");
```

After the setup, network packets can be sent by calling a send method from the socket object. This method takes a network buffer, destination port, destination address, and a callback error function as parameters. The function call is shown in the following JS code.

```
this.socket.send(compressNetworkBuffer, client.port, client.address, (err) => {
    if (err ?? undefined !== undefined) {
      ConsoleHandler.Log(err);
    }
  }
);
```

The client-side socket, in turn, was implemented as part of a struct-type NetworkHandler component. The handler contains a “CreateSocket” function that creates a new socket by passing a constant *network_socket_udp* value to a built-in *network_create_socket()* function as a parameter. It then assigns the received socket ID to the *socket* member variable. The “CreateSocket” function is shown in the following GML code.

```
/// @function     CreateSocket()
/// @description  Creates UDP socket and allocates network buffer
/// @return {Bool}
static CreateSocket = function()
{
  var isSocketCreated = false;
  if (network_status == NETWORK_STATUS.OFFLINE && is_undefined(socket))
  {
    socket = network_create_socket(network_socket_udp);
    pre_alloc_network_buffer = buffer_create(256, buffer_grow, 1);
    isSocketCreated = true;
  } else {
    global.ConsoleHandlerRef.AddConsoleLog(
      CONSOLE_LOG_TYPE.ERROR,
      "Client already connected or socket already exists"
    );
  }
  return isSocketCreated;
}
```

In GameMaker Studio 2, received network packets are written into an *async_load* DS map variable that holds the data in a network buffer. The *async_load* variable is globally accessible only from an *Asynchronous Networking Event* in instances triggered by an asynchronous event callback. It made fetching the data directly from the struct-type *NetworkHandler* impossible and, therefore, required to pass the data to the handler component another way. A helper object called *objNetwork* was created for that role with a corresponding event, shown in the GML code below. When the event is triggered, it checks the buffer size, fetches the data, and passes it to the *NetworkHandler*.

```
/// @description Asynchronous Networking Event
if (async_load[? "size"] > 0)
{
  var networkBuffer = async_load[? "buffer"];
  networkHandler.HandleMessage(networkBuffer);
}
```

Packet transmission was successfully established by selecting the correct client-side socket method for the job. GameMaker provides built-in network_send_udp() and network_send_udp_raw() functions to send data via UDP socket, but only the “raw” version seemed compatible with the selected Node.js module. The network_send_udp_raw() function, as shown in the GML code below, takes a socket ID, destination address, destination port, network buffer, and buffer size as parameters. The function then returns the sent data size in bytes.

```
networkPacketSize = network_send_udp_raw(
  socket, host_address, host_port,
  compressNetworkBuffer, buffer_get_size(compressNetworkBuffer)
);
```

The network_send_udp() function, in turn, caused the dgram socket to drop incoming network packets. This was most likely caused by the dgram protocol, which refused to handle packets with an additional 12-byte GameMaker information header provided by the engine.

7.2 Network packets

After the server and client successfully exchanged messages, they were still incapable of processing the data further. The receiving host had no clue how to handle or parse incoming network packets because the packets did not include further instructions or descriptions of their content. The packet structure was compact and contained only a base UDP header and a payload like a string or random value. Additionally, hosts were unaware if a sent packet had reached its destination. Therefore, a custom protocol layer was introduced, and the packet header was expanded with additional bytes. The resulting custom header structure is illustrated in Figure 11.

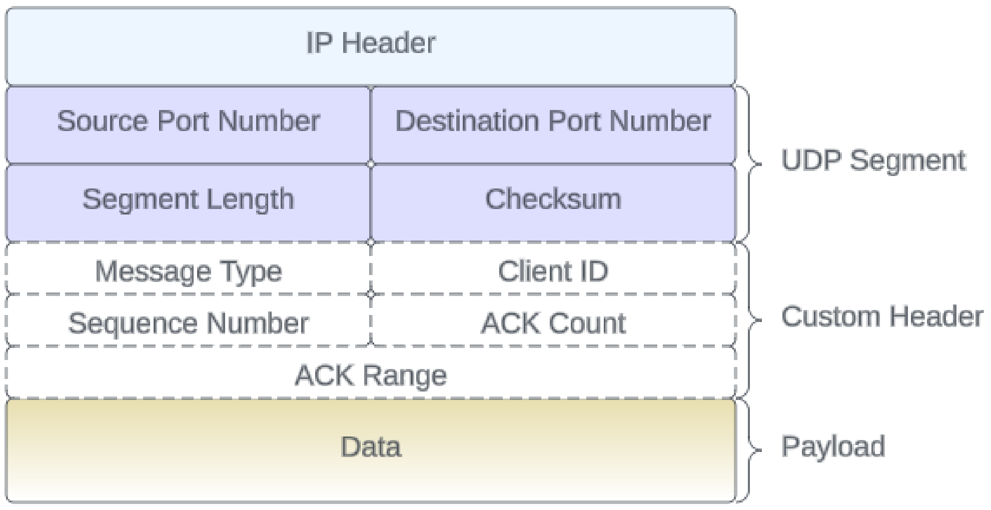


Figure 11. Custom packet header

7.2.1 Client ID and Message type

The expanded header structure now included a 37-bit client ID and an 8-bit message type value as part of the application-level protocol. The client ID is a unique string with a null terminator ending that can be used to identify packets' senders. The ID ensures that the network packet is sourced from a client that has gone through a connection registration process. This allows the receiving host to filter random spam from unknown sources and unregistered clients. The message type value, in turn, tells the receiving host what data the packet is expected to contain and how the host should process the message. Message types are categorized according to their purpose and data handling methods:

- Request
- Ping
- Sync
- Data
- Destroy
- Stream
- Invalid Request
- Error

A network packet with the *Request* message type is sent to perform a single stateless request or action. For instance, a client can send a request to join the game, start fast traveling the player, or fetch data from the server. The associated response can contain the requested data, a new state, or give the client permission to proceed with the action locally. A *Request* packet can also start a chain of actions, such as a client registration process. For clarification, in some applications, a player movement input could be sent via a *Request* packet to make the player character take a step. However, that model may work better in turn-based games. So, it was more logical to introduce a *Data* message type to send commands and state updates, such as movement and positions, in the current game.

Sync, *Data*, and *Destroy* message types are more or less equivalent to commands create, update, and destroy. A *Sync* packet is most often sent when a player arrives at a destination and is usually requested by the client during a *Room Start Event*. A *Data* packet can contain a snapshot of one or more objects, like their position or state, telling a receiving host to update their local game state. When a host receives a *Destroy* packet, the host is instructed to search the local object or target data and destroy it.

A *Stream* indicates a continuous data flow or a state with a start and end condition. When a client requests to start streaming data, the server registers a new active stream and responds with related metadata. Hosts transmit data during the stream bidirectionally by turns or one-way at intervals. Streaming allows hosts to continuously receive data updates from a desired source or balance bandwidth usage by transmitting large data collections in smaller blocks over time.

An *Invalid Request* packet, in turn, can be a response to a sent network packet with invalid information, improperly formatted data, or an outcome of an unhandled request or action. The response packet provides instructions for the original sender on how to proceed. For instance, the response can demand the host to cancel the action or perform a disconnection.

An *Error* packet always leads to an immediate disconnection. For instance, when the server has driven into a state where it cannot continue executing the tick loop, it broadcasts the *Error* packet to each participant before closing. On an unhandled server crash, clients do not receive the *Error* packet but are timed out and disconnected by themselves shortly after losing the connection.

7.2.2 Tagging outgoing packets

The UDP header with a client ID and message type alone still needed some crucial mechanism for the game to maintain communication reliably. Outgoing network packets had to be identified and tagged to keep track of their delivery status and ensure their correct arrival order. On the design table, the custom protocol layer sketch started to resemble increasingly similar to the TCP but still held its connectionless nature.

An implementation for packet-tracking was written into a *NetworkPacketTracker* component added to both the server and client sides. The component has three primary roles in network communication. Its first responsibility is tagging outgoing network packets and keeping track of their delivery statuses. The component was also designed to detect dropped packets and handle delivery failures. Its second role included validating the order of incoming packets. If the packets' arrival order differs from the sent order, the *NetworkPacketTracker* has to decide which packets can proceed to further handling and which are dropped. Its third role is to report to a sender host about received packets that have reached their destination successfully.

Sequence number

The added tag property for network packets was called a *SequenceNumber*. The property shares the same naming with a field from the TCP header but represents a unique ID for packets instead of bytes of data. The *SequenceNumber* contains an 8-bit value, from 0 to 255, and is delivered as part of the packet header. The value range was deliberately selected to supply several tracked packets with an identifier without overflowing. When the sequence number reaches its maximum, the value resets to 0. However, using such a small value can cause IDs to overlap when the program wraps around the selected range of numbers. Still, it is safe to use but requires some caution. Sent network packets must occasionally be removed from the packet tracking balanced with the sent rate to release the reserved IDs before the program runs out of available values.

The *NetworkPacketTracker* has to track both outgoing and incoming packets' sequence numbers. So, the component utilizes *OutgoingSequenceNumber* and *ExpectedSequenceNumber* variables. The *NetworkPacketTracker* patches every outgoing network packet with the *OutgoingSequenceNumber* and then increases the variable value by 1 for the next delivery, as shown in the JS code below. The sequence number gives just enough information for the receiving host to check and verify the incoming packet order.

```
if (++outgoing_sequence_number > max_sequence_number)
{
  outgoing_sequence_number = 0;
}
_networkPacket.header.sequence_number = outgoing_sequence_number;
```

When a host receives a network packet, the *NetworkPacketTracker* compares the sequence number of the packet header to the expected value. If the packet has the expected sequence number, the *NetworkPacketTracker* passes the packet to other components to handle and increases the *ExpectedSequenceNumber* by 1. If the value is smaller than expected, the packet gets silently dropped. If the packet contains a sequence number greater than expected, the *NetworkPacketTracker* handles the packet and updates the *ExpectedSequenceNumber* to 1 higher than the packet header has. Therefore, if the *ExpectedSequenceNumber* skips any values, it is safe to assume that at least one prior packet has been dropped. The related server-side *NetworkPacketTracker* method is shown in the following JS code.

```
/**
 * Function checks a given sequence number to validate
 * the correct packet order
 * and adds valid ones into the pending acknowledgments collection
 * @param {number} sequenceNumber
 * @param {string} clientId
 * @param {number} messageType
 * @return {bool} The sequence number is valid and packet handling can proceed
 */
processSequenceNumber(sequenceNumber, clientId, messageType) {
  let isSequenceNumberProcessed = false;
  if (clientId !== UNDEFINED_UUID) {
    const inFlightPacketTrack = this.getInFlightPacketTrack(clientId);
    if (inFlightPacketTrack !== undefined) {
      if (sequenceNumber === inFlightPacketTrack.expectedSequenceNumber) {
        // Successfully received the expected
        inFlightPacketTrack.expectedSequenceNumber = sequenceNumber + 1;
        if (messageType !== MESSAGE_TYPE.ACKNOWLEDGMENT) {
          inFlightPacketTrack.pendingAckRange.push(sequenceNumber);
        }
        isSequenceNumberProcessed = true;
      } else if (
        sequenceNumber > inFlightPacketTrack.expectedSequenceNumber
      ) {
        // Patch to past one of most recent
        ConsoleHandler.Log(
          `Received sequence number ${sequenceNumber} greater
          than expected ${inFlightPacketTrack.expectedSequenceNumber}`
        );
        inFlightPacketTrack.expectedSequenceNumber = sequenceNumber + 1;
        if (messageType !== MESSAGE_TYPE.ACKNOWLEDGMENT) {
          inFlightPacketTrack.pendingAckRange.push(sequenceNumber);
        }
        isSequenceNumberProcessed = true;
      } else if (
        sequenceNumber < inFlightPacketTrack.expectedSequenceNumber
      ) {
        // Drop stale data
        ConsoleHandler.Log(
          `Received sequence number ${sequenceNumber} smaller
          than expected ${inFlightPacketTrack.expectedSequenceNumber}`
        );
        isSequenceNumberProcessed = false;
      }

      if (
        inFlightPacketTrack.expectedSequenceNumber >
        inFlightPacketTrack.maxSequenceNumber
      ) {
        inFlightPacketTrack.expectedSequenceNumber = 0;
      }
    }
  }
  return isSequenceNumberProcessed;
}
```

Acknowledgments

Sequence numbers are associated directly with acknowledgments. Exchanging acknowledgments required additional tweaks to the network packet header structure. The header was expanded with an 8-bit *ACK Count* and varying *ACK Range* fields. The *ACK Count* indicates how many acknowledgment responses the packet contains, while the *ACK Range* field contains related sequence numbers of prior network packets that have reached their destination. It turned out that exchanging a single ACK response at a time caused the ACKs to postpone too much on the real-time network communication, which confirmed that the ACK responses had to be a dynamic collection.

When a host sends a network packet, the local *NetworkPacketTracker* adds the packet to an *InFlightPacket* collection and waits for a response. The receiving host parses the sequence number from the incoming packet and passes the value to the *NetworkPacketTracker* component. The sequence number is then added to a *PendingAckRange* collection, as shown in the JS code block below, where the value remains stored for the next outgoing packet.

```
inFlightPacketTrack.expectedSequenceNumber = sequenceNumber + 1;
if (deliveryPolicy.toInFlightTrack) {
  inFlightPacketTrack.pendingAckRange.push(sequenceNumber);
}
```

When a host sends a new network packet as a response, the *ACK Range* field in the packet header is patched with accumulated sequence numbers from the *PendingAckRange* collection. The method is shown in the following GML code.

```
/// @function     PatchNetworkPacketAckRange(_networkPacket)
/// @description  Patches the ACK Count and ACK Range
///               to the outgoing network packet
/// @param  {struct} networkPacket
/// @return {bool}
static PatchNetworkPacketAckRange = function(_networkPacket)
{
  var isAckRangePatched = false;
  if (_networkPacket.delivery_policy.patch_ack_range)
  {
    if (ds_list_size(pending_ack_range) > 0)
    {
      // CLONE ACK RANGE VALUES
      _networkPacket.header.ack_count = ds_list_size(pending_ack_range);
      ds_list_copy(_networkPacket.header.ack_range, pending_ack_range);
      isAckRangePatched = true;
    } else {
      if (_networkPacket.header.message_type == MESSAGE_TYPE.ACKNOWLEDGMENT)
      {
        global.ConsoleHandlerRef.AddConsoleLog(
          CONSOLE_LOG_TYPE.WARNING,
          "Unnecessary MESSAGE_TYPE.ACKNOWLEDGMENT dropped"
        );
      } else {
        isAckRangePatched = true;
      }
    }
    // PENDING ACK RANGE IS CLEARED AFTER PACKET IS SUCCESSFULLY SENT
  } else {
    // PATCH ACK RANGE SET TO FALSE IN DELIVERY POLICY
    isAckRangePatched = true;
  }
  return isAckRangePatched;
}
```

The destination host then loops through and parses sequence numbers from the *ACK Range* header field. Next, the *NetworkPacketTracker* compares these sequence numbers with the pending acknowledgments in the *InFlightPackets* collection and removes tracked in-flight packets that match these values. The same acknowledgment exchange pipeline repeats the other way around. The described process is shown in the following GML code.

```
/// @function     ProcessAckRange(_ackCount, _ackRange)
/// @description  Loops through a given ACK range
///               and removes matching packets from in-flight tracking
/// @param  {number} ackCount
/// @param  {list} ackRange
/// @return {bool}
static ProcessAckRange = function(_ackCount, _ackRange)
{
  var isAcknowledgmentProceed = true;
  for (var i = 0; i < _ackCount; i++)
  {
    var acknowledgmentId = _ackRange[| i] ?? 0;
    RemoveTrackedInFlightPacket(acknowledgmentId);
  }
  return isAcknowledgmentProceed;
}
```

In-flight packets that never received a matching acknowledgment remain stored in the *InFlightPackets* collection. That was an unwanted case. The issue was fixed by improving the *NetworkPacketTracker* logic to patch the tracked in-flight packets with a running timeout timer, as shown in the GML code below. If a remote host has not responded with an acknowledgment by the set time, it indicates a packet being dropped. The *NetworkPacketTracker* then requests the *NetworkHandler* to retransmit the packet, if necessary, restarts the linked timeout timer, and increases the delivery attempt count by one. If the retransmission fails multiple times, it triggers a safety mechanism that leads to an immediate disconnection.

