Master's Programme in Computer, Communication and Information Sciences

Deterministic and Synchronous Computation Between Client and Server in Mobile Games

The synchronized client-server architecture of Metaplay SDK, a case study

---

Henri Viitanen

Aalto University

MASTER'S THESIS

Aalto University
MASTER'S THESIS 2025

Deterministic and Synchronous Computation Between Client and Server in Mobile Games

The synchronized client-server architecture of Metaplay SDK, a case study

Henri Viitanen

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science.
Otaniemi, 31 July 2025

Supervisor: Professor Perttu Hämäläinen
Advisor: Master of Science Antti Hätälä

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Author
Henri Viitanen 1034897

Title
Deterministic and Synchronous Computation Between Client and Server in Mobile Games

School School of Science

Master's programme Computer, Communication and Information Sciences

Major Game Design and Development **Code** SCI3046

Supervisor Professor Perttu Hämäläinen

Advisor Master of Science Antti Hätälä

Level Master's Thesis **Date** 31 July 2025 **Pages** 65 **Language** English

Abstract

Mobile gaming has grown into a globally dominant entertainment sector, driven by advances in smartphone technology and a flourishing free-to-play (F2P) business model. Despite its broad reach and economic success, this arena poses unique challenges for developers, including the need to curb cheating, manage intricate game states, and maintain fair but profitable in-game economies. Responding to these demands, a server-authoritative yet distributed paradigm has emerged, synchronising execution of game logic across client and server without requiring constant connectivity.

This thesis examines how deterministic game progression, achieved through server-validated states and atomic operations, not only ensures fairness but also minimises network overhead by allowing localised client-side execution. Moreover, the paradigm supports dynamic live operations and over-the-air updates, enabling developers to adapt gameplay and monetisation strategies in real time. By structuring logic in a cohesive, monolithic framework, it simplifies complex game features, ranging from single-player modes to extensive multiplayer and social elements, while preserving maintainability throughout a game's long lifecycle.

This thesis documents and analyses a paradigm that has emerged within the Finnish mobile game industry but lacks formal academic treatment. The work presents the synchronous execution approach as a practical framework for mobile game development, addressing common challenges in free-to-play and Games-as-a-Service models through established industry practices.

Keywords mobile games, synchronous computation, client-server architecture, synchronised game logic, free-to-play, deterministic game logic, server-authoritative, live operations, game development

Tekijä
Henri Viitanen

Työn nimi
Deterministinen ja synkronoitu pelilogiikka pelaajalaitteen ja pelipalvelimen välillä

Korkeakoulu Perustieteiden korkeakoulu

Maisteriohjelma Computer, Communication and Information Sciences

Pääaine Pelisuunnittelu ja -kehitys **Koodi** SCI3046

Valvoja professori Perttu Hämäläinen

Ohjaaja diplomi-insinööri Antti Hätälä

Työn laji Diplomityö **Päiväys** 31.07.2025 **Sivuja** 65 **Kieli** englanti

Tiivistelmä

Mobiilipelaaminen on kasvanut maailmanlaajuisesti hallitsevaksi viihdesektoriksi, jota vauhdittavat älypuhelintekniikan kehitys ja menestyksekäs free-to-play (F2P) -liiketoimintamalli. Laajasta levinneisyydestään ja taloudellisesta menestyksestään huolimatta alaan liittyy kehittäjille ainutlaatuisia haasteita, kuten huijausten estäminen, monimutkaisten pelitilojen hallinta sekä oikeudenmukaisten ja samalla tuottavien pelitalouksien ylläpitäminen. Näihin tarpeisiin on vastattu hajautetulla, palvelinlähteisellä mallilla, jossa pelilogiikka ajetaan pelilaitteella ja palvelimella synkronoidusti ilman vaatimusta jatkuvasta verkkoyhteydestä.

Tämä opinnäytetyö tarkastelee, kuinka synkronoitu, deterministinen pelilogiikka varmistaa pelin reiluuden ja samalla vähentää verkon kuormitusta, kun osa pelilogiikasta voidaan suorittaa välittömästi pelilaitteella. Lisäksi kyseinen malli mahdollistaa dynaamiset live-ops -toiminnot ja verkon yli tapahtuvat sisältöpäivitykset, joiden avulla kehittäjät voivat hienosäätää pelilogiikkaa ja ansaintamalleja reaaliajassa. Toteuttamalla pelilogiikka työn esittelemällä mallilla, voidaan eri pelitilat, yksinpeleistä laajoihin moninpeleihin, toteuttaa tehokkaasti, säilyttäen samalla pelin ylläpidettävyyden koko sen pitkän elinkaaren ajan.

Tämä opinnäytetyö dokumentoi ja analysoi mallia, joka on kehittynyt suomalaisessa mobiilipeliteollisuudessa, mutta jota ei ole vielä aikaisemmin käsitelty akateemisesti. Työ esittelee synkronisen suoritusmallin käytännöllisenä viitekehyksenä mobiilipelikehitykselle, keskittyen Metaplay Oy:n kehittämään toteutukseen.

Avainsanat mobiilipelit, synkronoitu pelilogiikka, asiakas-palvelin-arkkitehtuuri, synkronoitu laskenta, free-to-play, pelilogiikka, hajautettu laskenta, pelipalvelin

Table of Contents

1. Glossary of Technical Terms

A* (A-star) A popular pathfinding algorithm used in game development for finding the shortest path between two points.

A/B testing A method of comparing two versions of a game element to determine which performs better.

Asynchronous multiplayer A multiplayer mode where players don't need to be online simultaneously to interact.

Backend servers Servers that handle game logic, data storage, and other behind-the-scenes operations.

Battle royale A genre of multiplayer games where a large number of players compete to be the last one standing, often in a shrinking play area.

Boilerplate Code that is reused across multiple parts of a program with little or no modification.

Cheat-proofing Measures taken to prevent players from manipulating game data or exploiting vulnerabilities to gain unfair advantages.

Checksums Values computed from data blocks to verify data integrity.

Cloud-based backend infrastructure Servers and services hosted in the cloud to support game operations.

Command Pattern A programming pattern that encapsulates an object with the necessary information and logic required to perform a specific action or operation.

Configuration Over-The-Air (OTA) The ability to update game settings remotely without requiring rebuilding or redeploying the game.

Continuous content pipeline A system for continuously creating, testing, and deploying new game content.

Counter-Strike A popular first-person shooter game series known for its team-based gameplay and competitive scene.

Dedicated server A server with a single purpose of hosting a game instance.

Delta encoding A strategy for state synchronisation by sending incremental changes.

Deterministic collections Collections whose iteration orders need to be explicitly or implicitly defined to enforce consistency.

Entity Actor In Metaplay, an Actor responsible for implementing the behavior of a Game Entity on the backend, categorised as Ephemeral or Persisted.

Eventful gaming The incorporation of time-limited, in-game events that offer unique and engaging experiences for players.

First-person shooter (FPS) A genre of video games where players experience the game through the eyes of the protagonist, typically involving shooting mechanics.

Fixed-time tick-based simulations Game simulations that update at regular, predetermined intervals.

Free-to-play A business model where the basic game is free, but players can pay for additional features or content.

Freemium See: Free-to-play.

Game-as-a-service (GaaS) model A business model where games are provided as an ongoing service rather than a one-time purchase.

Game engine A software framework for creating video games.

Godot High-Level Multiplayer API A multiplayer framework offered by the Godot game engine for synchronising game state across servers and clients by replicating game objects.