```
/// @function     PatchInFlightPacketTrack(_networkPacket)
/// @description  Patches the tracked in-flight packet's timeout properties
///               and starts the timeout timer if needed
/// @param  {struct} networkPacket
/// @return {bool}
static PatchInFlightPacketTrack = function(_networkPacket)
{
  var isPacketTrackPatched = false;
  if (_networkPacket.delivery_policy.in_flight_track)
  {
    _networkPacket.timeout_timer.StartTimer();
    ds_list_add(in_flight_packets, _networkPacket);

    if (is_undefined(_networkPacket.ack_timeout_callback_func))
    {
      var consoleLog = string(
        "Network packet with message type {0} is missing ACK timeout callback function",
        _networkPacket.header.message_type
      );
      global.ConsoleHandlerRef.AddConsoleLog(
        CONSOLE_LOG_TYPE.WARNING,
        consoleLog
      );
    }
    isPacketTrackPatched = true;
  } else {
    // IN-FLIGHT PACKET TRACK SET TO FALSE IN DELIVERY POLICY
    isPacketTrackPatched = true;
  }
  return isPacketTrackPatched;
}
```

However, in cases where a host successfully received and handled a network packet but had no outgoing payload data as a response, the *ACK Range* remained pending. The sender was unaware if the packet ever reached the destination and attempted unnecessarily to retransmit the packet all over again. Due to the error, the protocol was enhanced with a new message type for minimal acknowledgment response deliveries.

The *Acknowledgment* packet has an empty payload, and its only purpose is to deliver a pending *ACK Range* via header as a response for a handled message. If the *Acknowledgment* packets queue for sending and the pending *ACK Range* appear empty, the packets are silently removed from the queue, causing no extra network traffic.

During the development, it turned out that packet retransmissions and timeouts were not mandatory with every message type and in certain conditions. For instance, when the server throws an Internal Server Error and broadcasts a related *Error* packet to participants, it is unlikely that the server can respond to further network packets or send acknowledgments. It would be helpful for a client to receive the error message and the remaining *ACK Range* for pending acknowledgments in the same delivery. However, the outcome is equal in both cases where clients disconnect either by a timeout or notification of an Internal Server Error, regardless of which packets get dropped and which acknowledgments are still pending. Therefore, the *Error* packet was implemented to exclude these acknowledgments response requirements. As a further example, a *Ping* packet is another message type that does not require acknowledgments. This is because pinging has only two conditions. It either continues repeating at intervals or times out.

It required a closer inspection to map nearly all possible edge cases and scenarios where a failed packet delivery required retransmission or where the data could be patched afterward. As already might be leaked in the previous code examples, a new property called *DeliveryPolicy* was added to the network packet structure. It was designed to define local tracking and delivery rules for outgoing packets. The *DeliveryPolicy* is like a set of rules containing Boolean values, such as "add to in-flight packet track" and "patch *ACK Range*" flags. These rules instruct the *NetworkPacketTracker* on how the packet should be prepared for sending. This addition provided fully modular and customizable message handling in packet delivery.

7.3 Object replication

One major step in the development was creating schemas for network packet payload data. It would have been a waste of bandwidth to send a piece of data that clients can fetch from their local files or memory in runtime. This is because most assets and data files are locally stored or included in the game build and engine on the client side. Therefore, the focus was to deliver carefully selected and stripped information known as replication data. Replication data on networking is an indirect way to share and map data about game objects and states, independently from gameplay assets and classes (Glazer & Madhav 2015, pages 163). An example case of a replicated player data is illustrated in Figure 12.

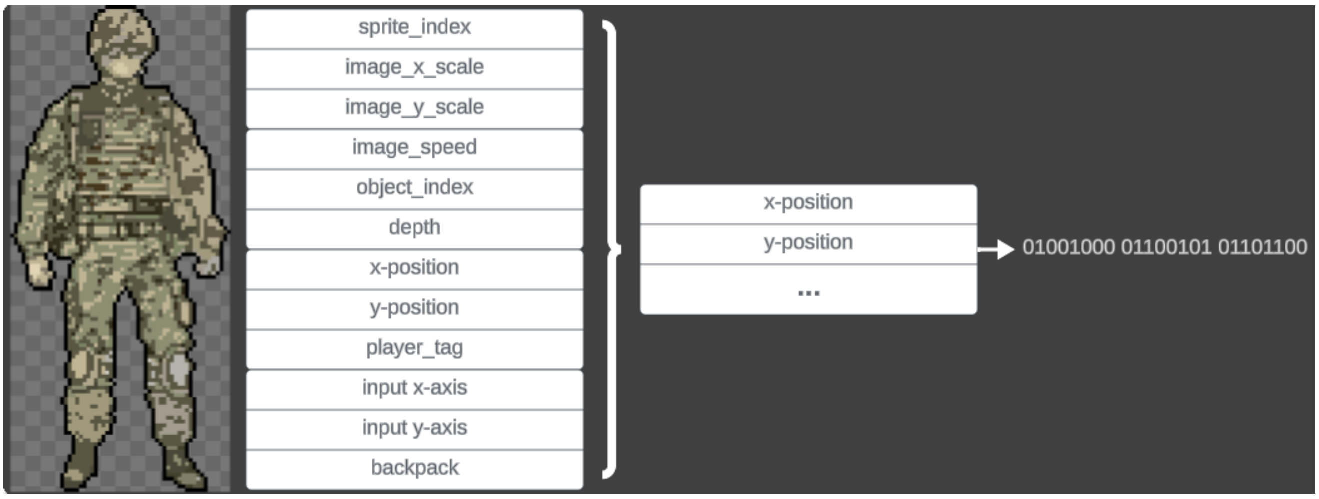


Figure 12. Player object replication

Tracking for networked objects and instances required unique network IDs to bind and associate replication data with in-game objects. The naming convention and the ID value type may vary by use case, but the objective was to keep IDs in sync between hosts. Proper packet processing would have been impossible if packets like *Update*, *Sync*, or *Destroy* contained no network ID indicators of target objects for the actions.

7.4 Data serialization and formatting

JSON is a well-known and standardized format for serializing data for transmissions, especially over HTTP(S). Data is stringified using encoding – like UTF-8 – before being converted into a bit-stream. Each class on the server side and each struct on the client side was enhanced with a helper function called toJSONStruct(). The function creates replicated data from an object by excluding static variables like assets and class/struct methods. It was also designed to format values and uniform character casings in property keys for cohesive naming in networking. An example case of an *Item* class with a described helper method is shown in the following GML code.

```
function Item(_name, _short_name, _icon, _size, _category, _type, _weight,
_max_stack, _base_price, _description, _quantity = 1, _metadata = undefined,
_is_rotated = false, _is_known = true, _grid_index = undefined) constructor
{
  name = _name;
  short_name = _short_name;
  icon = _icon;
  size = _size;
  category = _category;
  type = _type;
  weight = _weight;
  max_stack = _max_stack;
  base_price = _base_price;
  description = _description;
  quantity = _quantity;
  metadata = _metadata;

  is_rotated = _is_rotated;
  is_known = _is_known;
  sourceInventory = undefined;
  grid_index = _grid_index;

  static ToJSONStruct = function()
  {
    var formatMetadata = (!is_undefined(metadata)) ?
    metadata.ToJSONStruct(metadata) : metadata;
    var formatGridIndex = (!is_undefined(grid_index)) ?
    grid_index.ToJSONStruct() : grid_index;
    return {
      name: name,
      quantity: quantity,
      metadata: formatMetadata,
      is_rotated: is_rotated,
      is_known: is_known,
      grid_index: formatGridIndex
    };
  }

  // other methods...
}
```

A single network packet with JSON serialized payload seemed to create only minor network traffic, but the impact was more noticeable in the long run when sampling the bitrate. JSON notation that associates each value with a property name and utilizes key-value pairs requires larger buffer allocation than unmapped and loose values alone. Fortunately, it was possible to omit those property keys in data serialization.

The alternative technique writes only object property values into a buffer one by one, in a specific order and offset. The logic is shown in the GML code blocks below. However, it required accurate and coherent bit-by-bit mapping in buffer read and write operations on both the server and client side. The multiplayer game prototype gained advantages in network optimization by using this technique. Unfortunately, in some cases, structurally varying or complex data was overly complicated to map and write into a buffer value by value. JSON serialization was still relevant in these cases.

```
// ...
case MESSAGE_TYPE.PLAYER_DATA_POSITION:
{
  var playerPosition = _networkPacketPayload; // Already scaled
  buffer_write(_networkBuffer, buffer_u32, playerPosition.X);
  buffer_write(_networkBuffer, buffer_u32, playerPosition.Y);
  isPayloadWritten = true;
} break;
// ...
```

```
// ...
case MESSAGE_TYPE.REMOTE_DATA_POSITION:
{
  var parsedRemotePositionX = buffer_read(_msg, buffer_u32);
  var parsedRemotePositionY = buffer_read(_msg, buffer_u32);
  var parsedRemoteClientId = buffer_read(_msg, buffer_string);
  var parsedRemotePosition = ScaleIntValuesToFloatVector2(
    parsedRemotePositionX, parsedRemotePositionY
  );
  var remoteInstanceData = new InstanceObject(
    object_get_sprite(objPlayer), objPlayer,
    parsedRemotePosition, parsedRemoteClientId
  );
  parsedPayload = remoteInstanceData;
} break;
// ...
```

Value precision

During the research and development, value precisions required more and more attention in data replication to be considered. It was also mentioned and issued by Glazer and Madhav (2015, page 128) in their book:

> *"Further discussion and gameplay testing reveal that client-side positions only need to be accurate to within 0.1 game units. That's not to say that the authoritative server's position doesn't have to be more accurate, but when sending a value to the client, it only needs to do so with 0.1 units of precision."*

Based on that theory, the implementation featured writing the majority of network bitstreams into buffers using rounded integer values. Float values, such as player position, are converted to integers on the application layer by multiplying them by the precision multiplayer value 10. These values are then parsed by subtracting the value by the same precision multiplier. Both scaling functions are shown in the GML code blocks below for clarity. This method allows the server to cache integer-type values and relay them to clients. The value conversions are handled on the client side and are used to run simulations by the GameMaker engine.

```
/// @function       ScaleFloatValueToInt(_value)
/// @description    Converts a provided float value to int
/// @param  {number} value  Number to convert
/// @return {number}
function ScaleFloatValueToInt(_value)
{
  return round(_value * FIXED_POINT_PRECISION);
}
```

```
/// @function       ScaleIntValueToFloat(_value)
/// @description    Converts a provided int value to float
/// @param  {number} value  Number to convert
/// @return {number}
function ScaleIntValueToFloat(_value)
{
  return _value / FIXED_POINT_PRECISION;
}
```

Percentages were an exception. In some cases, the percentages can be scaled by the precision multiplier 1000 but can cause float precision issues in other cases. More precise output values were achieved by utilizing float-type buffers. Integer percentage value conventions were still suitable in some cases when properly used.

Compressing buffers

GameMaker Studio 2 provides a powerful built-in *buffer_compress()* function that waited a long time to be adapted to the prototype. The function utilizes a “zlib” library that offers a simple tool to compress outgoing and decompress incoming bitstream buffers. The corresponding Node.js module, *node:zlib*, was installed into the server project. It turned out that *deflate()* and *deflateSync()* for *compressing and inflate()* and *inflateSync()* for decompressing were the few functions recognized by and compatible with the GameMaker engine. The working module function calls are shown in the following JS code blocks.

```
// ...
// Compress the network buffer
if (networkBuffer !== undefined) {
  compressNetworkBuffer = zlib.deflateSync(networkBuffer);
}
// ...
```

```
// ...
// Decompress message buffer
let msg = zlib.inflateSync(compressMsg);
// ...
```

After brief testing, the compression seemed to have the most effect on larger bitstreams and less noticeable benefits on smaller network packets.

7.5 Connecting to the server

Before a player can join a multiplayer session, the client must initiate the communication by connecting to a server. The game opens a multiplayer connection window (see Figure 13) for the player to fill in server addresses such as IP and port. Meanwhile, the client creates a new UDP socket in the background. Then, it fetches the given server details and sends a connection request. If the request reaches the destination and the server responds with a new unique client ID, a *ClientHandler* component performs the remaining client registration steps. If the connection fails instead, the client attempts again a few times before timing out.

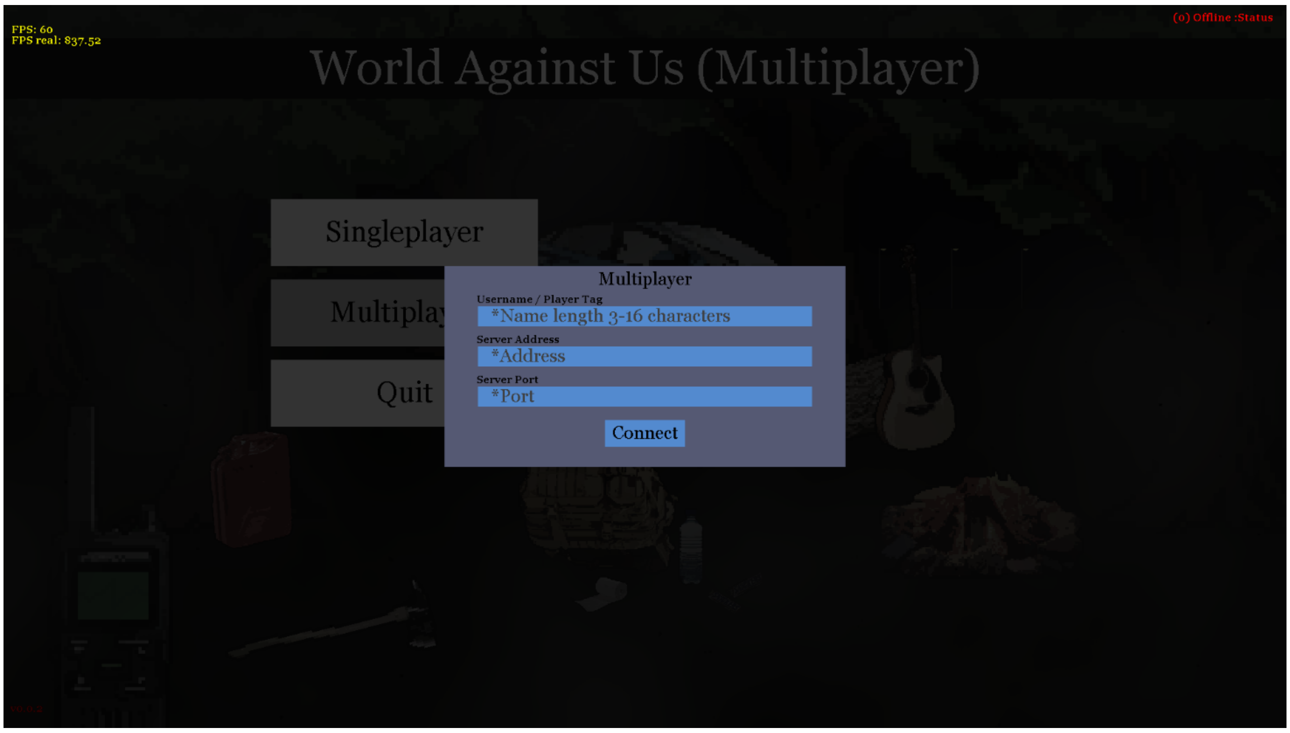


Figure 13. Multiplayer window

With the received ID, the client tags each outgoing network packet so the server can associate these packets with the right player data. However, before the player finds themself spawned into a game world, the custom protocol requires several data exchange and sync steps. The pipeline can be illustrated with the following message types: REQUEST_JOIN_GAME, SYNC_WORLD_STATE, SYNC_PLAYER_DATA, and SYNC_INSTANCE. GameMaker loads local data files, resources, and in-game rooms hand in hand with these steps to properly sync the game state.

The next development step included adapting an existing game save management system from the prior game prototype iteration to networking. When the player selects a character/profile they wish to play, the client sends a REQUEST_JOIN_GAME packet to the server and starts loading local resources. The server then creates a new player object, assigns it to Camp, and responds with related instance info, such as default instance ID, room index, and instance owner.

The following synchronization steps are SYNC_WORLD_STATE and SYNC_PLAYER_DATA. The first of two message types was designed to send world state-related data to the client, with a packet containing current in-game weather and time. The latter message type was created to transmit player-related data that could be useful to be stored on the server. When the server receives a request from a joining client, it patches its local player data, responds with an acknowledgment, and sends a broadcast about a joined player to other participants. Both message types were implemented with scalability in mind to transmit more comprehensive and complex data in the future.

The client has now received enough data to proceed to the final step. GameMaker loads a default in-game room based on the received instance info, and a game handler spawns the player. The client then sends a SYNC_INSTANCE packet to the server during a Room Start event and waits for a response. When the server receives the packet, it fetches instance data, such as local players, from local registries. It then responds with a SYNC_INSTANCE packet and broadcasts a notification to other participants. At this point, the client has successfully gone through the pipeline and connected to the server. For clarity, the whole pipeline, including component dependencies, is illustrated in Figure 14.

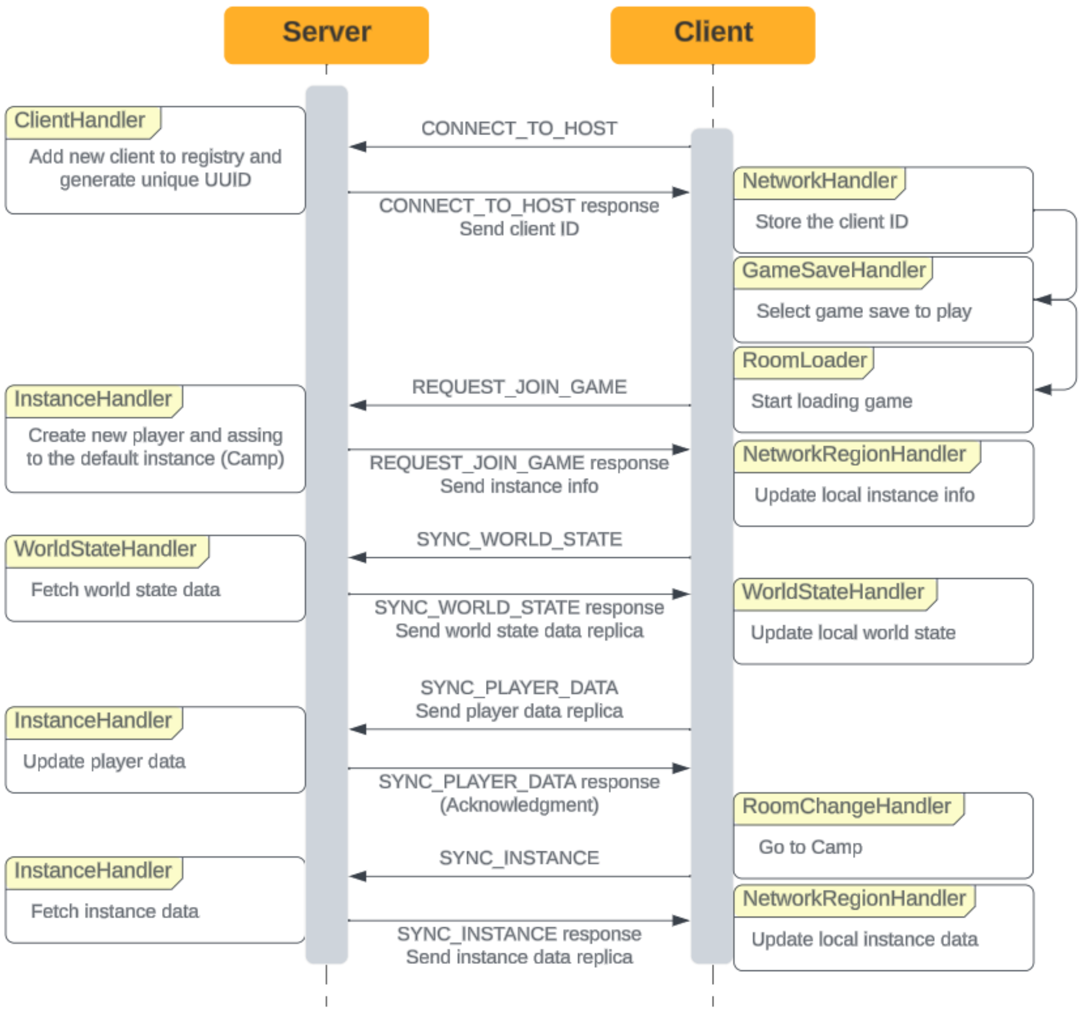


Figure 14. Client connection process

7.6 Game area instances

The designed game area instancing follows a predefined instance hierarchy. In this hierarchy, in-game locations are categorized into parent and sub-instances. While parent instances are accessible from the Camp via the world map, sub-instances are only accessible via fast-travel points from their parent instance. Their main difference comes from their location and game area nature. The current prototype iteration features two world map locations for players to travel. The first available location is a Town, and the second is a Forest. These outdoor game areas represent parent instances that can contain accessible buildings and structures. Entering a building, a player finds themself in an indoor area and has fast traveled from a parent instance to a sub-instance. The Camp instance, in turn, acts like the first and default starting point. The described hierarchy is illustrated in Figure 15.

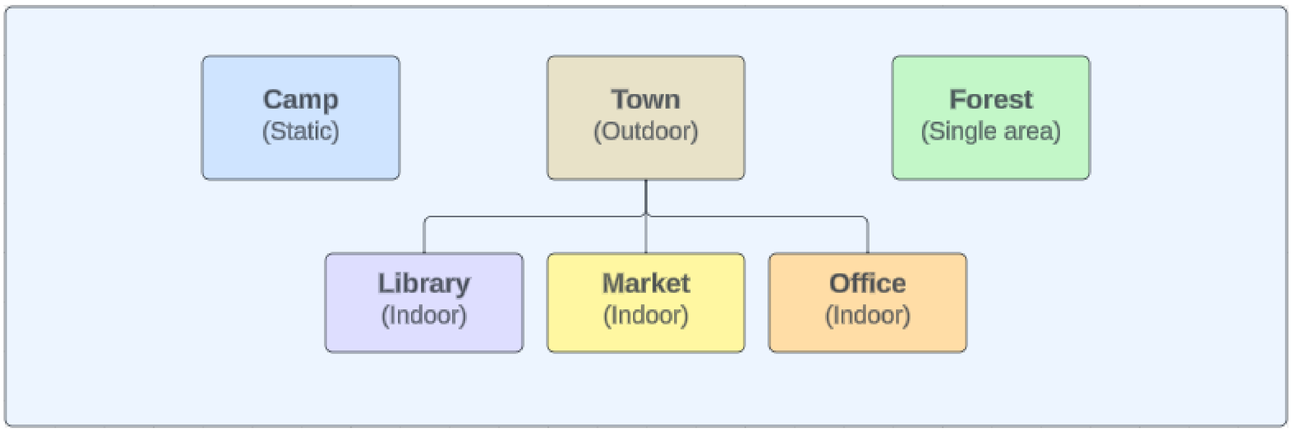


Figure 15. Instance hierarchy

7.6.1 Managing instances

Instance management was built into an *InstanceHandler*. This server-side component was designed to control game area instancing, handle fast traveling, and feature instance-related actions and data queries. The *InstanceHandler* has a registry that stores dynamically created instances and identifies them by unique IDs. Other components like *NetworkHandler* can utilize these functionalities, for instance, in client request handling and data broadcasting.

Owner-client

Because the Node.js server does not have a built-in game engine like GameMaker, it cannot perform simulations on the same level and as precisely as the game clients. Suppose game clients are allowed to perform simulations independently. In this scenario, none can tell if their calculations produce the same results as others because there is no direct communication between the clients in the current network topology. In the long run, the differences between the game states on individual clients would increase and cause unwanted desync. Therefore, each instance with active players also has a named owner-client.

With such limitation, the server can maintain base-level synchronization by employing owner-clients to perform simulations, like NPC behavior, inside each instance and boundaries. Owner-clients are instructed to construct categorized snapshots of their local game state and send them to the server. When the server receives these packets, it broadcasts the snapshots within their target instance boundaries to other participating clients. Those clients can then sync their local game state using the received data. The following JS code block shows an example of how the server receives a snapshot packet from an owner-client, then updates its local data and forwards the snapshot data via broadcast.

```
// ...
case MESSAGE_TYPE.PATROLS_SNAPSHOT_DATA:
{
  const patrolsSnapshotData = networkPacket.payload;
  if (patrolsSnapshotData !== undefined) {
    if (patrolsSnapshotData.instanceId === instance.instanceId) {
      if (patrolsSnapshotData.localPatrols.length > 0) {
        patrolsSnapshotData.localPatrols.forEach(
          (patrolSnapshotData) => {
            const patrol = instance.getPatrol(
              patrolSnapshotData.patrolId
            );
            if (patrol !== undefined) {
              patrol.position.x = patrolSnapshotData.position.x;
              patrol.position.y = patrolSnapshotData.position.y;
              patrol.routeProgress = patrolSnapshotData.routeProgress;
              patrol.routeTime =
                patrol.totalRouteTime -
                patrol.totalRouteTime * patrol.routeProgress;
            }
          }
        );
      }

      // Broadcast patrols snapshot data withing instance
      const clientsToBroadcast =
        this.clientHandler.getClientsToBroadcastInstance(
          instance.instanceId,
          client.uuid
        );
      const broadcastNetworkPacketHeader = new NetworkPacketHeader(
        MESSAGE_TYPE.PATROLS_SNAPSHOT_DATA,
        client.uuid
      );
      const broadcastNetworkPacket = new NetworkPacket(
        broadcastNetworkPacketHeader,
        patrolsSnapshotData,
        PACKET_PRIORITY.DEFAULT
      );
      this.networkHandler.broadcast(
        broadcastNetworkPacket,
        clientsToBroadcast
      );

      // Patrol snapshot data is routine
      // and only the latest message takes effect
      // No guarantee for delivery required
      isPacketHandled = true;
    }
  }
}
break;
// ...
```

When a named owner-client leaves their instance, the server checks for new owner availability. The *InstanceHandler* first checks if the instance still has local players; a new owner-client is picked among them. The related *Instance* class method is shown in the JS code block below. When the owner-client changes at one point or another, the server broadcasts a SYNC_INSTANCE_OWNER network packet within the instance, providing new owner details to clients. The server ignores owner assignments if the instance appears empty. Instances without active local players are set to an idling mode or keep performing necessary calculations on the simulation layer.

```
/**
 * Resets the owner client
 * @return {bool} Owner successfully reset
 */
resetOwner() {
  let isOwnerReset = false;
  if (this.getPlayerCount() > 0) {
    const playerId = this.getPlayerIdFirst();
    if (playerId !== undefined) {
      this.setOwner(playerId);
      isOwnerReset = true;
    } else {
      this.setOwner(undefined);
      isOwnerReset = true;
    }
  } else {
    this.setOwner(undefined);
    isOwnerReset = true;
  }
  return isOwnerReset;
}
```

The owner-client role was designed to patch the server's incapabilities and provide assistance in defining the final and synchronized global game state. With snapshot broadcasting, the server can ensure consistency between clients' local game states and minimize potential desynchronization. Nonetheless, the server still holds the privilege and authority over the global world state even though it needs to trust client-side authority in some cases temporarily.

Instance hierarchy

Each instance must have a valid room index that matches the name mapping on both the server and client sides. Room indices are string values associated with the numeric GameMaker room_index variables. In the current implementation, they also define positions in the parent-child instance hierarchy. Because GameMaker uses numeric room_index values, rearranging the game's room load order also changes these numbers, which would require room index remapping each time. In this case, strings are more static indicators for room indices.

An additional *parentInstanceId* property was added to the instance data structure to link relations between the instances in the parent-child hierarchy. The property also indicates if the instance has a parent. Each created instance is assigned to its parent, if it has one, using a *setInstanceParenthood()* method call, as shown in the following JS code.

```
/**
 * Sets dependencies between given child and parent instances
 * and validates existence of the given room index in the instance hierarchy
 * @param {string} childInstanceId
 * @param {string} roomIndex
 * @param {string} parentInstanceId
 * @return {void}
 */
setInstanceParenthood(childInstanceId, roomIndex, parentInstanceId) {
  const parentInstance = this.getInstance(parentInstanceId);
  const parentInstanceHierarchy =
    WORLD_MAP_LOCATION_HIERARCHY[parentInstance.roomIndex];
  if (Object.keys(parentInstanceHierarchy).includes(roomIndex)) {
    const childInstance = this.getInstance(childInstanceId);
    if (childInstance !== undefined) {
      childInstance.parentInstanceId = parentInstanceId;
    }
  }
}
```

Static zones

Dividing the world map and game areas into static zones for the multiplayer mode was already half-done and implemented during the prior single-player mode development. Because rooms in the single-player mode were already small or medium-sized, the networking did not require drawing for new static zones. In an unpleasant case, it would have required more development time to create new virtual boundaries to divide the game world into static zones, like in games with massive open-world maps.

The server-side network packet transmitting was optimized by utilizing global and scoped broadcasting. Preset zone boundaries help the server choose which information is relevant to individual clients. If two players scavenge and travel in different instances, they do not necessarily need to receive updates outside their local boundaries. The *ClientHandler* component provides several query methods (shown in the JS code block below) to fetch client ID lists depending on the selected scope.

```
getClientsToBroadcastGlobal(excludeClientId = UNDEFINED_UUID) {
  return this.getAllClients().filter((client) => {
    return client.uuid !== excludeClientId;
  });
}

getClientsToBroadcastInGame(excludeClientId = UNDEFINED_UUID) {
  return this.getAllClients().filter((client) => {
    return client.instanceId !== undefined && client.uuid !== excludeClientId;
  });
}

getClientsToBroadcastInstance(instanceId, excludeClientId = UNDEFINED_UUID) {
  return this.getAllClients().filter((client) => {
    return (
      client.instanceId === instanceId && client.uuid !== excludeClientId
    );
  });
}
```

Components like *InstanceHandler* can use these *ClientHandler* class methods to target broadcasting to desired client groups. Furthermore, preset static zones also helped optimize memory usage and utilize selective rendering.

Client-side game state

A *NetworkRegionHandler* component was added to the client side to control instance-related data. For clarity, its name differs from the server-side *InstanceHandler* component because GameMaker reserves the term “instance” for object instances. The *NetworkRegionHandler* has an essential role in the local game state patching when it receives Sync, Data, and Destroy requests from the *NetworkHandler*.

The logic was distributed further into smaller helper components, such as *NetworkRegionRemotePlayerHandler*, *NetworkRegionObjectHandler*, and *NPCHandler*. Each component has caches for specific networked instance object types and implementation for related *Sync*, *Data*, and *Destroy* methods. They were designed to perform actions on target in-game characters, props, and objects and track different objects’ states. The following GML code block shows an example of a *NetworkRegionRemotePlayerHandler* method that updates remote players’ positions.