Hot seat A local multiplayer mode where players take turns using the same device or controller.

IL2CPP Unity's solution for converting C# code to C++, improving performance and enabling broader platform support.

In-app purchases Transactions made within a game to buy virtual goods or features.

LiveOps Short for "live operations", refers to the ongoing management and improvement of a game after its initial release.

Lockstep mechanisms A synchronisation method where all players' actions are processed simultaneously before updating the game state.

Microservice A software development technique where an application is divided into smaller, independent services that communicate with each other.

Micro-transactions Small financial transactions within a game, typically for virtual items or currency.

Model In Metaplay, a Model is a data structure that represents the state of a Game Entity, e.g. a player profile.

Monolithic Server Architecture A server architecture where all game logic and state management is handled by a single server instance.

Online dashboard A web-based interface for managing various aspects of game operations and analytics.

Overwatch A team-based multiplayer first-person shooter game developed by Blizzard Entertainment.

Peer-to-peer network A network where computers communicate directly with each other without a central server.

Play-by-mail games A type of game where players take turns by sending their moves through postal mail.

PubSub system Short for Publish-Subscribe, a messaging pattern where senders of messages (publishers) do not program the messages to be sent directly to specific receivers (subscribers).

Real-time multiplayer A type of multiplayer game where all players interact with the game world simultaneously in real-time.

Reconciliation The process where the client adjusts its simulation to match the server's authoritative game state after client-side prediction.

Reification The process of making an abstract concept concrete. In programming, it often refers to treating a concept as a first-class object.

Rollback strategies Used in server reconciliation to revert the client's game state to the last known authoritative state and re-simulate.

Roslyn compiler The .NET Compiler Platform, also known as Roslyn, providing open-source compilers and code analysis APIs for C# and Visual Basic .NET languages.

Server infrastructure The hardware and software systems that support online game functionality.

Server-authoritative logic Game logic that runs on the server, which has the final say on game state and player actions.

Social connectivity Features in games that allow players to interact, compete, or cooperate with each other.

Source Engine A game engine by Valve Software, its multiplayer networking is characterised by the server running the entire game state simulation, with clients sending inputs and receiving updates.

Synchronous execution paradigm A game development approach where game logic is executed simultaneously on both client and server.

Thin-client A client that only handles rendering and user input, relying on the server for logic and state management.

Tick-based approach A method of game simulation where the game state is updated at fixed time intervals called "ticks".

Turn-based multiplayer A type of multiplayer game where players take turns to play.

UDP (User Datagram Protocol) A communications protocol that facilitates the exchange of messages between computers in a network with low latency but without guarantee of delivery, ordering, or duplicate protection.

Unreal Engine Iris Replication System A multiplayer framework within Epic Games' Unreal Engine that facilitates game state synchronisation through the replication of game objects between the server and clients.

2. Introduction

Mobile gaming has emerged as a dominant force within the entertainment industry, evolving rapidly alongside advancements in smartphone technology [1]. With billions of smartphone users worldwide, mobile gaming has developed into a highly competitive and dynamic market. Staying relevant in the mobile gaming space requires developers to maintain their games over long periods from years to decades, reacting to seasonal events and adapting to changing player preferences and market trends. To meet these demands, developers must create games that are secure, scalable, and maintainable while staying relevant for their playerbase.

These requirements have led to the development of a common set of tools and practices among mobile game developers, particularly in the free-to-play segment where games are offered for free with optional in-app purchases. The multifaceted monetisation schemes of free-to-play games require developers to implement complex game economies, which requires a robust backend infrastructure for maintaining game state, managing player data, and offering continuous new content.

An overview of the toolset required for maintaining a free-to-play mobile game will be presented in this thesis, but the focus will be on the game state management which lies at the core of it all. While tools for e.g. managing the game economy with its obtainable items and in-game currencies, tracking player behavior and engagement, or a control platform for player support agents for managing individual player states, are all aspects of the toolset, the game state management is the central piece that ties all these together.

This thesis will explore a certain paradigm, which has emerged, in multiple forms, in the Finnish mobile game industry over the past couple of decades. This paradigm is based on both the author's experience in the industry, as well as Metaplay Oy's proprietary game development framework, studied during the author's employment at the company. The theory behind the paradigm will be based on the author's experiences and observations, and the case study of Metaplay Oy's proprietary game development framework will be used to illustrate a practical implementation of the paradigm.

Central to this paradigm is the distributed yet synchronised execution of game logic across both client and server (Synchronous Execution). Un-

like traditional client-server architectures, this paradigm enables a fluid game experience without requiring a constant connection to the server. By synchronising key aspects of game logic while allowing localised client-side execution, it ensures smooth gameplay even in environments where connectivity may be intermittent. While recent developments in mobile networking allows for faster more reliable connections, mobile game developers still need to consider the possibility of network disruptions. As the paradigm has its roots in the era of less reliable mobile networks, it has a strong focus on maintaining a smooth gameplay experience even in the face of such disruptions.

The explored paradigm employs server-authoritative logic to maintain consistency and fairness, while ensuring deterministic progression of the game state and allowing client-side execution. This structure allows the server to validate player actions and maintain integrity without being required to execute simultaneously with the client, reducing the burden on network resources and enhancing scalability. The use of atomic operations further optimises this approach by facilitating secure, structured, and efficient authorship and execution of the gameplay. It's an approach that bridges a gap between the traditional client-server multiplayer game architectures, and the microservice architectures (a collection of small, autonomous services) that have become a standard in mobile app development.

While mobile and web apps adopted microservice architectures to allow scaling their services horizontally, video games have a different set of requirements that make microservices less suitable. The overall game state is a complex structure, with a higher degree of interconnection between its parts than in most web services. This complexity makes it difficult to split the game state into separate services, while a traditional dedicated game server approach struggles to scale horizontally on the scale required in mobile platforms. The explored paradigm threads the needle between these two approaches, offering a framework for developing and maintaining the game state and auxiliary services in a more integrated and scalable way.

This thesis offers an overall view of the free-to-play mobile game development, explaining the requirements, challenges, and opportunities that mobile game developers face when creating continuously evolving, service-like games. It presents the key concepts of the synchronous execution paradigm, illustrating how it addresses these challenges and provides a comprehensive framework for modern mobile game development.

Despite widespread adoption of this paradigm in the Finnish mobile game industry, it has not been formally documented or analysed. This thesis aims to fill this gap by giving a comprehensive overview of the paradigm, providing a foundation for developers to utilise it for developing scalable games that adapt to the ever-changing landscape of mobile gaming. While a specific implementation by Metaplay Oy is used as a case study, the concepts and principles discussed within the paradigm are broadly applicable to any choice of tools and technologies, as is evident by the multiple forms of the paradigm that have emerged in the Finnish mobile game industry.

The aim of this thesis is to articulate the synchronous execution paradigm and its utility in modern mobile game development. By providing a comprehensive framework for game logic execution, state management, and live operations, this paradigm addresses many of the challenges facing contemporary mobile game developers. It facilitates the creation of games that are robust, fair, and adaptable, offering solutions for maintaining game integrity, supporting complex monetisation strategies, and driving player engagement in an increasingly competitive market.

3. Online Games

Online connectivity has transformed video games from simple multiplayer setups to complex ecosystems influencing design, engagement, and business models. This chapter explores how server infrastructure affects both multiplayer and single-player experiences.