```
/// @function     UpdateRegionRemotePosition(_remoteInstanceObject)
/// @description  Updates remote player's position inside the local instance
///               using provided instance object data
/// @param  {struct} remoteInstanceObject
/// @return {bool}
static UpdateRegionRemotePosition = function(_remoteInstanceObject)
{
  var isPositionUpdated = false;
  if (!is_undefined(_remoteInstanceObject))
  {
    var remotePlayer = GetRemotePlayer(_remoteInstanceObject.network_id);
    if (!is_undefined(remotePlayer))
    {
      var positionThreshold = 50;
      var distance = point_distance(
        _remoteInstanceObject.position.X,
        _remoteInstanceObject.position.Y,
        remotePlayer.position.X,
        remotePlayer.position.Y
      );
      if (distance > positionThreshold)
      {
        var playerInstanceRef = remotePlayer.instance_ref;
        if (instance_exists(playerInstanceRef))
        {
          remotePlayer.position.X = playerInstanceRef.x;
          remotePlayer.position.Y = playerInstanceRef.y;
          remotePlayer.start_position = remotePlayer.position;
          remotePlayer.StartInterpolateMovement(
            _remoteInstanceObject.position,
            50
          );
        }
      }
      isPositionUpdated = true;
    }
  }
  return isPositionUpdated;
}
```

The *NetworkRegionHandler* component also keeps track of the current instance ID and owner, room index, and previous instance ID. Without a stored prior instance ID value, the game cannot map the path between instances that a player has traveled. For example, entering a building causes GameMaker to load a new room, a new sub-instance, and erase the old room from memory. Later, when the player desires to exit that building, the game redirects them to a different instance – unexpectedly finding themselves a bit off from the original starting point. This can occur if the *NetworkRegionHandler* has no records of the traveled path.

7.6.2 Fast travel

Fast travel is a popular method in video games to move game characters from one location to another. Games can let players travel from their standing point, offer a UI world map, or require players to find a specific spot like a wayshrine in the game world. The fast travel system was enhanced with a UI world map and available instances list (see Figure 16). In the prior prototype iteration, players could only fast travel via interactable fast travel points in several game world locations. The fast travel system was adapted to the networking by moving the authority from the client to the server.

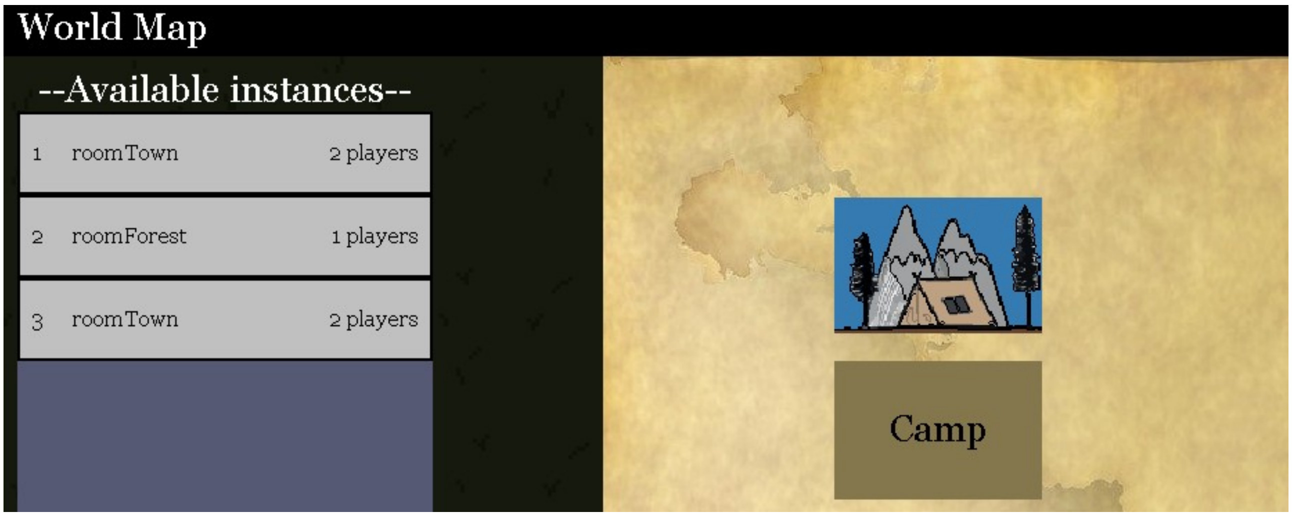


Figure 16. World map UI

Fast traveling in the multiplayer mode was designed to utilize several new protocol steps, while the process was more straightforward in the single-player mode. In the first step, a player selects a desired destination using fast travel points or a UI map, which triggers the client to send a REQUEST_FAST_TRAVEL packet to the server. The related request method is shown in the following GML code.

```
/// @function       RequestFastTravel(_fastTravelInfo)
/// @description    Requests room change in singleplayer mode
///                 or sends a fast-travel request to the server
/// @param  {struct} fastTravelInfo
/// @return {void}
static RequestFastTravel = function(_fastTravelInfo)
{
  if (!global.MultiplayerMode)
  {
#region Offline
    if (RequestRoomChange(_fastTravelInfo.destination_room_index))
    {
      if (!RequestCacheFastTravelInfo(_fastTravelInfo))
      {
        global.ConsoleHandlerRef.AddConsoleLog(
          CONSOLE_LOG_TYPE.ERROR,
          string(
            "Unable to fast travel to room '{0}'",
            _fastTravelInfo.destination_room_index
          )
        );
      }
    }
#endregion
  } else {
#region Multiplayer
    var guiState = new GUIState(
      GUI_STATE.WorldMapFastTravelQueue, undefined, undefined,
      [
        CreateWindowWorldMapFastTravelQueue(
          GAME_WINDOW.WorldMapFastTravelQueue, -1
        )
      ],
      GUI_CHAIN_RULE.OverwriteAll, undefined, undefined
    );
    if (global.GUIStateHandlerRef.RequestGUIState(guiState))
    {
      // REQUEST FAST TRAVEL
      var networkPacketHeader = new NetworkPacketHeader(MESSAGE_TYPE.RE-
QUEST_FAST_TRAVEL);
      var networkPacket = new NetworkPacket(
        networkPacketHeader,
        _fastTravelInfo,
        PACKET_PRIORITY.DEFAULT,
        AckTimeoutFuncResend
      );
      if (global.NetworkHandlerRef.AddPacketToQueue(networkPacket))
      {
        global.PlayerCharacter.is_fast_traveling = true;

        // DEBUG MONITOR
        global.DebugMonitorMultiplayerHandlerRef.StartFastTravelTimeSampling();
      } else {
        show_debug_message("Failed to request fast travel");
      }
    }
  }
#endregion
}
```

When the server receives the request, it parses the packet and checks that the destination value points to an existing instance ID or, optionally, to a room index. The *InstanceHandler* component either creates a new instance on demand based on the provided room index or transfers the player between two existing instances. The following JS code block shows this server-side logic.

```
/**
 * Fast travels a player from source instance to destination,
 * and validates provided fast travel details
 * @param {string} clientId
 * @param {string} sourceInstanceId
 * @param {string} destinationRoomIndex
 * @param {string} destinationInstanceId
 * @return {number} New instance ID
 */
fastTravelPlayer(
  clientId,
  sourceInstanceId,
  destinationRoomIndex,
  destinationInstanceId
) {
  let newInstanceId;
  const sourceInstance = this.instances[sourceInstanceId];
  if (sourceInstance !== undefined) {
    const player = sourceInstance.getPlayer(clientId);
    if (player !== undefined) {
      if (this.isRoomIndexValid(destinationRoomIndex)) {
        if (destinationRoomIndex === ROOM_INDEX.ROOM_CAMP) {
          if (this.removePlayerFromInstance(clientId, sourceInstanceId)) {
            newInstanceId = this.addPlayerToDefaultInstance(clientId, player);
          }
        } else {
          let priorityInstanceId =
            sourceInstanceId === destinationInstanceId
              ? undefined
              : destinationInstanceId;
          // This logic supports only 1-level of parent-child hierarchy
          // No sub-instances with sub-instances
          if (sourceInstance.parentInstanceId === undefined) {
            if (priorityInstanceId === undefined) {
              this.getInstanceIds().forEach((instanceId) => {
                const instance = this.getInstance(instanceId);
                if (
                  instance.parentInstanceId === sourceInstanceId &&
                  instance.roomIndex === destinationRoomIndex
                ) {
                  priorityInstanceId = parseInt(instanceId);
                }
              });
            }
          }
          newInstanceId = this.addPlayerToInstance(
            clientId,
            destinationRoomIndex,
            player,
            priorityInstanceId
          );
          if (newInstanceId !== undefined) {
            // Set instance hierarchy parenthood for new instances
            if (priorityInstanceId === undefined) {
              this.setInstanceParenthood(
                newInstanceId,
                destinationRoomIndex,
                sourceInstanceId
              );
            }
            if (this.removePlayerFromInstance(clientId, sourceInstanceId)) {
              // Reset player position
              player.resetPosition();
            }
          }
        }
```

After the server has patched its player and instance registries, it responds to the client with a fast travel output state via a REQUEST_FAST_TRAVEL packet. The response contains the patched source instance ID, destination instance ID, and destination room index. GameMaker can then load a new room on the client side using the received information and transfer the player character to the requested instance.

In the final step, the client sends a SYNC_INSTANCE packet to the server. The server then requires each participant to synchronize their local instance states using received broadcast data and acknowledge the changes.

7.6.3 Instance life cycle

The instance creation process was built with dynamic behavior and modularity in mind. The *InstanceHandler* does not automatically allocate memory for associated child instances during a parent creation process. Instead, child instances are created only on demand. This approach shares similarities with lazy loading, where data is fetched when needed. It comes with advantages in terms of processing time and memory usage. The related *InstanceHandler* method is shown in the following JS code.

```
/**
 * Creates a new instance from a given room index,
 * validates the room index, and adds the created instance to registry
 * @param {string} roomIndex
 * @return {number} Created instance ID
 */
createInstance(roomIndex) {
  let createdInstanceId;
  if (this.isRoomIndexValid(roomIndex)) {
    const createdInstance = new Instance(
      this.nextInstanceId,
      roomIndex,
      this.networkHandler
    );
    createdInstanceId = this.nextInstanceId;
    this.instances[createdInstanceId] = createdInstance;
    this.nextInstanceId++;
  }
  return createdInstanceId;
}
```

After a boot-up, the server remains idle without a single initialized instance while empty. When the first client connects, the server creates a default Camp instance with a predefined ID and, from now on, executes various update calls on every tick. The default instance creation method is shown in the following JS code.

```
/**
 * Creates default Camp instance
 * with the default instance ID and storage container
 * @return {bool} Instance is successfully created
 */
createDefaultCampInstance() {
  let isInstanceCreated = false;
  if (this.getInstance(this.campId) === undefined) {
    const campInstance = new Instance(
      this.campId,
      ROOM_INDEX.ROOM_CAMP,
      this.networkHandler
    );
    campInstance.containerHandler.addContainer(CAMP_STORAGE_CONTAINER_ID);
    this.instances[this.campId] = campInstance;
    isInstanceCreated = true;
  }
  return isInstanceCreated;
}
```

The *Instance* class has an update method that executes on every server tick. Instances were designed to perform different actions and simulations. They also hold registries for networked objects like players, containers, and patrols. The update method (shown in the JS code block below) is simple and calls just a patrol behavior update method. While still easily expandable in the future. Whenever a player position changes or an owner-client delivers a snapshot, the global state and data are patched in these registries on the server side. The data is then easily accessible when clients request instance sync actions or when the data is scheduled for broadcasting.

```
/**
 * Called on every server tick
 * @param {number} passedTickTime
 * @return {bool} Updated
 */
update(passedTickTime) {
  let isUpdated = true;
  if (this.patrolRoute !== undefined) {
    isUpdated = this.updateLocalPatrols(passedTickTime);
  }
  return isUpdated;
}
```

When player characters leave a game area empty, the *InstanceHandler* performs several condition checks on the related instance. The handler decides to delete the instance either immediately or later. Because instances are mapped into a hierarchy, child instances cannot exist without a parent. Therefore, they are deleted recursively with their parent. The *InstanceHandler* was designed to store child instances in a registry, even empty ones, in case players revisit them. If a sub-instance, like an indoor office, is deleted too early, revisiting the game area from the same parent instance would redirect a player to a new instance instead of the original, deleted one.

If a parent instance, excluding the Camp, becomes empty of local players, does not have a parent of its own, and all its child instances appear empty, the handler deletes them with a *deleteInstance()* method (see the JS code below). However, if one or more child instances appear occupied, the parent instance remains performing update routines. Without active players, an instance cannot request its owner-client to perform instance-related simulations nor receive player data updates for broadcasting. Instead, these empty instances live on the simulation layer.

```
/**
 * Deletes an instance and performs clean-up
 * @param {string} instanceId
 * @return {bool} Instance deleted successfully
 */
deleteInstance(instanceId) {
  let isInstanceDeleted = false;
  var instance = this.getInstance(instanceId);
  if (instance !== undefined) {
    this.onInstanceDelete(instance);

    delete this.instances[instanceId];
    isInstanceDeleted = true;
  }
  return isInstanceDeleted;
}
```

7.7 Simulation layer

Upon discovery that certain multiplayer game elements require processing in the background, independently from networking, clients, and the GameMaker engine, a new framework called the *Simulation layer* was introduced. The simulation layer was formed from a handful of embedded components and designed to model several game element behaviors and provide global and instance-level simulations. These simulations were specially built to run on the server side, under its full authority.

Day-Night cycle

In many video games, the virtual world lives in an imaginary universe and time. Players can adventure in a world where time runs in the day-night cycle, making the universe and environment more believable. The developed prototype also followed these popular practices.

The simulation layer was implemented to model in-game time and day-night cycles. In-game time is calculated on every server tick using delta time, allowing the server to synchronize multiple timers and simulations in distributed components. In-game time is scaled by using a default 24:1 timescale ratio. In other words, one day in the game is equivalent to one past real-time hour. The timescale was also designed to be safely adjustable without breaking the game loop.

In-game time and configured timescale are synchronized between the server and clients. The current time value and properties are written into a *Sync* packet, sent to each new client during the client registration process, and checked occasionally afterward. The in-game time value is then rendered on the client side onto the UI using a numeric clock. Day-night cycles and light-darkness transitions, in turn, are illustrated by changing color palettes and using filters and shaders. Sky light level is calculated using in-game time and preset dark hours edge values. An example of client-side lighting is shown in Figure 17.

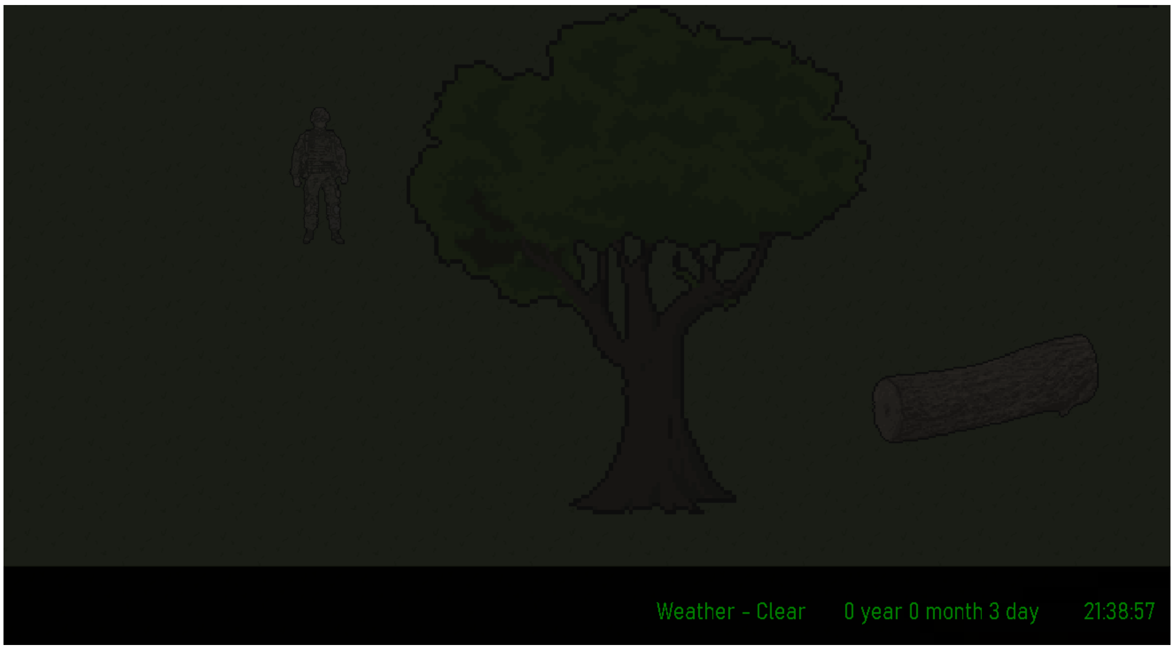


Figure 17. In-game dusk

Weather

The global in-game weather condition is calculated on the simulation layer under the server's authority. The prevailing weather condition is linked with in-game time and rerolled (see the JS code below) in predefined intervals.

```
/**
 * Rolls a new weather condition
 * @return {bool} Weather rolled successfully
 */
rollWeather() {
  let isWeatherRolled = false;
  const keys = Object.keys(WEATHER_CONDITION);
  const randomKey = keys[(keys.length * Math.random()) << 0];
  const newWeatherCondition = WEATHER_CONDITION[randomKey];
  if (this.weather !== newWeatherCondition) {
    this.weather = newWeatherCondition;
    isWeatherRolled = this.networkHandler.broadcastWeather(this.weather);
  } else {
    isWeatherRolled = true;
  }
  return isWeatherRolled;
}
```

The resulting condition is presented by an Enum index value and broadcast in a *Sync* packet to clients. On the client side, the local weather condition is synchronized and visually changed based on the received Enum index value, as shown in the following GML code.