Mishra et al. [2] characterises online game architectures into three categories: "client-server, in which a centralised server manages the game world; peer-to-peer (P2P), in which peers share the management load; and hybrid, in which a centralised server handles sensitive and compute-intensive tasks, while others, such as local screen updates and optimisation techniques, are distributed to peers". They also discuss challenges each architecture faces: "client-server architectures suffer from low scalability, inferior fault tolerance, and high cost, whereas P2P architectures suffer from imperfect consistency control, higher cheating potential, and greater implementation complexity. Hybrid architectures partially cope with weaknesses from each approach, but don't entirely solve these drawbacks." The paradigm explored in this thesis aligns with the hybrid approach, combining server-authoritative logic with client-side execution to balance scalability, consistency, and responsiveness.

Contemporary online multiplayer is generally implemented as client-server architecture. Multiplayer networking in Valve Software's Source engine can be seen as a typical example where the server runs the whole game state simulation, while clients send inputs and receive updates [3]. Commercial game engines offer out-of-the-box multiplayer frameworks, such as Unity's Netcode for GameObjects [4], Epic Games' Unreal Engine Iris Replication System [5], and Godot's High-Level Multiplayer API [6]. These frameworks all offer similar tools for synchronising game state by replicating game objects (such as players, enemies, bullets, etc.) across the server and the clients, which is more modular and scalable than the Source engine's approach. However, these frameworks still rely on a centralised server to act as the authoritative source of truth for the synchronised game objects, and therefore the game state.

Multiplayer games require other services beyond game state synchronisation, such as matchmaking. These services are often implemented as separate microservices, communicating with the game server and each other. The aforemetioned game engines offer tools for implementing these

services, but developers also rely on third-party services. Free-to-play games have additional requirements, such as player inventories, in-game stores, and analytics, which are often implemented as microservices. These services are crucial for managing the game economy, player progression, and monetisation, and they often present challenges in infrastructure maintenance and cost management.

The evolution of online games mirrors broader digital trends, where backend servers extend and manage functionality. From social media to e-commerce, server-side operations are crucial for creating sophisticated, responsive, and personalised user experiences. Despite added complexity and cost, this reliance enables the dynamism and adaptability defining the modern digital landscape. Free-to-play game-as-a-service models exemplify this trend, offering ongoing content, events, and monetisation strategies to maintain player engagement and drive revenue [7].

Multiplayer gameplay ranges from asynchronous and turn-based interactions to real-time, immersive worlds. Technical approaches have evolved from lockstep mechanisms where player inputs are executed in strict order [8], to tick-based systems, where game logic progresses in fixed increments, and the game state is resolved based on what player inputs happen on each tick. More on tick-based simulation in Section 4.1.1.

3.1 Multiplayer Games

Multiplayer games have significantly evolved from the early days of networked gaming. This section explores various multiplayer approaches, their historical context, and their relation to the paradigm explored in this thesis.

3.1.1 Asynchronous Multiplayer

Asynchronous multiplayer originated from traditional play-by-mail games, where players took turns in games like Chess and Go [9] and sent moves via postal mail. In the digital age, this concept evolved to accommodate varied player schedules, especially in mobile gaming and for example a build-and-battle genre of games, where players build bases and attack each other's bases without the need for the defending player to be online simultaneously. Historically, these asynchronous games relied on simple server-side storage of game states, with minimal processing between turns.

Players submitted moves, stored until their opponent logged in to respond.

In the paradigm, server-authoritative logic (where the server executes all game actions resulting into the authoritative state, to which clients need to adhere to) ensures consistent game state across devices and sessions, allowing a more robust architecture to achieve asynchronous multiplayer with cheat-proof implementation. The Command Pattern [10] facilitates recording and replaying player actions, allowing more complex interactions than traditional "store and forward" methods. Tick-based simulations enable precise timing and synchronisation of game events, even when players are not online simultaneously, allowing more dynamic asynchronous gameplay.

3.1.2 Turn-Based Multiplayer

Turn-based multiplayer games have a long history in both digital and analog formats. Early digital implementations often used a simple "hot seat" (players performing their turns on the same machine) approach for local multiplayer or basic message passing for networked play.

As online gaming evolved, many turn-based games adopted a lockstep method for synchronisation, particularly in peer-to-peer setups. Each client executed their turn and transmitted the updated game state to other clients. While functional, this approach had drawbacks in terms of latency and potential for cheating.

The paradigm offers several advantages over these traditional approaches: server-authoritative logic eliminates the need for peer-to-peer synchronisation, providing a client-server architecture for executing the player turns. The Command Pattern allows for efficient representation and transmission of player turns between client and server, a more sophisticated approach than simple state transmission. Tick-based simulations provide a framework for managing turn order and timing, even allowing for simultaneous turns, a feature difficult to implement in traditional lockstep systems.

3.1.3 Real-Time Multiplayer

Real-time multiplayer games pose significant challenges for synchronisation and responsiveness. Two primary architectures address these challenges: the dedicated server model (i.e. client-server model) and the paradigm explored in this thesis. Both rely on tick-based simulations but differ in how game logic is distributed.

Dedicated Server Model

The dedicated server model is prevalent in low-latency, fast-paced games like first-person shooters. Here, the server executes the full game simulation, processes client inputs, and broadcasts state updates. Clients focus on rendering and input, often employing client-side prediction to reduce latency and smoothen the gameplay.

This model excels in enforcing server authority, supporting large player counts, and minimising client-side computational needs. However, it demands more server resources, uses significant bandwidth for constant state updates, and struggles with high-latency connections. It is also susceptible to input-based cheats, such as aimbots, where an additional software on player's machine steers the player's input with high accuracy.

Synchronous Execution

The synchronous execution uses client-server collaboration to share game logic. Both client and server execute game logic using the same tick-based simulation. The Command Pattern transmits player actions, with the server remaining authoritative, validating client results and correcting discrepancies.

This approach offers immediate responsiveness, reduced perceived latency, efficient bandwidth usage, and enhanced cheat-proofing. It suits deterministic multiplayer and replay systems. However, it requires clients to run the full game simulation, increasing computational demands, and can result in noticeable corrections in high-latency scenarios. It is less effective for games with very large player counts in a single instance, as the increased computation resulting from bigger number of players needs to be performed on both server and clients.

A high-level example of a real-time multiplayer game using the synchronous paradigm explored in this thesis is introduced in chapter 4.3.

3.2 Single Player Yet Online

The concept of a single-player game that operates online might initially appear contradictory, yet it aligns well with prevailing trends and business models in modern game development, especially in the free-to-play sector. By maintaining a mirrored player profile on a server-side database, developers retain full authority over progression and content, even in a nominally single-player environment. This architecture ensures that the

server holds the definitive record of each player's state and progress, allowing any discrepancies between client- and server-side computations to be detected and corrected.

Such a model proves particularly advantageous in preserving game integrity. In scenarios where players attempt to manipulate client-side data, perhaps by awarding themselves additional resources, the server, as the ultimate source of truth, can invalidate these attempts. This arrangement not only combats cheating but also guarantees the reliability of player data over time, as any corruption or loss on the client side can be rectified through reference to the authoritative server record. Consequently, this online component supports a vital goal for free-to-play games: the integrity of the in-game economy. When resources have a direct monetary value, ensuring that in-app purchases remain secure and untainted by unauthorised manipulation becomes essential for maintaining player trust and the viability of the overall business model.

Beyond these technical and economic considerations, incorporating an online component in nominally single-player games also opens the door to extensive live operations (the ongoing management and improvement of a game after its initial release). Developers can adjust game content, difficulty, or events in real time, delivering updates and personalised offers that reflect observed player behavior. By keeping the primary source of data and computation on the server, developers also gain valuable insights into player habits, preferences, and progression bottlenecks, enabling data-driven decisions about future features and content. Cheat-proofing is also achieved by maintaining the server as the ultimate source of truth, preventing cheating and ensuring integrity of the game economy. Player behaviour analytics, by tracking player actions server-side, provide insights into player behaviour, preferences, and pain points, informing future development decisions. Dynamic content updates allow developers to push new content, balance changes, and features to all players simultaneously, or based on individual players' prior behaviour. Cross-platform progress enables players to seamlessly continue their game across different devices, as their progress is stored server-side. Social features, even in single-player games, are enabled by online connectivity, allowing for leaderboards, friend lists, and asynchronous multiplayer elements.

While the online single-player model offers many benefits, it also presents challenges. Players may need a constant internet connection to play, which can be a barrier in some situations, even though the explored paradigm

offers some affordances for offline play. Developers must maintain server infrastructure, which incurs ongoing costs and requires regular maintenance. Once servers are shut down, the game may become unplayable, raising concerns about long-term preservation of these games as cultural artifacts.

3.3 Free-to-play Games

Free-to-play, or freemium, games have transformed the gaming landscape by providing access to high-quality games without upfront costs. This model relies on monetisation strategies like in-app purchases, advertising, and premium features to generate revenue. While offering the game for free lowers the entry barrier for users, it presents unique challenges for developers.

The monetisation of free-to-play games involves several key strategies. Players can buy virtual goods, currency, or premium features within the game through in-app purchases. Games may include banner ads, interstitial ads, or rewarded video ads. Some games offer a 'premium' version with additional features or ad-free gameplay. Time-limited progression systems, known as battle passes, offer rewards for completing challenges.

Developers of free-to-play games face several unique challenges. They must design engaging content to keep players interested over extended periods and create compelling incentives for purchases without making the game feel 'pay-to-win' (paying players gaining progress, which is seen as unfair advantage by non-paying players). Tailoring experiences and offers to different player types and spending habits is crucial. Regularly adding new content and features is necessary to maintain player interest, and constantly analysing player behavior and adjusting the game accordingly is essential for success.

Robust management frameworks in free-to-play games support rapid iteration by quickly implementing and testing new features or balance changes. They gather and interpret data on how players interact with the game for player behavior analysis. Dynamic content delivery is achieved by efficiently delivering new content and updates to players. Flexible monetisation allows for easy adjustment of pricing, offers, and monetisation strategies, sometimes based on the player behaviour analytics. Community management tools are essential for interacting with and managing the player community.

The paradigm explored in this thesis provides such a robust foundation for implementing these management frameworks, either through the synchronised logic execution, or the auxiliary services that can be built around the server-authoritative logic.

3.3.1 Game-as-a-service

Always-online, free-to-play games are part of a broader shift towards the games-as-a-service (GaaS) model. In this model, games are viewed not as standalone products but as evolving services providing ongoing content and experiences. This approach is being increasingly adopted by developers to maintain player engagement, drive revenue, and extend the lifespan of their games, not only on mobile platforms but across all gaming platforms.

The GaaS model represents a fundamental shift from the traditional game-as-a-product (GaaP) approach, marking "a departure from the studio-centric model focused on the release of new titles, towards a player-centric model based on updates, recurring revenues and player retention" [11]. Under this model, studios indeterminately support and periodically release content incrementally for existing games instead of developing new games or stand-alone sequels, generating more stable revenues while resulting in fewer new game releases.

The development process for GaaS projects follows a different trajectory compared to standard PC or console games. Instead of a linear process with discrete stages ending at game launch (design, pre-production, production, testing, distribution, and retail), GaaS emphasises post-launch activities where new content development and player retention efforts overlap. As Dubois and Weststar [11] note, "releasing a game to market is no longer the end but rather the beginning of ongoing service operations in which regular content updates, game optimisation, support, and community management are key to enduring success."

By leveraging online connectivity even in single-player experiences, developers can create more dynamic, engaging, and potentially profitable games. However, this approach also requires a commitment to ongoing development and maintenance, as well as careful consideration of player expectations and preferences. The GaaS model also poses challenges such as maintaining server infrastructure, ensuring data security, and balancing monetisation strategies to avoid alienating players.

The GaaS model offers several advantages. Continuous updates and content keep the game fresh and encourage players to remain engaged for

an extended duration. By consistently introducing new features, levels, events, and other content, developers can maintain player interest and prolong the game's lifespan. Data-driven improvements allow developers to gather player feedback and data to inform further improvements, making data-driven decisions to enhance the game experience. Flexible monetisation strategies, such as micro-transactions, battle passes, or subscription models, can potentially lead to more sustainable long-term revenue. Community building is fostered by the ongoing nature of GaaS, as regular updates and events give players reasons to come back and interact with each other, creating a more engaged and loyal player base. Rapid response to market is possible with the ability to update games quickly, allowing developers to respond to shifting trends, competitor actions, or emerging player preferences much faster than traditional game development models allow.

This shift has significant implications for game developers themselves. Research by Dubois and Weststar [11] suggests that working on GaaS projects requires developers with specific competency profiles, including "being comfortable with ambiguity and constant feedback, as well as displaying poise, initiative, and flexibility to account for an ever-changing work reality." The relationship with players also transforms fundamentally, as GaaS developers find themselves "on the front-line engaging with players, enabling co-creation, and building much deeper relationships" than in traditional game development where developers remain largely insulated from direct player interaction.

The GaaS model presents several challenges. Ongoing server costs can become a financial burden if the game is no longer profitable, raising questions about the long-term availability of the game for players. Game preservation concerns arise as server-requiring online games may cease to be available if the servers are no longer seen as financially viable to maintain, posing challenges for archiving and preserving these games for future generations. Player expectations and burnout can occur as regular updates and content releases are anticipated, and any interruption or delay in the service can lead to player dissatisfaction.

Development pressure is significant, as the need for constant updates can lead to issues such as crunch time and developer burnout. This is particularly pronounced in GaaS, where Dubois and Weststar found that developers consider themselves "more susceptible to burnout and extreme work situations than their GaaP peers" due to the never-ending cycle of

updates and fixes. The perpetual nature of GaaS development, described by one developer as "like running a marathon and being told to run another 5KM once you see the finish line. You run that extra 5KM, only to find that 2KM was added" [11], creates unique challenges for workforce sustainability.

The paradigm explored in this thesis provides several advantages for implementing the GaaS model effectively. It allows for efficient updates by pushing updates and new content seamlessly through the synchronised logic layer, without requiring extensive client-side changes. Reliable data collection is achieved through server-side tracking of player behavior and game metrics, facilitating data-driven decision making. Flexible monetisation is supported by easier implementation and adjustment of various monetisation strategies, facilitated by server-side tools. Cheat prevention is enhanced with improved security against cheating and exploitation through the syncronised execution model, maintaining a fair playing field.

3.3.2 Continuous Content

In GaaS free-to-play games, a continuous content pipeline is the processs in which new content for the ever-updating game is created. These systems allow developers to deliver new content, updates, and features without disrupting gameplay or requiring too frequent manual client updates. The paradigm explored in this thesis provides a robust foundation for implementing such a pipeline effectively.