```
/// @function     SetWeather(_weather)
/// @description  Set a new weather condition
///               and updates visual effects
/// @param  {number} weather
/// @return {bool}
static SetWeather = function(_weather)
{
  var isWeatherSet = false;
  if (weather != _weather)
  {
    // CLEAR CURRENT WEATHER
    var currentFxLayerName = string(
      "{0}{1}",
      LAYER_EFFECT_PREFIX,
      array_get(WEATHER_TEXT, weather)
    );
    var currentFxLayerId = layer_get_id(currentFxLayerName);
    if (layer_exists(currentFxLayerId))
    {
      if (layer_fx_is_enabled(currentFxLayerId))
      {
        layer_enable_fx(currentFxLayerId, false);
      }
    }
    // SET NEW WEATHER
    weather = _weather;
    UpdateWeatherEffect();
    isWeatherSet = true;
  } else {
    // NO CHANGES
    isWeatherSet = true;
  }
  return isWeatherSet;
}
```

The new weather condition is then visualized on the client side using corresponding color pallets, shaders, and filters (see Figure 18). For instance, the in-game scene has a layer of white floating fog on a foggy day and a blueish and chilly atmosphere during rain.

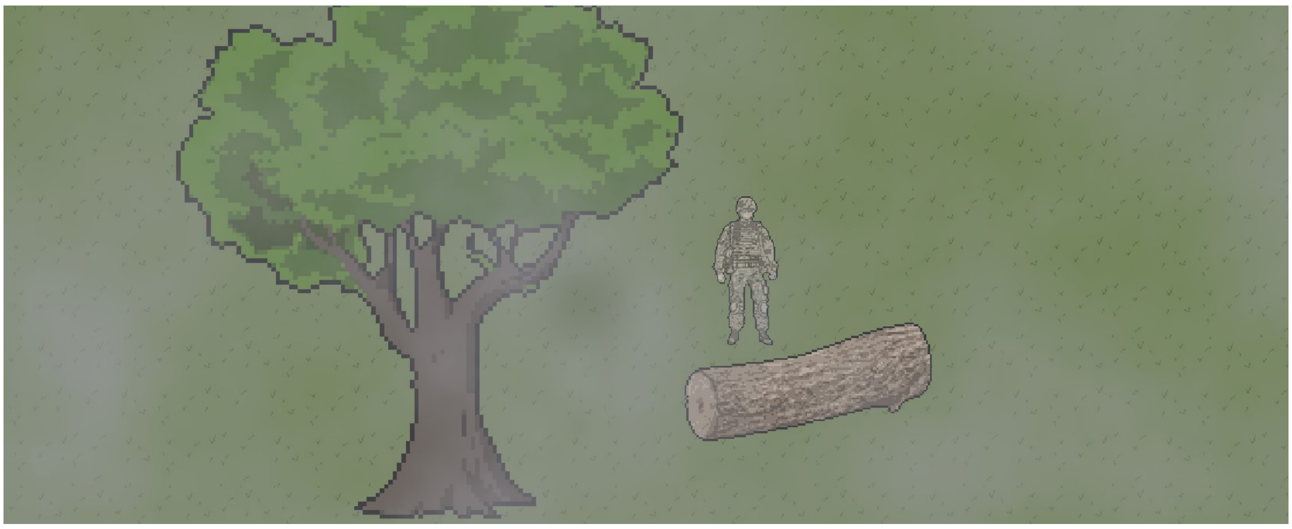


Figure 18. In-game fog

7.7.1 World persistence

After the developed prototype iteration reached the point where multiplayer logic required the server to run simulations in the background and carry that data from one session to another, it was time to build world persistence components onto the simulation layer.

In a virtual world experience, world persistence indicates how the virtual environment lives and behaves independently from the players' presence. There are three different approaches to building persistence. The first of three resets a virtual world to its initial condition for every new session. When participants enter the virtual world, they find the world new and pure without any sign of previous player activity or past time. (Sherman & Craig 2003, page 405.)

The second approach records the world state and preserves its condition from one session to another. With this persistence level, participants can continue their virtual journey and progress where they were left, experiencing the virtual world like an ongoing story. However, this setup requires a place like a database to store the world state at the end of each session and a way to load the data when a new session begins. (Sherman & Craig 2003, page 405.)

The third option maintains continuous simulations and makes a virtual world fully persistent. Without presented participants, time passes in the world, and elements, like growing vegetables, continue their simulated cycle. (Sherman & Craig 2003, page 405.)

The multiplayer mode was designed to load the latest game progress on each new session. In other words, the game was meant to continue from the same state and condition it was left on each server shutdown. Such a feature requires storing world state attributes like time and date, time scale, and a weather condition in a safe place at the end of each session. This was solved using server-side save files, as introduced in the "Planning and design" section. These save files provided a way for the server to fetch the latest world state of a saved multiplayer session.

A server-side save file contains world state data, separate from client-side player data. The data is written in JSON format in the runtime once every 10 minutes on an autosave action and during the last client disconnection. The save file content structure was designed to provide enough data for the server to continue simulations at the start of each game session.

The multiplayer mode logic was designed to reset players back to the Camp when they join the game. Therefore, from a gameplay progression perspective, there was no reason to store data about dynamically created instances. That is because instances only hold temporal networked game objects and are flushed by the end of each session.

7.7.2 Enemy patrols

Enemy behavior in the prior prototype iteration was programmed using a finite hierarchical state machine. The model was chosen because of its modularity, allowing smaller sub-state machines to be embedded into the hierarchy. At this point in development, AI behavior was simple-structured and included a few AI states such as *Travel*, *Patrol*, *Chase*, *Partol Return*, and *Patrol End*.

GameMaker Studio 2 offers a room editor tool for drawing different pathways onto the ground tiles, as illustrated in Figure 19 by light blue lines. Each path contains a set of coordination points that form either a loop or a finite path with a start and end point. Game objects can then be programmed to follow these drawn paths with built-in AI logic. The engine also features alternative methods for dynamically generated AI navigation in runtime. It was finally time to adapt the AI logic to networking.

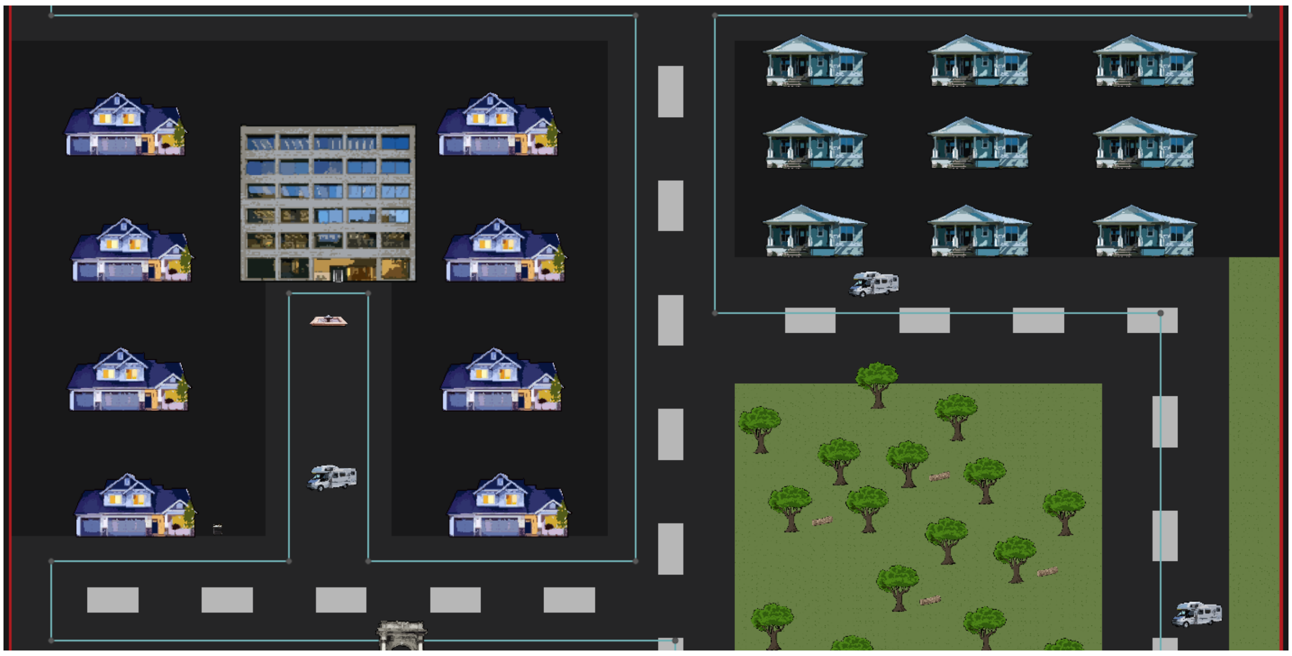


Figure 19. In-game AI pathing

When instances are created, the ones with a predefined patrol route will initiate a random number of incoming enemy patrols (see the JS code block below). At first, these enemy patrols are set to the *Travel* state, each with a ticking timer. They live on the simulation layer during the traverse until they reach their destination after the timer expires.

```
// ...
const localPatrolIds = this.getAllPatrolIds();
if (localPatrolIds.length <= 0) {
  const randomPatrolCount = GetRandomIntFromRange(
    this.patrolRoute.minPatrolCount,
    this.patrolRoute.maxPatrolCount
  );
  const randomTravelTime = GetRandomIntFromRange(
    this.patrolRoute.minTravelTime,
    this.patrolRoute.maxTravelTime
  );
  for (let i = 0; i < randomPatrolCount; i++) {
    const newPatrol = new Patrol(
      this.availablePatrolId,
      this.instanceId,
      this.patrolRoute.routeTime,
      randomTravelTime
    );
    this.addPatrol(newPatrol);
  }
  isPatrolsUpdated = true;
}
// ...
```

After arrival, the AI state changes from *Travel* to *Patrol*, and the *InstanceHandler* writes the replicated patrol state into a *Sync* packet and broadcasts it within the current instance. From that point, depending on the instance state, the AI behavior is calculated on the simulation layer under the server's authority or outsourced to a named owner-client.

Because the Node.js server cannot access GameMaker functions, it can either utilize the simulation layer and simpler AI models or request help from a current instance owner-client. In cases where the current instance appears empty of players and has no named owner-client, the AI behavior remains on the simulation layer. Server-side AI models were designed to run independently from networking and clients. They are lightweight and utilize only timers and approximated route time values instead of 2D physics and complex AI methods. The simulation continues until local patrols reach their route endpoint and leave the game area, the current instance is destroyed, or when a player enters the instance.

In another case, the owner-client controls the patrols inside an instance. The GameMaker client utilizes built-in engine methods and 2D physics for more precise and complex AI behavior. With such capabilities, clients can, for instance, run base game AI simulations where enemy bandits can deviate from their current route path and start threading players by chasing them. The Node.js server lacks such modules, and building a replica for a similar implementation on the server side would require plenty of additional development hours. Even though the owner-client has full authority over local patrols within a current instance, it is responsible for sending related patrol snapshot *Data* packets to the server. The following GML code block is snipped from a *NetworkRegionObjectHandler*, showing client-side logic on how owner-clients provide patrol snapshot data for the server.

```
/// @function     Update()
/// @description  Executed on every game loop
static Update = function()
{
  if (IS_ROOM_PATROL_ROUTED)
  {
    if (global.NetworkRegionHandlerRef.IsClientRegionOwner())
    {
      // PATROL UPDATE
      patrol_update_timer.Update();
      if (patrol_update_timer.IsTimerStopped())
      {
        // FETCH PATROL SNAPSHOT DATA
        var patrolNetworkIDs = global.NPCPatrolHandlerRef.GetPatrolNetworkIDs();
        var patrolCount = array_length(patrolNetworkIDs);
        if (patrolCount > 0)
        {
          var formatPatrols = [];
          for (var i = 0; i < patrolCount; i++)
          {
            var patrol = global.NPCPatrolHandlerRef.GetPatrol(
              patrolNetworkIDs[@ i]
            );
            if (!is_undefined(patrol))
            {
              var patrolSnapshot = new PatrolSnapshot(
                patrol.patrol_id,
                patrol.route_progress,
                patrol.position
              );
              array_push(formatPatrols, patrolSnapshot.ToJSONStruct());
            }
          }
          // SEND PATROL SNAPSHOT NETWORK PACKET
          var networkPacketHeader = new NetworkPacketHeader(
            MESSAGE_TYPE.PATROLS_SNAPSHOT_DATA
          );
          var networkPacket = new NetworkPacket(
            networkPacketHeader,
            {
              region_id: global.NetworkRegionHandlerRef.region_id,
              local_patrols: formatPatrols
            },
            PACKET_PRIORITY.DEFAULT,
            undefined
          );
          if (!global.NetworkHandlerRef.AddPacketToQueue(networkPacket))
          {
            global.ConsoleHandlerRef.AddConsoleLog(
              CONSOLE_LOG_TYPE.WARNING,
              "Failed to queue patrol snapshot network packet"
            );
          }
        }
        // RESTART PATROL UPDATE TIMER
        patrol_update_timer.StartTimer();
      }
    }
// ...
```

7.7.3 Operations Center

The operations center is an interactable game object in the Camp, featuring a common UI interface with a real-time map view. It seemed like an ordinary client-side game object in the first place because the logic was simple and straightforward in the single-player mode. However, the network adaptation required implementing more complex building blocks and additional mechanisms under the hood.

In the prior prototype iteration, the operations center was created to provide a monitoring tool for a player to scout outdoor game areas remotely in single-player mode. It helped solo players avoid dangerous enemies by revealing patrol locations on the UI map view without requiring players to enter the scouted game area. However, the multiplayer mode implementation required heavy refactoring and changes to adapt these elements to networking.

In all its simplicity, the monitoring tool uses color-coded map markers to draw a miniature-like map view from a selected game area. First, a *MapDataHandler* loads data like in-game positions, sizes, and asset types from pre-generated map data files for static room elements. The related *MapDataHandler* method is shown in the following GML code.