The continuous content pipeline includes several stages, both in terms of development efforts and game server states. Content creation involves collaboratively developing new assets, features, and gameplay elements, often based on knowledge gained through player behaviour analytics. Integration of the new content into the existing game often utilises branched development version of the game server. Testing of the new content is done on another branched, more stable server, which aims to reflect the whole game with the new content. Deployment involves delivering content through client updates, configuration changes, and production server updates. The performance of the deployed update and new content is monitored using LiveOps systems and analytics. Community management and game analytics collects and examines data on player reception and engagement. Finally, iteration based on player reception and analytics leads to further improvements and adjustments, feeding eventually back into the next content update.

Updates through the continuous content pipeline allow developers to deliver updates and fixes to players almost instantaneously. The need for large, cumbersome client updates is minimised, as many aspects of the game can be fixed and tweaked through server-side configuration. Rapid iteration enables developers to quickly implement improvements based on player feedback and data analysis. The game remains evolving and adapting, providing a compelling long-term experience for players.

The paradigm explored in this thesis supports a continuous content pipeline by enabling seamless integration of new content through server-side configuration, minimising client-side changes. Player state on the server side helps developers gain trustworthy insights into player behavior and progress, informing the decisions on new content and updates. The server-authoritative logic ensures that new content is integrated into the game state accurately and consistently, and any problems with the new content can be quickly identified and fixed.

3.3.3 Cheat-proofing

Ensuring a fair and enjoyable gaming experience requires robust cheat-proofing measures. As Teittinen [12] notes: "Cheating in video games has always been a problem for the players and the game developers. The problem is not just limited to the open platforms like computer games but also to closed platforms like console and mobile games". By preventing data manipulation and vulnerability exploitation, developers maintain game integrity. The server-authoritative, synchronous logic execution paradigm introduced in this thesis provides a solid foundation for effective cheat-proofing.

Server-Authoritative Logic

While traditional server-authoritative designs, as Teittinen describes, often have "the game client [not running] any actual game logic but instead... immediately [sending] the actions made by the player to the server" [12], the explored paradigm implements what more closely resembles what Teittinen terms a "client optimistic server authoritative model." This hybrid approach maintains server authority while allowing deterministic game logic to run on both client and server.

Teittinen's analysis concludes that "the server-side protection methods are clearly more reliable to prevent cheating in games and they are also much harder to bypass or break" [12], which remains true in the explored

paradigm. However, unlike purely thin-client approaches, the synchronous model preserves responsive gameplay by executing logic client-side while ensuring the server remains the ultimate authority on game state. This approach aligns with Teittinen's finding that "the server authoritative model provides great protection against cheaters" [12] while addressing the latency and responsiveness concerns inherent in purely server-side execution models.

Synchronous Execution

The synchronous execution ensures both client and server run identical game logic, offering unique cheat-proofing benefits. Discrepancies between client and server outcomes are easily detected, identifying potential cheating attempts. As Teittinen explains, "To keep the game states synchronised between the client and the server, the game logic needs to be deterministic. This means that the game logic on the client and the server must always produce the same output when processing the game actions" [12]. The reduced attack surface limits opportunities for exploiting client-side vulnerabilities. Players receive immediate feedback through client-side execution, while their actions are also validated by the server.

Checksum Verification

Checksum verification is a key synchronisation technique in the explored paradigm, using checksums for game state altering action parameters and model states. A checksum, a unique identifier for a data block, is calculated and compared on both client and server sides to identify inconsistencies. Teittinen describes this approach where "the client can calculate a checksum value of the client's game state and send it to the server along the actions to prevent the game states going out of sync between the client and the server" [12]. Matching checksums validate actions, while mismatches prompt corrective actions, such as rejecting invalid actions or restoring the correct game state. This process enhances the gaming experience by ensuring fair play, and crucially for free-to-play games, maintaining a trustworthy and balanced in-game economy.

The checksum verification process works as follows: The client performs an action and calculates a checksum of both the action parameters and of the resulting game state, sending them to the server with action details. The server independently executes the action, calculates both checksums, and compares them with the client's. As detailed by Teittinen, "After processing the command's action, the server calculates a checksum value

from the player's game state that is stored on the server. If the action was marked as a ServerClient command, the server compares the checksum value to the checksum value that was stored to the command after the action was processed on the client" [12]. Matching checksums validate the action, while mismatches may indicate cheating or synchronisation issues. During typical operation of the game, while checksums match, there is very limited overhead on both computation and network communication.

However, when a mismatch is detected, the server can take corrective actions, restoring the correct game state, which may involve rolling back the game state to a previous consistent state. In Teittinen's implementation, "If the validation step failed or if the checksum values are not matching on the server, we assume that the player tried to cheat and reject the changes. If the changes are rejected, the server responds with a result indicating that a command was rejected and that the client needs to rollback to the last valid game state that is stored on the server" [12]. This resync process results in increased network communication and computation overhead, but ensures game integrity and fairness, while remaining mindful that "Especially on mobile devices the added latency from the client-server communication is noticeable. Also, constant communication between the client and the server can quickly drain the battery of the mobile device" [12]. This overhead is mitigated by the explored paradigm by allowing the synchronisation process to be performed between longer intervals, as the paradigm enables the client to run the game logic independently, and only synchronise the game state with the server immediately when more important game actions are performed, such as an in-game purchase. The synchronisation interval can be adjusted based on the game, and can be multiple seconds.

3.3.4 Social Connectivity

Social connectivity is crucial in modern mobile games, significantly enhancing their appeal, longevity, and overall player experience. The paradigm explored in this thesis supports social engagement by providing a reliable, server-centric environment for implementing player-connecting features. By ensuring a consistent game state across all clients and promoting a secure, cheat-free experience, the explored paradigm creates a robust foundation for various social features that enhance the gaming experience.

Key social features include leaderboards, cooperative gameplay, in-game events, friend systems, guilds or clans, and chat systems. The server-

authoritative design ensures accurate leaderboard implementation, fostering healthy competition by securely managing game state and player scores. Cooperative gameplay is facilitated by synchronising game events and player actions, promoting smooth interaction. Engaging in-game events, such as seasonal festivities and tournaments, bring players together for simultaneous participation.

The paradigm allows developers to implement these social features in a more maintainable fashion, compared to typical microservices or client-centric approaches. By centralising game logic and state management, these features can be implemented similarly to core gameplay mechanics, ensuring consistency and maintainability.

3.3.5 LiveOps

LiveOps, or "live operations," is crucial for managing and developing free-to-play and games-as-a-service models. As Prokkola [13] explains, LiveOps treats games as "lasting service[s]" rather than finished products, with development continuing well after launch. It involves continuously managing, updating, and improving a game post-release. According to Prokkola, LiveOps "can include all the changes to a product after the initial release of the application".

Activities include introducing new content, balancing gameplay, fixing bugs, running events, analysing player data, adjusting monetisation, conducting A/B testing, managing community engagement, and handling support queries. Prokkola [13] notes that successful events are "arguably the most important part of LiveOps" as they keep content fresh and players engaged. More about in-game events in chapter 3.3.8.

LiveOps relies heavily on data analytics to inform decisions. Prokkola [13] emphasises that the "data will optimally tell a story and a company needs somebody to translate that story to spoken language. This is the defining feature of product manager in the gaming industry". Through this data-driven approach, games can continuously evolve to meet player needs and market demands.

The paradigm explored in this thesis supports effective LiveOps strategies by providing a server-authoritative framework, which allows implementation of LiveOps features in the backend, while helping ensure a consistent game state across all clients. This approach ensures the game state is managed effectively, ensuring the game remains maintainable for the long lifetime that's characteristic of modern free-to-play games.