```
/// @function       ReadStaticMapDataFromFile(_fileName)
/// @description    Reads static map data from a target file
///                 and overwrites current map data
/// @param  {string} fileName File name (without a path)
/// @return {boolean}
static ReadStaticMapDataFromFile = function(_fileName)
{
  var isMapDataReaded = false;
  var formatFileName = string("{0}{1}", "/map_data/", _fileName);
  var staticMapDataStruct = ReadJSONFile(formatFileName) ?? EMPTY_STRUCT;
  var parsedMapData = ds_list_create();
  ParseJSONStructToList(
    parsedMapData, staticMapDataStruct[$ "icons"] ?? undefined,
    ParseJSONStructToMapIcon
  );
  // DESTROY PREVIOUS ICONS DS LIST
  static_map_data.OnDestroy();
  // UPDATE AND SORT ICONS
  static_map_data.icons = parsedMapData;
  static_map_data.SortIcons();
  isMapDataReaded = true;
  return isMapDataReaded;
}
```

Objects like NPCs and game characters, in turn, appear on the map view as dynamic map markers. Map icon styles are fetched from other data files on resource loading during the game boot-up. Finally, the player can inspect game areas on the map view by controlling a virtual flying scouting drone (see Figure 20).

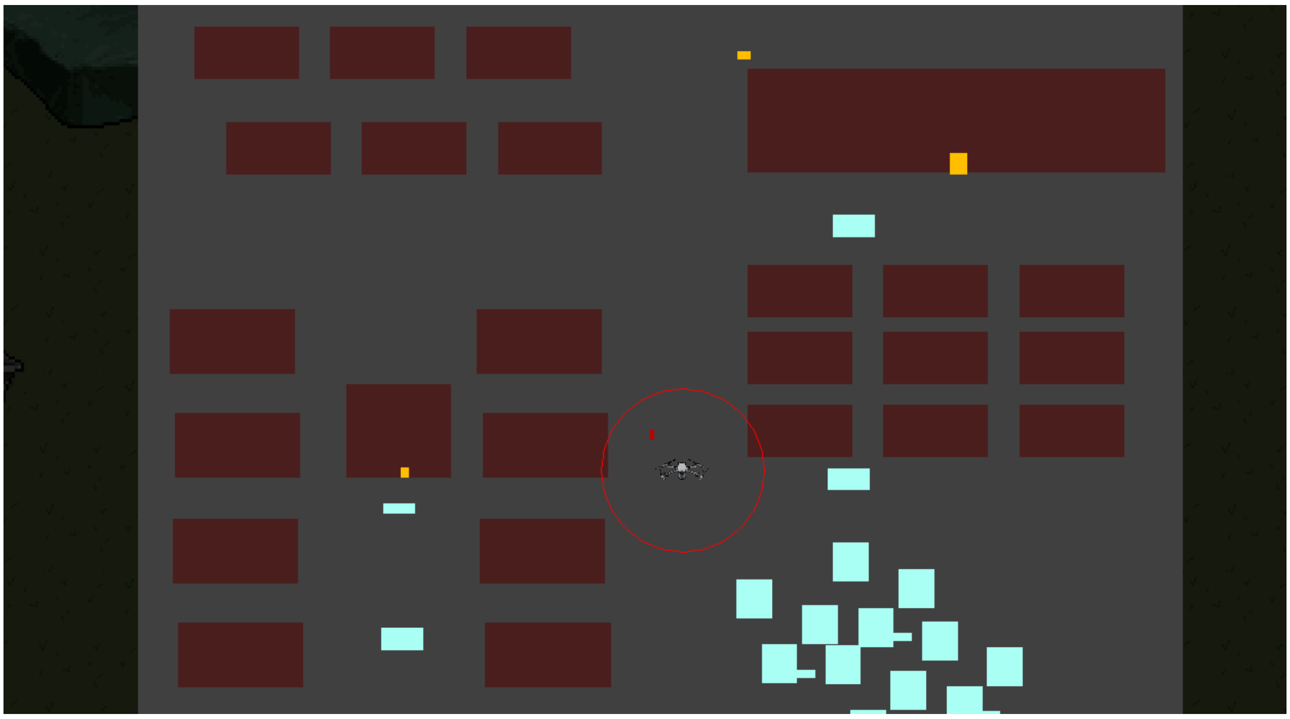


Figure 20. Scouted Town game area in a minimap view

In contrast to the single-player mode, the operations center was designed to monitor dynamically created game area instances instead of static world map locations in multiplayer. Additionally, the scouting drone was meant to be visible to players who adventure in the scouted instance. The networking required transmitting data and implementing new dependencies between the client-side map controller and server-side components.

When a player desires to operate the scouting drone, the client sends a *Request* packet to the server. On the server side, the *InstanceHandler* fetches a list of available instances by looping through its instance registry and filtering out sub-instances (see the JS code below). When the client receives the response, it asks the player to select a target location from the list. The player input then updates the map view and triggers a bidirectional data streaming between the client and the server.

```
/**
 * Fetches a list of available instance
 * based on specified flags
 * @param {boolean} excludeCamp
 * @param {boolean} onlyRootHierarchy
 * @return {Array} Available instances
 */
getAvailableInstances(excludeCamp, onlyRootHierarchy) {
  let availableInstances = [];
  let instanceIds = this.getInstanceIds();
  if (excludeCamp) {
    instanceIds = instanceIds.filter((instanceId) => {
      return parseInt(instanceId) !== this.campId;
    });
  }
  if (onlyRootHierarchy) {
    instanceIds = instanceIds.filter((instanceId) => {
      let instance = this.getInstance(instanceId);
      return instance.parentInstanceId === undefined;
    });
  }
  availableInstances = instanceIds.map((instanceId) => {
    let instance = this.getInstance(instanceId);
    const playerCount = instance.getPlayerCount();
    const patrolCount = instance.getPatrolCount();
    return new AvailableInstance(
      instanceId,
      instance.roomIndex,
      playerCount,
      patrolCount
    );
  });
  return availableInstances;
}
```

Next, the client sends a *Request* packet with target instance details to the server to initialize the streaming. On the server side, the *InstanceHandler* registers a new scouting stream and, from now on, restricts other players from accessing the operations center. Then, the server responds with a *Start Stream* packet and broadcasts a *Sync* packet to other clients within the scouted instance, who are instructed to spawn a flying drone into their game.

The client who operates the scouting sends outgoing *Stream* packets with replicated scouting drone data to the server. The newest drone data is then broadcast to targeted clients. In return, the server responds with an instance snapshot *Stream* packet sent continuously at a predefined interval. If the scouted instance has local players, the operating client can patch its map view using an extensive picture of the instance. Besides data about local players, the incoming snapshot *Stream* packets contain replication data on local patrols that a named owner-client has simulated using GameMaker's AI logic. The following GML code block is snipped from a *MapDataHandler* and its *SyncDynamicMapData* method, showing logic on how the handler patches existing player icons and deletes outdated ones.

```
// ...
if (!is_undefined(existentPlayerIndex))
{
  var player = _regionSnapshot.local_players[@ existentPlayerIndex];
  dynamicMapIcon.simulated_instance_object.StartInterpolateMovement(
    player.position, INSTANCE_SNAPSHOT_SYNC_INTERVAL
  );
  array_delete(_regionSnapshot.local_players, existentPlayerIndex, 1);
  playerCount = array_length(_regionSnapshot.local_players);

  // PRIORITIZE ICON
  dynamic_map_data.AddPrioritizedIcon(dynamicMapIcon);
} else {
  if (dynamicMapIcon.object_name == object_get_name(objPlayer))
  {
    // DELETE OUTDATED PLAYER MAP ICONS
    var player = dynamic_map_data.icons[| i];
    DeleteStruct(player);
    ds_list_delete(dynamic_map_data.icons, i--);
    dynamicIconCount = ds_list_size(dynamic_map_data.icons);
  }
}
// ...
```

If the scouted instance, instead, does not have local players, the incoming snapshot *Stream* packets are much smaller. They then only contain replicated data on local AI patrols that the server has simulated on the simulation layer using simpler AI models.

With the advantages of bidirectional data exchange, the operations center can monitor target instances in real time while the server can broadcast scouting drone data outward concurrently. The participant clients who can physically observe the flying drone inside the scouted instance utilize simple interpolations to replicate the drone's movement. The method mimics the drone's movement using received replication data to project the drone's original position from the miniature UI map into the in-game world (see Figure 21).

Figure 21. Flying drone next to a player

When the operating client desires to close the stream and stop controlling the scouting drone, it sends an *End Stream* packet to the server. The server then unregisters the stream and broadcasts a *Destroy* packet to clients as an instruction to destroy their locally flying drone.

In summary, GameMaker has no reason to run precise 2D physics on miniature-sized map icons because dynamic map markers are just replicated data and projections of game objects. Instead, the map view and monitoring logic utilize simpler models to simulate objects' behavior and movement, which reminds very similar modeling to the simulation layer on the server side. With the same processing logic, the game client can cut corners on 2D physics processing and simulate objects' movement on the map view. It can perform very lightweight calculations independently from the original and complex built-in object behavior and just focus on drawing icons onto the UI map.

The server-side simulation layer, in turn, allows the server to simulate enemy patrols that wander outside the players' presence, making the game world more persistent.

7.8 Debugging and testing

GameMaker Studio 2 offers a powerful built-in Debugger with a Debugger Workspace, including Debug mode, console, and visual graphs like an in-game Debug Layout (see Figure 22). These are essential tools for code debugging, performance and resource monitoring, profiling, error analyzing, and bug tracking on the instance, variable, and graphical levels. (Alexander 2021.)

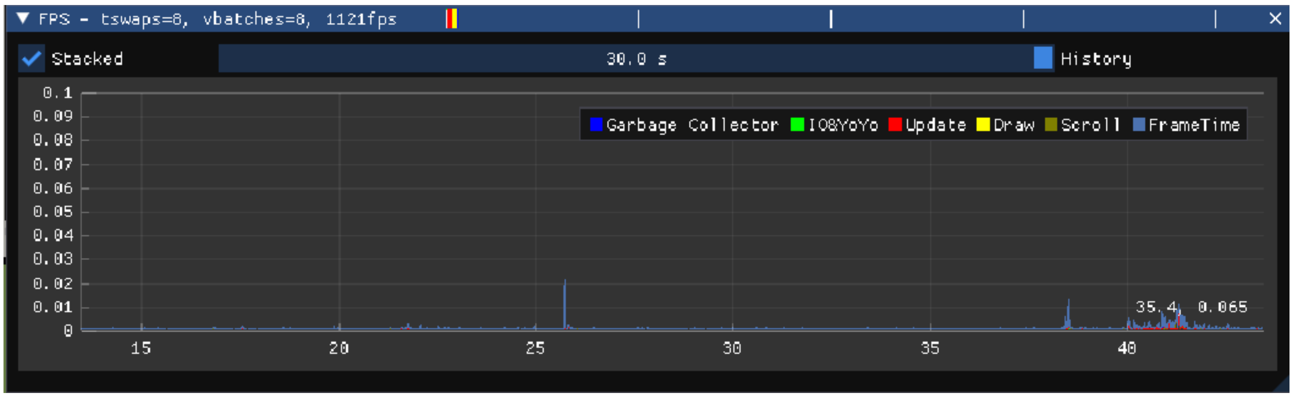
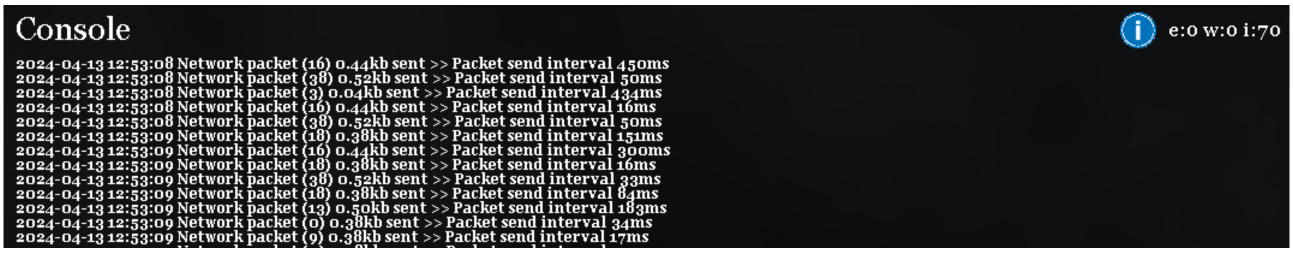


Figure 22. Debug Overlay

These built-in debugging tools have been widely used during single-player development and have, so far, fully covered the project’s needs. Unfortunately, they are not versatile enough for multiplayer development and lack crucial features for network monitoring. The Debugger does not have samplers and components that track data transmissions or sample incoming and outgoing network traffic. Furthermore, because the Debug Mode utilizes breakpoints to pause the executed code on client-side debugging, it may cause unavoidable desync in asynchronous networking.

Even though third-party software and packet analyzers, such as Wireshark, could have potentially filled the gap, they will be added to the project toolkit later. This is because the author is a bit inexperienced with their usage. Therefore, several custom data samplers and debug monitor components, including a custom in-game console, were implemented into the game prototype to expand the built-in Debugger features.

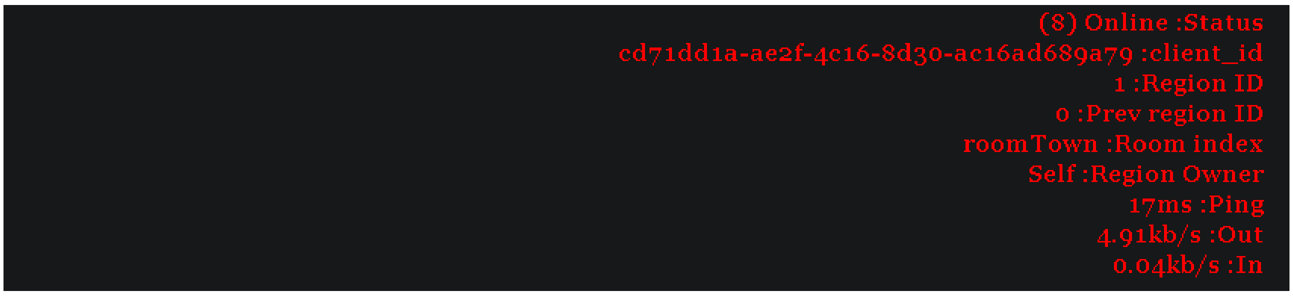
The custom console was designed to ease debugging, offering an in-game UI element that prints out the most recent console log history (as illustrated in Figure 23). Custom logs are formatted text messages that can be constructed from, for instance, a timestamp, network packet type, sent data size, and transmission interval values. The console provided a simple way to track network packet transmissions, data traffic, event history, warnings, and errors. Unfortunately, it turned out that when network traffic reached high flow peaks, it was sometimes hard to read the log prints without pausing the game loop.



```
Console                                                                 e:0 w:0 i:70
2024-04-13 12:53:08 Network packet (16) 0.44kb sent >> Packet send interval 450ms
2024-04-13 12:53:08 Network packet (38) 0.52kb sent >> Packet send interval 50ms
2024-04-13 12:53:08 Network packet (3) 0.04kb sent >> Packet send interval 434ms
2024-04-13 12:53:08 Network packet (16) 0.44kb sent >> Packet send interval 16ms
2024-04-13 12:53:08 Network packet (38) 0.52kb sent >> Packet send interval 50ms
2024-04-13 12:53:09 Network packet (18) 0.38kb sent >> Packet send interval 151ms
2024-04-13 12:53:09 Network packet (16) 0.44kb sent >> Packet send interval 300ms
2024-04-13 12:53:09 Network packet (18) 0.38kb sent >> Packet send interval 16ms
2024-04-13 12:53:09 Network packet (38) 0.52kb sent >> Packet send interval 33ms
2024-04-13 12:53:09 Network packet (18) 0.38kb sent >> Packet send interval 84ms
2024-04-13 12:53:09 Network packet (13) 0.50kb sent >> Packet send interval 183ms
2024-04-13 12:53:09 Network packet (0) 0.38kb sent >> Packet send interval 34ms
2024-04-13 12:53:09 Network packet (9) 0.38kb sent >> Packet send interval 17ms
```

Figure 23. Custom in-game console

Another new custom component features drawing real-time network statistics onto the screen (see Figure 24). This new in-game overlay provided valuable information about desired elements like network status, client identifier, instance metadata, measured ping, and incoming/outgoing data rates. The data was then used in debugging to track network traffic and detect mismatching states between the client and the server.