3.3.6 Configuration Over-The-Air

Configuration Over-The-Air (OTA) is essential in modern mobile game development and a cornerstone of effective LiveOps strategies. It enables developers to remotely update and configure game aspects without requiring manual client updates. This approach streamlines deploying new features, game balance adjustments, bug fixes, and content updates. The explored paradigm, with its server-authoritative logic and synchronised configuration system, facilitates efficient and robust OTA configuration.

A core concept in regards to OTA configuration is feature flagging [14]. Feature flags allow developers to: enable/disable features remotely, roll out changes gradually to specific user segments, conduct A/B testing (more in chapter 3.3.9), and revert problematic changes instantly. OTA configuration also allows developers to quickly address player feedback, emerging issues, and market trends. Players enjoy a seamless experience, accessing the latest improvements without interruptions. Reduced app store dependency is achieved by bypassing lengthy review processes for minor changes.

With the explored paradigm, server-side control allows seamless application of updated configurations, ensuring all players experience the same changes simultaneously. Consistency is guaranteed across the player base through the synchronisation of game logic and configurations, avoiding fragmentation from different client configurations. Predictable updates are facilitated by fixed-time tick-based simulations, providing a clear framework for applying changes at specific points in the game's execution.

In Metaplay Oy's implementation, Google Sheets is integrated into the OTA pipeline, in addition with a web-based dashboard. This integration allows developers to import game configurations directly from Google Sheets into the game server. The dashboard adds version control on top of the Google Sheets, making it easier to see differences between versions compared to Google's own version control. The dashboard also provides an easy way to revert to previous versions of the configuration, and to see which version is currently in use by the game server.

3.3.7 Player Segmentation

Player segmentation is a critical technique in modern game development, particularly for free-to-play and games-as-a-service models. It involves dividing a game's player base into distinct groups based on various char-

acteristics. These characteristics can include demographic information, in-game behavior, spending habits, progression, or engagement levels.

The goal of player segmentation is to better understand the diverse player base and tailor the game experience to maximise engagement and monetisation. By identifying different player segments, developers can personalise content, offers, and communication strategies. For example, new players might receive different tutorials or offers compared to veteran players. Similarly, high-spending players might be offered exclusive items or bundles.

The paradigm explored in this thesis supports effective player segmentation by providing the necessary data and flexibility. Within the server-authoritative framework, the shared game logic layer is crucial, where a complete player model exists on both the server and the client. This allows for robust tracking of player behavior and the storage of player data, as the server maintains the whole up-to-date state. This is essential for identifying and defining player segments. Furthermore, the ability to update game configurations and deliver content over-the-air enables developers to dynamically adjust the game experience for different segments. This includes A/B testing different approaches on specific player groups to optimise various aspects of the game, such as monetisation strategies or content delivery.

3.3.8 Eventful Gaming

Eventful gaming involves incorporating time-limited, in-game events that offer unique and engaging experiences for players. These events add variety and excitement to gameplay, providing new challenges, rewards, and interaction opportunities. The server-authoritative design of the paradigm allows developers to efficiently implement and manage these events while maintaining game state integrity.

The paradigm's server-authoritative design and synchronous logic execution provide an ideal foundation for implementing eventful gaming. Centralised control allows the server to coordinate and control events across all clients, ensuring simultaneous player experiences and fostering community. Real-time updates, utilising LiveOps and over-the-air configuration, enable seamless event management without client updates. Consistent event state is maintained through synchronous logic execution, preventing discrepancies from client-side manipulations. Flexible event management allows developers to adjust event parameters, duration, and

rewards in real-time, facilitating quick responses to player feedback or issues.

By leveraging the paradigm's capabilities, developers can create dynamic and engaging event-driven experiences that keep players invested in the game over the long term. The ability to rapidly deploy and adjust events allows for continuous refinement of the player experience, contributing to the game's longevity and success.

3.3.9 A/B Testing

A/B testing, or split testing, is essential in mobile game development for evaluating and optimising game elements. It involves comparing variations of a game element to identify the most effective one based on specific metrics. The paradigm explored in this thesis, with its LiveOps tools and OTA configuration, offers a strong foundation for implementing effective A/B testing strategies.

The paradigm's features support effective A/B testing. Server-authoritative logic enables controlled distribution of game versions to specific player segments. OTA configuration allows quick deployment of test variations without client updates. Synchronous execution ensures consistent behavior across clients in a test variant. Server-side data collection supports accurate performance metric gathering for each variant.

3.3.10 Online Dashboard for Game Management

The integration of logging, monitoring, LiveOps tools, OTA configuration, and A/B testing creates a complex environment requiring efficient management to maintain a high-quality gaming experience. An extensive online dashboard serves as a centralised solution for managing these aspects, providing developers with a comprehensive toolkit for game operations and analysis.

The online dashboard integrates seamlessly with the paradigm. The online dashboard can interact directly with the server, functioning as an interface for the server-side logic, configuration, and data management. It allows developers to monitor game performance, player behavior, and server health in real-time, providing developers with a view into the game similar to how players experience it on the client device.

Data-driven changes to the game state can be made through the server without a need for client updates, while changes to the logic commands will

require a new client version. Therefore implementing the game logic in a data-driven way is crucial for keeping the game easily updatable through the dashboard.

By leveraging a comprehensive online dashboard, developers can effectively manage the complex ecosystem of modern online games. This approach not only simplifies game operations but also enhances the ability to deliver high-quality, engaging experiences to players, ultimately contributing to the game's long-term success and sustainability.

3.4 Conclusion

This chapter has explored the multifaceted world of online games, examining how the integration of server infrastructure has shaped both multiplayer and single-player experiences. We've delved into various aspects of modern free-to-play game development, from the technical intricacies of different multiplayer modes to the challenges of maintaining persistent online worlds.

Key points discussed include the evolution of multiplayer games, from asynchronous to real-time interactions, and how the paradigm explored in this thesis addresses challenges in each mode. The concept of online single-player games leverages server authority to support modern business models. The rise of free-to-play games and the games-as-a-service model has fundamentally changed how games are developed, monetised, and maintained. The importance of continuous content delivery, live operations, and over-the-air updates is crucial in keeping games fresh and engaging. The critical role of cheat-proofing maintains fair and enjoyable gaming environments. The power of social connectivity features fosters player engagement and community building. The use of eventful gaming and A/B testing optimises player experiences and game performance. The necessity of comprehensive online dashboards is essential for effective game management in this complex ecosystem.

Throughout the chapter, we've seen how the server-authoritative, synchronous logic execution paradigm explored in this thesis provides a robust foundation for implementing these various aspects of modern online games. In the following chapters, we will delve deeper into the technical implementation of this paradigm, exploring how it can be applied to create robust, scalable, and engaging online game experiences.

4. The Paradigm

The paradigm explored in this thesis melds the dedicated server approach with the flexibility and scalability of cloud-based backend infrastructure, addressing the evolving needs of modern video games as discussed in Chapter 3. This approach is particularly well-suited to support the games-as-a-service model and the diverse multiplayer experiences ranging from asynchronous to real-time interactions.

In traditional dedicated game servers, the entire game engine, including the physics simulation, is run in a tick-based loop. While this provides accurate and synchronised gameplay, it can lead to significant computational demands and resource usage, hindering scalability. The paradigm addresses these issues by implementing a tick-based simulation of dedicated servers but diverges when it comes to the execution of the game engine. Instead of running the full physics simulation on the server, the paradigm utilises a distinct, tick-based logic model that is more optimised for cloud computing.