```
                                         (8) Online :Status
            cd71dd1a-ae2f-4c16-8d30-ac16ad689a79 :client_id
                                             1 :Region ID
                                        0 :Prev region ID
                                    roomTown :Room index
                                        Self :Region Owner
                                               17ms :Ping
                                              4.91kb/s :Out
                                               0.04kb/s :In
```

Figure 24. In-game network statistics overlay

7.8.1 Playtest

Playtesting was part of the final development stage of the game prototype iteration. It played an essential role in evaluating the project's success rate. The playtest was designed to monitor server-side load and stability and test the reliability of the custom protocol layer in a realistic environment. It also helped assess whether the game ran as expected and fulfilled the functional requirements. Additionally, it provided a brief analysis of the gameplay experience from the users' perspective.

Before the playtest was executed, built-in Debugger functionalities were expanded with custom data samplers, stopwatches, and debug monitor components to reach the desired test coverage. These samplers were built to be lightweight and easy to integrate into the code base, minimizing the impact on the game's performance. Collected samples and data could then be visualized on custom graphs. This stage took some development effort because GameMaker and its built-in Debugger did not deliver customizable samplers and debugging tools out of the box.

Finally, the playtest was executed with four actively playing client PCs and one dedicated server host machine, each running on Windows OS. Test samples were collected during a 30-minute game session. Playtesters were instructed to interact with primary game mechanics while still leaving room to explore the content freely. The chosen test scenario was designed to generate enough network traffic and noise for sufficient monitoring and stress testing to provide reliable and valuable data for analysis. The client count was purposely low during the playtest. That is because there was no need to simulate extreme workload conditions or intentionally overflow the game's stress resilience by exceeding the initial count of supported players.

Test analysis

One of the game clients was a named sampler that collected the data during the playtest. For clarity, data samples were split into three collections (10-minute periods) and embedded into a single graph with color coding to save a bit of text page space.

The first and most interesting client-side test sector was the data rate sampling. As every networked in-game object and every player input produced some amount of network traffic, the data rate reveals how successfully or poorly the client-side networking code was implemented. As seen on the curve (see Figure 25), the data rate (out) stays under 4 kb/s most of the time and varies around 1.8 kb/s (visual estimate) on average.

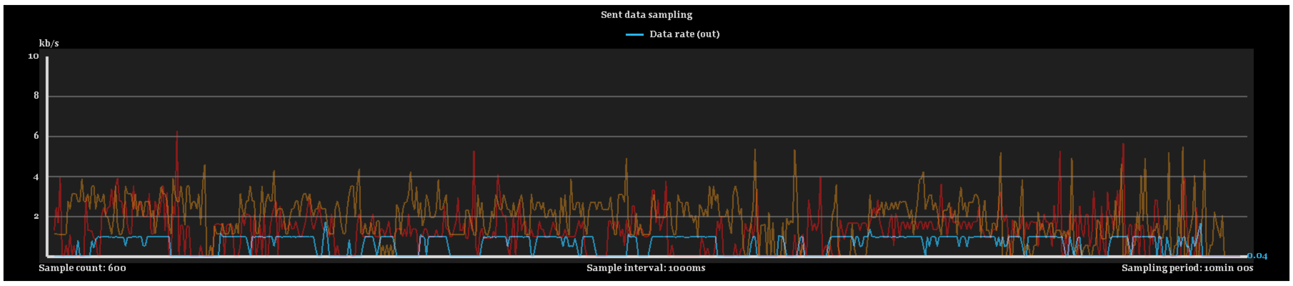


Figure 25. Graph data rate (out)

The following data rate (in) graph curve (see Figure 26) has noticeable sections where the rate is relatively high and suddenly reduces. The benefits and side effects of instancing can explain this phenomenon. High data rates usually happen when players gather into a single instance, and conversely, the rate drops when players scatter around the game world. However, as seen in the previous graph (see Figure 25), the data rate (out) peaks less because the rate (out) scales along local player interactions and not by player count.

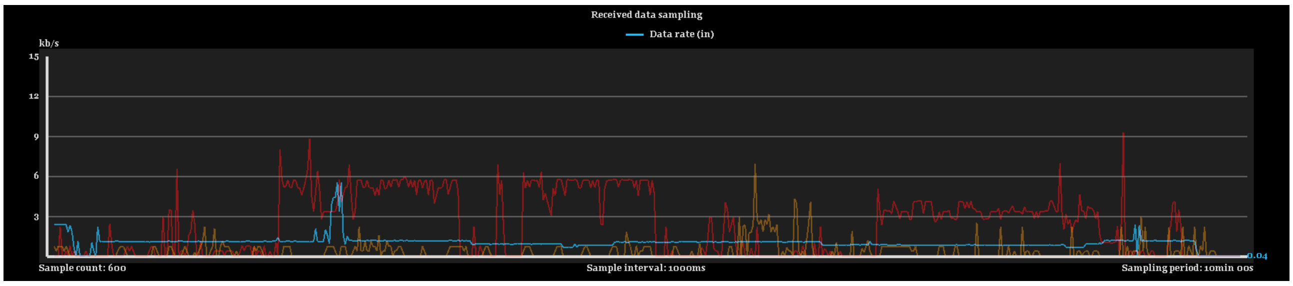


Figure 26. Graph data rate (in)

As a reference, the very first version of client-side networking produced over 10 kilobits (out) and over 16 kilobits (in) of data per second on average in identical scenarios. The code has gone through many optimization and testing cycles ever since, which have successfully shown excellent results.

Through precise optimization, both data rates stayed reasonably low. Networking leaned on sending data only when needed, including updating and syncing states in average 100-500 millisecond intervals or on user input. Additionally, most replicated data was transmitted in compressed form and written into buffers with bitwise mapping. Low networked entity count (see Figure 27) also affected the data rates.

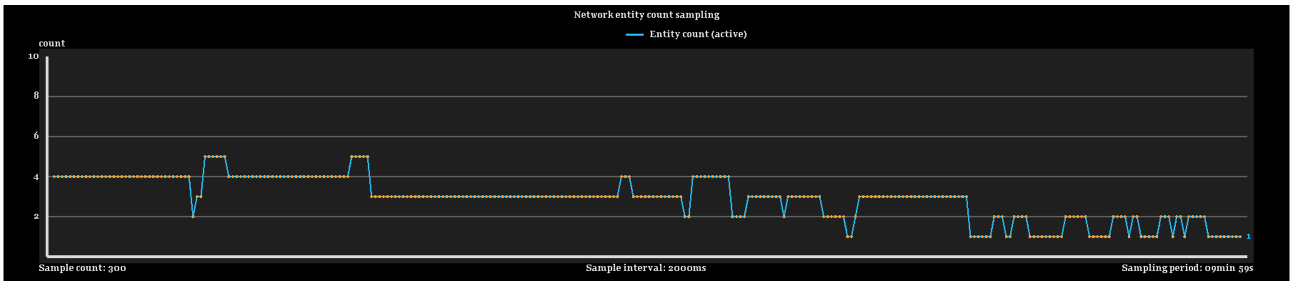


Figure 27. Graph entity count

The server-side testing, in turn, utilized a node module called Clinic.js, an open-source performance diagnostic tool. The test results show that the memory usage built up to 9 MB (see Figure 28) on average, while the RSS (resident set size) stayed around 40 MB, with no signs of noticeable memory leakage.

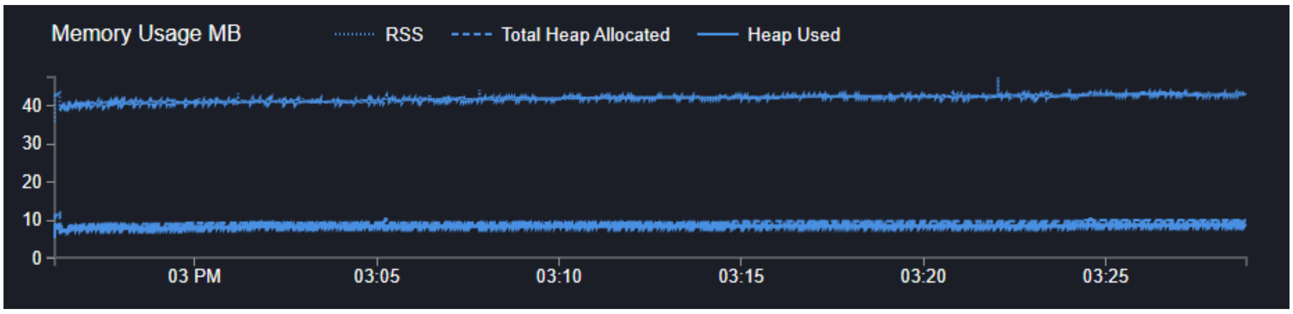


Figure 28. Graph memory usage

The server-side game loop was built to lean on Node.js runtime and its event loop routines, with no manual adjustments. As seen on the graph (see Figure 29), the event loop spikes a bit on server boot-up, but seconds later, it stabilizes and stays at 16 ms on average. This server-side loop speed fits nicely with game clients that run at 60 frames per second (capped). The test results also confirm that the server-side processing was well-optimized and had no noticeable lag spikes during the playtest.

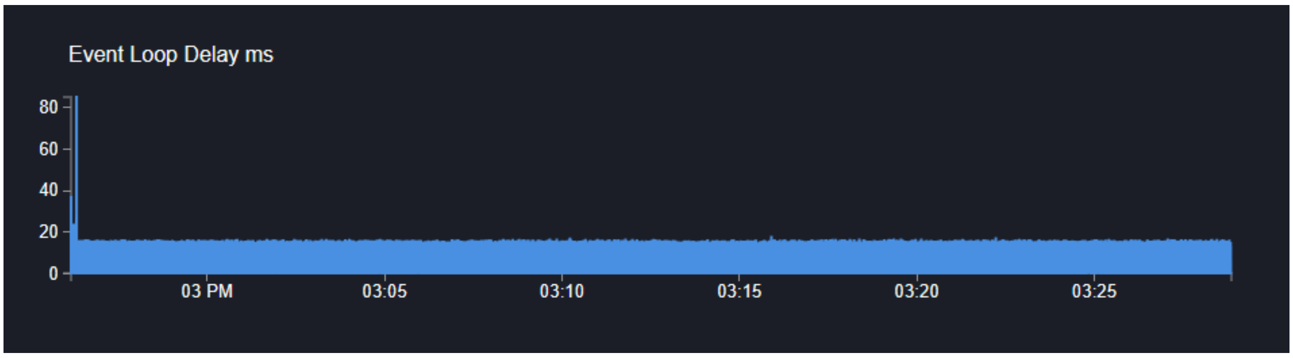


Figure 29. Graph event loop delay

8 Summary

8.1 Result analysis

The objective of this thesis was to build an integration between the GameMaker client and the Node.js server. The results met all the predefined requirements, and the playtesting showed a green light. The implementation included new multiplayer core features and real-time network communication built on an existing game prototype. The research provided brief theory and discussions of the benefits of different transport layer protocols and the basics of reliability and synchronization in network communication. The project itself followed common game design and agile development. The technological decision to use JavaScript with the Node.js runtime and the direct compatibility between the GameMaker engine and the node:dgram UDP socket noticeably eased the work.

The research questions defined the scope of this thesis topic, and the research successfully answered all of these questions. The literature review provided essential elements and fundamentals of network communication. The study compared different communication protocols and network topologies for their benefits and features, which influenced the decision-making on the game prototype's architecture and design. The literature review about game world zoning and instance systems in multiplayer games provided practical theories on easily manageable game area instancing. Topics like game designing and world-building, in turn, provided valuable instructions and examples for the simulation layer and its building blocks.

8.2 Research integrity and ethics

Research integrity involves various methods that direct the research and the study in question to follow good research practices throughout the work. Its fundamentals include dependability, honesty, respect, and responsibility. The research integrity instructs the author to ensure dependability in the research planning, methods, execution, and analysis. It also leads to honesty, where one shares and reports their research openly and fairly, without a sign of whitewash or bias. One must also respect the others and their work. The research integrity forbids all kinds of plagiarism, insult, and disrespect. It also forbids publication of someone's properties, material, and personal/sensitive information without proper references and permission. One must also show maturity and responsibility for their research and its results. (Hyvä tieteellinen käytäntö (HTK) 2023; Hyvä tieteellinen käytäntö ja sen loukkausepäilyjen käsitteleminen Suomessa 2023.)

These presented instructions and terms have been considered throughout this thesis and research. I have tried my best to follow the research integrity. Additionally, the game prototype itself does not contain any sexual, political, or racist content that could be considered violating or unethical.

8.3 Limitations

This multiplayer game project was developed by a single student with a limited network coding background and narrow experience with industrial standards. The project also lacked collaboration and code peer-reviewing, making the working entirely self-sufficient and self-directed.

Since this thesis covers just a fraction of the presented game project and related multiplayer development, there is still much to learn and explore. Multiplayer development as a concept is overwhelmingly complex, extensive, and impossible to fit into a single thesis paper. Therefore, something must have been left out of discussions. The study focused on specific tools, leaving room for discoveries and research from different perspectives. This thesis excluded comparisons between available game platforms and frameworks, which might have revealed alternative solutions. Additionally, client-side rendering and lag compensation were only mentioned in passing, leaving readers with open questions.

Because of these factors, these results are a bit optimistic and cannot be generalized to a broader context. Every step forward in development opens new questions, and every article or book, whether left out or newly published, can reveal new remarkable discoveries. Nonetheless, this research answered the predefined questions within the set scope.

8.4 Conclusions and further development

My personal targets included strengthening my game development skills and deepening my knowledge of used technologies and languages. The project naturally featured developing my very own indie game and evolving my project management skills, and I was used to a self-sufficient working style. However, the expedition to study and research something so complex with GameMaker Studio 2 was a whole new story, besides the difficulties with the project's schedule. Even though my prior knowledge of tools and languages carried me throughout the work, building network components to both the server and client side from scratch was occasionally time-consuming. It was a whole new exploration for me, stepping out of my comfort zone and proceeding with the game prototype into the unknown – into the dimension of multiplayer game development.

Because of my brief experience in multiplayer networking, it was pretty challenging to get the elements to fall into place on the first try. From the beginning, it took great effort to plan accurate project scheduling and write requirements specifications for the developed multiplayer core features. Especially as a newcomer, I had only a slight understanding of how much work the project would require and only a few explicit references on how the development would progress. It led to delaying the project from the planned schedule and tweaking the Software Requirements Specification documentation on the run until the full picture finally cleared in my mind.

The game prototyping required continuous testing with trials and errors, including tweaking and polishing. I found myself reading theory and learning new subjects from books and the Internet – all over again. As mentioned in the Research design, this research was just a scratch of the networking in multiplayer games. The developed prototype still lies far from the final state and lacks various critical mechanisms and components to meet the qualification for eligibility for publication. Further development should include performing more playtesting in different environments and scenarios to map potential reliability and stability issues. The remaining work also requires solving issues like minor desync, fluctuation in latency (known as jitter), and rubber banding. The game briefly utilizes client-side predictions and server reconciliation. With such improvements, the multiplayer gameplay experience would be way smoother and more enjoyable.

The hybrid server model is still under investigation and reworking. The model features several security mechanisms that aim to restrict players from cheating, but only partially. Because of the designed authority distribution, clients can still send false information that the server can only partially validate as correct or fake. Therefore, it might be the most noticeable security issue in the selected network topology that should be considered in future development.

The research topic was intentionally scoped to circulate the developed game prototype. The thesis mainly focused on building an integration between the GameMaker client and Node.js server but lacked comprehensive analytics and comparisons with alternative technologies. The newest GX.games platform module, in turn, has shown great potential to reduce development time in multiplayer game projects. However, ongoing studies have yet to discover its true compatibility, reliability, and performance in more complex and medium-scale multiplayer games.

In conclusion, this research confirmed that GameMaker Studio 2 stands out as a user-friendly and full-featured game development platform with the potential for multiplayer development. As a result, the game prototype progressed a step towards publication. Additionally, the research documentation turned into a compact information packet that can provide valuable discoveries and assist game developers in diving into the fascinating field of multiplayer networking. The developed game prototype may also provide usable references and blueprints for new multiplayer game projects.

The more people are willing to openly share their knowledge and findings on both successful and failed game projects, the greater the potential for development platforms like GameMaker to grow.

References

About Node.js. N.d. NodeJS website. Retrieved October 8, 2023, from https://nodejs.org/en/about.

About npm. N.d. Documentation on npm website. Retrieved October 10, 2023, from https://docs.npmjs.com/about-npm.

About packages and modules. N.d. Documentation on npm website. Retrieved October 10, 2023, from https://docs.npmjs.com/about-packages-and-modules.

Alexander, M. 2019. Beginners Guide to Networking. A blog on GameMaker website. Updated on October 10, 2019. Retrieved October 18, 2023, from https://gamemaker.io/en/blog/beginners-guide-to-networking.

Alexander, M. 2021. The Debugger. A blog on GameMaker website. Updated on January 1, 2021. Retrieved April 13, 2024, from https://gamemaker.io/en/tutorials/debugger.

An introduction to multiplayer network and server models. N.d. Unity website. Series of how-to articles. Retrieved October 15, 2023, from https://unity.com/how-to/intro-to-network-server-models#what-peer-peer-model.

Balasubramanian, K. 2022. Beginners Guide to Networking. A blog on Gameopedia website. Updated on August 23, 2022. Retrieved February 4, 2024, from https://www.gameopedia.com/online-multiplayer-games/.

Bramble, R. 2023. What platforms can I export my game to with GameMaker?. A blog on GameMaker website. Updated on June 13, 2023. Retrieved October 16, 2023, from https://gamemaker.io/en/blog/export-with-gamemaker.

Client-Server Network: Definition, Advantages, and Disadvantages. 2023. Zenarmor website. Network Basics. Updated October 4, 2023. Retrieved October 14, 2023, from https://www.zenarmor.com/docs/network-basics/what-is-client-server-network.

Eddy, W. 2022. Transmission Control Protocol (TCP). Internet Engineering Task Force (IETF). Published on August, 2022. Retrieved September 24, 2023, from https://datatracker.ietf.org/doc/html/rfc9293.

Fairhurst, G., Trammell, B. & Kuehlewind, M. 2017. Services Provided by IETF Transport Protocols and Congestion Control Mechanisms. Internet Engineering Task Force (IETF). Published March, 2017. Retrieved September 24, 2023, from https://datatracker.ietf.org/doc/html/rfc8095.

Ford, J. 2009. Getting Started with Game Maker. Australia, Brazil, Japan, Korea, Mexico, Singapore, Spain, United Kingdom, Unites States: Cengage Learning PTR.

GameMaker Manual. 2023. YoYo Games website. Retrieved October 16, 2023, from https://manual.yoyogames.com/#t=Content.htm.

GameMaker. 2023. Article from Wikipedia. Updated on September 16, 2023. Retrieved October 16, 2023, from https://en.wikipedia.org/wiki/GameMaker.

Glazer, J. & Madhav, S. 2015. Multiplayer Game Programming: Architecting Networked Games. United Kingdom: Pearson Education.

Guide To Using Buffers. 2023. An additional part of a manual on YoYo Games website. Retrieved October 20, 2023, from https://manual.yoyogames.com/Additional_Information/Guide_To_Using_Buffers.htm.