This decoupling of game logic from the full physics simulation not only reduces the computational demands on the server but also allows for more efficient resource allocation, thereby enabling greater scalability. The paradigm explored here implements what more closely resembles what Teittinen [12] terms a "client optimistic server authoritative model" (discussed in chapter 3.3.3). This hybrid approach maintains server authority while allowing deterministic game logic to run on both client and server. It's important to note that when physics simulation is needed, it can be compartmentalised and integrated into the logic system. This modular approach allows for flexible allocation of computational resources based on the specific needs of each game.

The paradigm harnesses the benefits of cloud computing, such as dynamic scalability and cost-effectiveness, while ensuring the responsiveness and consistency of gameplay characteristic of dedicated servers. As such, it combines the robustness of the dedicated server model with the agility and scalability of the cloud, to deliver a more cost-efficient and scalable solution for game servers.

A key advantage of this paradigm is its unified logic layer, which simplifies the implementation of various online and multiplayer features. Modern cloud-based backends often rely on microservices, wherein dif-

ferent services such as matchmaking, player accounts, or analytics are maintained as separate, loosely coupled modules. While this approach can excel in highly scalable or rapidly evolving systems, it can also introduce complexity through network overhead, service orchestration, and version management, reducing the maintainability and coherence of the game system.

In contrast, the paradigm described here aligns more closely with a monolithic architecture, offering a single, consistent framework where all of the game's features, from basic gameplay mechanics to complex social interactions, operate under one roof. This unified approach not only streamlines development and testing but also provides a more straightforward path for maintaining coherence across all game elements. By reducing the number of discrete services that need to be integrated, developers benefit from lower operational overhead and simpler debugging, ensuring that gameplay features remain consistent and easier to evolve over time.

Moreover, this paradigm provides a robust foundation for implementing key features of modern online games, including cheat-proofing, LiveOps, over-the-air updates, and support for eventful gaming. It also facilitates effective A/B testing and integration with comprehensive game management dashboards, addressing the complex needs of free-to-play and games-as-a-service models. By offering a unified logic layer, the paradigm enables developers to more easily implement and manage these features, resulting in a more cohesive and efficient game development process.

In the following sections, we will delve deeper into the theoretical underpinnings of this paradigm and explore its practical implementation, including how Metaplay SDK utilises this approach to create robust and scalable game systems.

4.1 The Theory

Traditionally any modern game or application with a cloud-based backend implements the client-side and server-side as completely separate logical entities, with their own choices of programming languages and patterns. The paradigm explored in this thesis takes a different approach, where identical logic code is executed both on the client and the server. The code is authored once, and either compiled directly to executables in the different ends (like in Metaplay's C# case with Unity C# client and .NET backend) or the logic code is transpiled from the client-side language to

one run on the server (e.g. C++ to Java or C++ to Scala). While some video games (such as first-person shooter computer games) typically run the whole game simulation, physics engine included, on the backend as a dedicated server, this approach is often computationally too expensive to run on a large-scale cloud-based backend.

Based on the author's experience working as a programmer in the Finnish mobile game industry, as well as casual conversations with other game developers, the one-code approach has been adopted several times by different game companies. The roots of this solution can be traced back to the early 2010s, when Java was a popular choice for servers. Additionally, wireless networks were not as reliable as they are today, and mobile devices were not as powerful, so the game logic was kept simple and executed on both ends. This allowed the game to run smoothly even in the face of network disruptions, as the client could execute the game logic locally without relying on the server. At the same time, Unity was becoming a popular choice as a mobile game engine, and it allowed developers to write the game logic in C#, eventually making C# a commonly used language for game development. Finally, as Microsoft introduced an open-source, cross-platform successor to .NET Framework, .NET Core in 2014 [15], it became possible to run C# code on linux servers, leading to the possibility of running the same code on both Unity client and the .NET server without the need for transpilation. This opportunity was utilised by Metaplay in 2019 to create a game development framework that allows developers to write the game logic in C# and run it on both the Unity client and the .NET server, thus implementing the true one-code approach, without transpilation. In 2020, .NET Core was renamed to .NET with the announcement of .NET version 5.0 [16], so the term .NET is used in this thesis to refer to the cross-platform successor of .NET Framework.

The one-code approach taken by the paradigm allows the developer to author the code only once, making the development process easier in multiple ways. A single logic code is easier to maintain compared to the two-sided approach, where the server code is a separate codebase entirely. Development work is also easier to conduct with an 'offline-mode', where the server end is simulated or mocked on the client, as the server is not a significant, separate entity in operating parts of the game, but runs instead the same logic code synchronously.

Following the paradigm, executing the same game logic code on both the client and server in a synchronous manner takes a different approach

to implementing cloud-based backend. Instead of the server acting as a completely separate component, the server's execution of the game logic acts as a trusted carbon-copy of the game logic, ensuring the developer that the results of any gameplay actions run by the server are entirely legitimate.

So, following the paradigm and executing game logic on both the client and server in a server-authoritative manner provides an enhanced control to developers, enabling them to effectively manage game states and promptly address any irregularities that may occur during gameplay. Additionally, server-authoritative execution allows for robust error handling and validation of game actions. The server acts as a central authority that validates and enforces the rules of the game, reducing the likelihood of unauthorised or invalid actions from the clients.

However, the use of cloud backend services to support server-authoritative gameplay does introduce additional costs. Running and maintaining the necessary server infrastructure can contribute to increased overheads for game developers. This includes expenses related to server hosting, data storage, network bandwidth, and ongoing maintenance and updates. Developers need to carefully assess the financial implications and ensure that the game's revenue model can sustain these additional costs. This is especially relevant when implementing the paradigm on a completely single-player game, where the backend would not be needed for gameplay features, but is implemented for the other benefits.

In the following sections, the key concepts of ensuring the synchronised state of the game logic following the paradigm are discussed. Maintaining a synchronised state, where the server and clients have a consistent view of the game simulation, is crucial for the paradigm. Changes to the game state are synchronised by utilising Command Pattern as atomic operations in a tick-based simulation model. Encapsulating each gameplay action as a discrete, self-contained state change in the logic model, the paradigm ensures that the state-changing actions are executed in a deterministic order. Error handling and validation of game actions are also intertwined with the commands, allowing the server to validate and enforce the rules of the game, reducing the likelihood of unauthorised or invalid actions from the clients. Network latency poses challenges to the synchronisation, which are mitigated and addressed through client-side prediction and server reconciliation, respectively. Lastly, the concept of embedding actions into the game state is discussed, which is a technique where state changes

are introduced from different possible input mechanisms. Such embedded actions are a potent feature especially for LiveOps, while they require careful consideration to ensure that they do not complicate the game state management or introduce inconsistencies.

4.1.1 Synchronised State

State synchronisation is a crucial aspect of the paradigm, as it ensures that the server and clients have a consistent view of the game simulation. Atomic operations in a tick-based simulation model are used to achieve this synchronisation. The main approach is to utilise Command Pattern in order to encapsulate each gameplay action as a discrete, self-contained state change in the logic model. Placing the commands into the tick-based simulation model, the paradigm ensures that the state-changing actions are executed in a deterministic order.