Hyvä tieteellinen käytäntö (HTK). 2023. Article in Tutkimuseettinen neuvottelukunta website. Published 2023. Retrieved February 3, 2024, from https://tenk.fi/fi/tiedevilppi/hyva-tieteellinen-kaytanto-htk.

Hyvä tieteellinen käytäntö ja sen loukkausepäilyjen käsitteleminen Suomessa. 2023. Instruction documentation. Tutkimuseettinen neuvottelukunta website. Updated on October 9, 2023. Retrieved February 3, 2024, from https://tenk.fi/sites/default/files/2023-03/HTK-ohje_2023.pdf.

Kananen, J. 2015. Opinnäytetyön kirjoittajan opas : näin kirjoitan opinnäytetyön tai pro gradun alusta loppuun. Jyväskylän Ammattikorkeakoulu. Retrieved February 1, 2024, from https://janet.finna.fi, Booky.

Mead, A. 2018. Learning Node.js development: Learn the fundamentals of Node.js, and deploy and test Node.js applications on the web. Birmingham, England; Mumbai, [India]: Packt. Retrieved October 8, 2023, from https://janet.finna.fi, Ebook Central Academic Complete International Edition.

Minor, J. 2022. GameMaker Studio 2 Review. Review blog on PC Magazine website. Updated on July 21, 2022. Retrieved October 16, 2023, from https://www.pcmag.com/reviews/gamemaker.

Node.js v20.8.0 documentation. N.d. Official API reference documentation for Node.js. Retrieved October 10, 2023, from https://nodejs.org/dist/latest-v20.x/docs/api/.

Pernaa, J. 2013. Kehittämistutkimus tutkimusmenetelmänä. Article in HELDA website. University of Helsinki Open Repository. Published 2013. Retrieved February 1, 2024, from http://hdl.handle.net/10138/317958.

Porting from client-hosted to DGS - Client-hosted vs DGS-hosted. 2023. Documentation on Unity website about multiplayer networking. Updated on June 13, 2023. Retrieved October 15, 2023, from https://docs-multiplayer.unity3d.com/tools/current/porting-to-dgs/client-vs-dgs/.

Relay servers. N.d. Documentation on Unity website. Retrieved October 15, 2023, from https://docs.unity.com/ugs/en-us/manual/relay/manual/relay-servers.

Sherman, W. & Craig, A. 2003. Understanding Virtual Reality: Interface, Application, and Design. Netherlands: Elsevier Science.

TCP and UDP in Transport Layer. 2021. Data Communication Tutorial. Java Networking. Updated November 5, 2021. Retrieved September 24, 2023, from https://www.geeksforgeeks.org/tcp-and-udp-in-transport-layer/.

Tremblay, K. 2023. Collaborative Worldbuilding for Video Games. eBook. Retrieved October 22, 2023, from https://www.amazon.com/Collaborative-Worldbuilding-Video-Kaitlin-Tremblay-ebook/dp/B0BS767HBV/ref=tmm_kin_swatch_0?_encoding=UTF8&qid=1693912674&sr=1-1, Amazon Kindle.

User Datagram Protocol (UDP). 2023. Data Communication Tutorial. Java Networking. Updated May 11, 2023. Retrieved September 24, 2023, from https://www.geeksforgeeks.org/user-datagram-protocol-udp/.

Wexler, Y. & Simpson, K. 2019. Get programming With Node.js. Shelter Island, New York: Manning Publications. Retrieved October 8, 2023, from https://janet.finna.fi, Skillsoft Books ITPro.

What is OSI Model? – Layers of OSI Model. 2023. Computer Network Tutorial. Updated September 15, 2023. Retrieved September 24, 2023, from https://www.geeksforgeeks.org/open-systems-interconnection-model-osi/.

What is TCP/IP?. N.d. Learning Center. Cloudflare. Retrieved September 24, 2023, from https://www.cloudflare.com/learning/ddos/glossary/tcp-ip/.

What is UDP?. N.d. Learning Center. Cloudflare. Retrieved September 24, 2023, from https://www.cloudflare.com/learning/ddos/glossary/user-datagram-protocol-udp/.

Young, A., Meck, B., Cantelon, M. & Oxley, T. 2017. Node.js in action. Second edition. New York: Manning Publications. Retrieved October 8, 2023, from https://janet.finna.fi, Skillsoft Books ITPro.

YoYo Games. 2023. Article from Wikipedia. Updated on April 5, 2023. Retrieved October 16, 2023, from https://en.wikipedia.org/wiki/YoYo_Games.

Appendices

Appendix 1. Software Requirements Specification

(On the next page)

Software Requirements Specification

PROJECT: WORLD AGAINST US

GameMaker Studio 2 and Node.js server

Eetu Aaltonen

13/2/2024
PROTOTYPE V0.0.2

Contents

Introduction

Purpose

This document is part of the World Against Us game project. The developed game version should meet these predefined requirements and features for complete quality assurance.

Intended Use

The prototype iteration v0.0.2 is one of the milestones on the current indie game development roadmap. The roadmap is designed to divide the development process into smaller manageable iterations. These iterations progress the project — step by step — toward a planned release date, estimated to land in 3-5 coming years.

Project Scope

The objective is to extend the current game version with multiplayer capabilities by building core features for the next prototype iteration. The multiplayer mode will bring new value to the game from the gameplay experience perspective. The resulting core features will not replace the game elements or the support for single-player mode. Instead, new network components will be embedded into the existing game logic.

Overall Description

The new multiplayer mode requires establishing network communication between a server and game clients. The networking will utilize the UDP transport layer protocol to integrate GameMaker Studio 2 with a dedicated Node.js server. The server will have authority over the global world state and responsibility for maintaining the synchronization between participating hosts.

Game clients, in turn, will run client-side game logic and communicate directly with the server. They also act like outsourced workers for the server, providing processing work and simulations that require code execution using GameMaker engine logic.

Assumptions and Dependencies

The implemented architecture will only support Windows platforms.

System Features and Requirements

Description

The first version of the multiplayer mode will include game area instancing and a simulation layer built on new network components. The current game is planned to adapt to networking, and the game logic is intended to expand with a new instance management system, global simulations, and extended AI behavior. The new instancing will extend the GameMaker logic through server-side instance management that casts world map locations into independent instances. Current world map locations and in-game rooms will be mapped onto a parent-child hierarchy as part of the instance management system.

New global simulations will feature global time, dynamic weather, and new patrolling AI bandits. New Operations Center features will utilize the simulation layer and replicated data to render player characters and enemy bandits onto a miniature-sized map view in real time in multiplayer mode.

Functional Requirements

Category	ID	Description	Scope
NETWORKING			
	FR_0	Clients should be able to connect to the server	Client
	FR_1	Clients should be able to disconnect from the server	Client
CLIENT			
	FR_2	Game client should have a DEBUG mode	Client
INSTANCES			
	FR_3	Players should be able to fast-travel to a new instance	Game
	FR_4	Players should be able to request a list of available instances	Game
	FR_5	Players should be able to fast-travel to an available instance	Game
	FR_6	Players should be able to return back to the Camp via local fast-travel points	Game
GAME AREA			
	FR_7	Players should be able to enter local buildings via their entrance door inside a world map location	Game
	FR_8	Players should be able to exit buildings via their entrance door	Game
BACKPACK			
	FR_9	Players should be able to open and browse their backpack	Game
	FR_10	Players should be able to move, rotate, and delete items inside their backpack	Game
CAMP STORAGE			
	FR_11	Players should be able to open and browse the Camp storage	Game
	FR_12	Players should be able to deposit items into the Camp storage	Game
	FR_13	Players should be able to withdraw items from the Camp storage	Game
	FR_14	Players should be able to move, rotate, and delete items inside the Camp storage	Game
CONTAINERS			
	FR_15	Players should be able to open and loot containers	Game
	FR_16	Players should be able to withdraw items from loot containers	Game
	FR_17	Players should be able to move, rotate, and delete items inside loot containers	Game
PATROLS			
	FR_18	Players may encounter patrolling bandits outside the Camp area	Game
OPERATIONS CENTER			
	FR_19	Players should be able to use the Operations Center at the Camp	Game
	FR_20	Players should be able to browse available instances via an UI list	Game
	FR_21	Players should be able to select an active instance location from the list	Game
	FR_22	Players should be able scroll and move the map view	Game
	FR_23	Players should be able to control a flying scouting drone	Game

Nonfunctional Requirements

Category	ID	Description	Scope
NETWORKING			
	NR_0	Network communication should be stateless and lightweight	Protocol
	NR_1	Server should have a client registry	Server
	NR_2	Server should be able to send and broadcast messages to clients	Server
	NR_3	Server should be able to receive network packets from clients	Server
	NR_4	Clients should be able to send network packets to the server	Client
	NR_5	Clients should be able to receive network packets from the server	Client
	NR_6	Server should register each connected client	Server
	NR_7	Clients should receive an UUID on connection process	Protocol
	NR_8	Server should delete disconnected clients from the registry	Server
	NR_9	Server should be able to disconnect clients in certain conditions	Server
	NR_10	Server should broadcast a message about a joined client	Server
	NR_11	Server should broadcast a message about a disconnected client	Server
	NR_12	Clients should not be able communicate and exchange messages directly with other clients	Client
	NR_13	Players should be redirected back to main menu after the game client disconnects from the server	Game
SAVE FILE			
	NR_14	Player data should be saved in client-side save files	Client
	NR_15	Global world state should be saved in server-side save files	Server
	NR_16	Server should autosave every 10 real-time minutes	Game
	NR_17	Server should broadcast autosave commands to clients	Server
	NR_18	Clients should execute local autosave on disconnection	Client
INSTANCES			
	NR_19	Server should have an instance registry	Server
	NR_20	Players should be redirected to the default Camp instance when they join the multiplayer session	Game
	NR_21	Players should be spawned to the entrance door of a building they exit or enter	Game
	NR_22	Server should erase empty parent instances from memory only in certain conditions when all players leave the area	Server
	NR_23	Server should erase empty child instances from memory only via recursive deletion with their parent instance	Server
WORLD STATE			
	NR_24	World state should have a global time	Game
	NR_25	World state should have a day-night cycle	Game
	NR_26	24 in-game hours should be equivalent to one real-time hour by default	Game
	NR_27	World state should have dynamic weather	Game
	NR_28	World state weather should reroll every two in-game hours	Game
	NR_29	World state should be synchronized with joining clients	Protocol
	NR_30	Global weather should be synchronized with game client on room start event	Protocol
CONTAINERS			
	NR_31	Server should have a container registry	Server
	NR_32	Containers should have unique IDs	Game
	NR_33	Generated loot and items should appear unidentified by default	Game

	NR_34	Loot rolls should be based on the predefined loot tables	Game
INVENTORY STREAM			
	NR_35	Server should be able to manage active inventory streams	Server
	NR_36	Inventory streams should have three stages to function: start, stream, end	Game
	NR_37	Inventory streams should have an item limit for each data transmission	Protocol
ITEMS			
	NR_38	Game client should have database for items' data	Client
	NR_39	Server should only store and handle items in their replication form	Server
	NR_40	Item identification should take a few seconds before revealing the full item	Game
	NR_41	Unidentified items should only have an 'Identify' interaction option	Game
PATROLS			
	NR_42	Enemy bandits should be able patrol in each patrol-routed world map location	Game
	NR_43	Patrol count should vary between 1 to 3 for each world map instance	Game
	NR_44	Patrols should have varying queue and travel times	Game
	NR_45	Bandits should be able to follow predefined patrolling routes inside game areas	Game
	NR_46	Bandits should only patrol outdoor areas	Game
	NR_47	Bandits should be able to chase players	Game
	NR_48	Bandits should be able to rob players	Game
	NR_49	Bandits should be able to find their way back to their patrolling routes	Game
	NR_50	Bandits should have vision radius to detect players	Game
	NR_51	Players should lose all their items from backpack on occurred robbery	Game
	NR_52	Players should be redirected back to Camp after being robbed	Game
OPERATIONS CENTER			
	NR_53	Operations Center should project game areas onto a miniature map view	Game
	NR_54	Operations Center should draw and sync players on the map view	Game
	NR_55	Operations Center should draw and sync patrols on the map view	Game
	NR_56	Scouting drone should have vision radius that reveals objects on the map view	Game
	NR_57	Scouting drone should be visible and synchronized with clients within the scouted instance	

Performance Requirements

Category	ID	Description	Scope
NETWORKING			
	PR_0	Network packets with integers and reasonably short string should encapsulate the data in optimized bitwise format	Protocol
	PR_1	Network packets with reasonably large and complex data structures should encapsulate the data in JSON format	Protocol
GAME CLIENT			
	PR_2	Game client should perform stable 60fps over 95% of the time when running on modern computers	Client

Reliability Requirements

Category	ID	Description	Scope
NETWORKING			
	RR_0	Game client should receive a response or an acknowledgment from the server on every registration process step	Protocol
	RR_1	Server should receive a response or an acknowledgment from a joining client on every registration process step	Protocol
	RR_2	Server should respond with matching sequence numbers for corresponding acknowledgments	Protocol
	RR_3	Game client should respond with matching sequence numbers for corresponding acknowledgments	Protocol
	RR_4	Game client should resend critical messages after 3 seconds on acknowledgment timeout	Protocol
	RR_5	Server should resend critical messages after 3 seconds on acknowledgment timeout	Protocol
	RR_6	Server should validate the content of each incoming network packet	Server
	RR_7	Server should notify a sender client about an invalid network packet	Server
	RR_8	Game client should sample PING every 1 second	Client
	RR_9	Server should sample and track PING for each client separately	Server
	RR_10	Server should disconnect clients that have not sent PING messages by 3 seconds	Server
	RR_11	Server should disconnect clients that have a PING value >1000ms	Server
	RR_12	Game client should disconnect from the server if the server has not responded to a PING message after 5 seconds	Client
ERROR HANDLING			
	RR_13	Server should shut down shortly after an Internal Server Error	Server
	RR_14	Server should send notifications to clients about occurred errors	Server

Usability Requirements

Category	ID	Description	Scope
NETWORKING			
	UR_0	Players should be able to join to a server via main menu interface	Player
	UR_1	Players should be able to join an ongoing multiplayer session	Player
	UR_2	Players should be able to disconnect from the server at any time	Player
UI			
	UR_3	Game client should show the client ID on UI	Client
	UR_4	Game client should show the PING on UI	Client
	UR_5	Game client should show the instance ID on UI	Client
	UR_6	Game client should show the global time on UI	Client
	UR_7	Exit button should be located on ESC menu	Client
	UR_8	Game client should hide the single-player related options from the ESC menu in multiplayer mode	Client
GAME CLIENT			
	UR_9	Game client should show a loading window on connect queue	Client
	UR_10	Game client should show a loading window on fast-travel queue	Client

	UR_11	Game client should visualize the day-night cycle	Client
	UR_12	Game client should visualize weather conditions	Client
	UR_13	Game client should show a loading window on a robbery	Client
	UR_14	Game client should show an icon indicator on autosaving	Client

Security Requirements

Category	ID	Description	Scope
NETWORKING			
	SR_0	Server should identify each client using UUIDs	Protocol
	SR_1	Server should not share IP addresses or port numbers of any clients to other clients	Protocol
	SR_2	Server should have authority and control over the client registry	Server
	SR_3	Server should have authority and control over the global world state	Server
	SR_4	Server should respond with an error to unknown clients who has invalid or lack identification after the registration process	Server