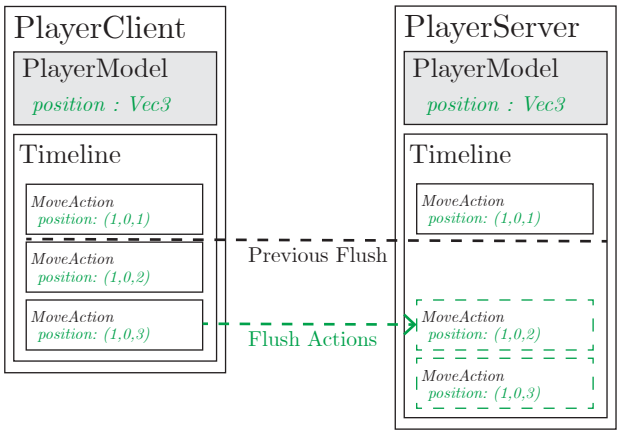

Figure 4.1. Logic action flushing to the server. The client periodically flushes the actions to the server, which then executes them against its own model. In a situation where the resulting model state is different from the client's model state, either a rollback or a reconciliation is performed to ensure that the client and server models are in sync.

One key mechanism that supports this synchronisation is the Command Pattern [10] (see figure 4.1). By encapsulating each gameplay action as a discrete, self-contained command object, developers separate the logic of performing an action from its invocation. This design choice not only simplifies debugging and logging but also makes it feasible to validate, replay, or roll back actions on the server. In single-player scenarios, the client can immediately execute commands, while the server processes the same commands for authoritative state management and cheat prevention. For multiplayer games, commands are exchanged between client(s) and server, ensuring that all participants observe identical state changes, thus

preserving fairness and consistency.

Command Pattern

The Command Pattern, originally introduced by the Gang of Four [17] and further elaborated by Nystrom [10], is a design pattern that allows function calls to be encapsulated as objects in object-oriented programming. In the context of the paradigm, the Command Pattern is a valuable technique for implementing atomic actions, supporting undo/redo functionality, and ensuring consistency in the game state across clients.

In the Command Pattern, function calls are transformed into objects, referred to as command objects. These command objects encapsulate the necessary information and logic required to perform a specific action or operation within the game. By reifying the state-changing function calls as command objects, the game logic becomes more modular and flexible.

One of the key benefits of the Command Pattern is its support for undo/redo functionality. By maintaining a command history on the server, individual actions can be easily tracked and undone or redone as needed. Clients can request the server to undo or redo a particular action, and the server can execute the appropriate command from the command history. This allows for a consistent game state across all clients, even in the presence of undo/redo actions.

Furthermore, the Command Pattern enables a logic-agnostic approach to executing commands. The game logic does not need to have prior knowledge of the specific actions performed by the commands. Instead, the game logic simply executes the commands as instructed. This separation of concerns allows for a more modular and maintainable codebase.

In a server-authoritative multiplayer game, the server can transmit the command history to new clients when they join the game. This ensures that new clients start with the same game state as existing clients and allows for consistent gameplay experiences across all participants. However, in practice, the server usually sends the current game state to the clients, but the pattern allows this kind of full replay, if it's desired. By storing the command chain, even a complex gameplay scenario can also be replayed later e.g. for re-validating competitive multiplayer results.

In a single-player game, the Command Pattern can be used to implement a server-side replay system, where the server applies the command history to recreate the player's actions. This can be useful for reducing the computation resources required on the server, as the server does not need to

simulate the game in real-time. Instead, it can replay the player's actions afterwards to verify the outcome of the game.

By utilising the Command Pattern, developers can design their game logic with clearly defined atomic actions, encapsulated within command objects. This approach facilitates undo/redo functionality, enables consistent game state management, and promotes a modular and flexible design. The Command Pattern is a powerful tool for implementing server-authoritative game logic, ensuring the integrity of the game state across all parties.

Tick By Tick

In game development, server-side game code execution is based on equal-length update frames, known as ticks. These ticks emulate frame-by-frame execution seen in real-time rendered games on client devices, ensuring a consistent gaming experience. This concept is also prevalent in physics engines used in game engines, where tick-based simulations with fixed time intervals are employed. Such simulations ensure uniform and precise behaviour across various devices [18].

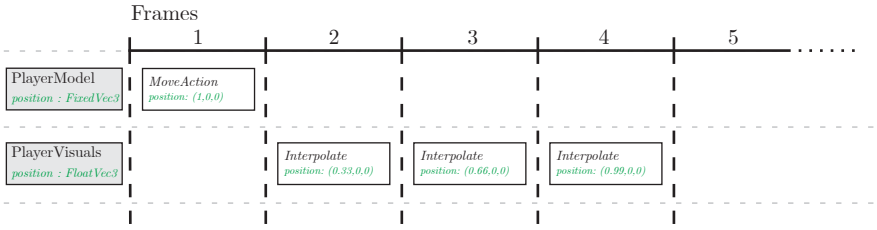

Figure 4.2. Separation of logic ticks and GPU-rendered update frames. Action is executed during frame 1, after which the visuals interpolate the player position over multiple frames for smoother movement. This simple case exemplifies how the logic ticks often represent distinct game states, while the GPU-rendered frames provide a smooth visual experience between ticks. The logic also uses fixed-point arithmetic to ensure determinism, as discussed in chapter 4.2.4, while the GPU-rendered frames can use floating-point arithmetic for visual rendering.

Client-side visuals are rendered in variable-time GPU update frames, which are determined by the GPU rendering speed to guarantee smooth visuals. In contrast, the logic code operates in fixed timesteps on both the client and server. This separation of fixed-length logic ticks from GPU-rendered update frames achieves smooth rendering for an optimal user experience. Additionally, it simplifies simultaneous execution on both client and server, reducing potential disparities.

The tick timestep is typically longer than the frame time of smooth rendering, which helps lessen the computational demands on servers (e.g., 15 ticks per second on the server, 60+ frames per second on the client). This arrangement does necessitate additional visualisation logic

on the client-side to bridge the gap between the more infrequent logic ticks. To illustrate, developers may use interpolation and/or extrapolation techniques to create smooth transitions between game states represented by each logic tick: between the logic ticks, the client can interpolate a player's position, resulting in a smoother movement. Or the client can try to predict the future position based on player's velocity, extrapolating the position from the last logic tick. However, this prediction might need to be scrapped and the true position of the player reconciled in the next logic tick (see chapter 4.1.2). While this adds complexity to the client-side code, it allows for greater scalability on the backend. This strategy exemplifies an effective optimisation approach in game development, striking a balance between user experience and server efficiency.

Atomic Operations

State synchronisation is crucial for any server-authoritative multiplayer game. It ensures that all players share a consistent view of the game world, even in the face of network latencies, packet loss, and other unpredictable factors. Two common strategies for achieving this synchronisation are delta encoding and atomic operations.

Delta encoding (or state synchronisation via deltas) involves sending only the incremental changes that have occurred since the last update, rather than transmitting the entire game state. For instance, when a character moves from point A to point B, only the new position data (the delta) is sent to the clients. This approach conserves bandwidth, an important consideration in environments with limited or expensive network capacity, yet it also requires that both server and client share a consistent baseline and apply each delta in the correct sequence. Packet loss or out-of-order delivery can complicate matters, as missed or jumbled updates may lead to divergent game states.

Atomic operations, by contrast, treat each action as a single, indivisible unit that updates the game state on the server. The server processes these operations in a defined order, ensuring that each operation is either executed fully or not at all, thereby avoiding partial updates. The result of each operation can be seen as a delta: once the operation completes on the server, the state change (or delta) is broadcast to all clients, who apply it to their local copy of the game state. While this may involve transmitting more data than delta encoding in some scenarios, particularly if operations occur in rapid succession, it simplifies the overall logic. Because each

operation encapsulates a higher-level action, such as a character “move” or “shoot,” the server can manage all sub-steps (like collision detection or resource consumption) in a single atomic command. If an error or conflict arises, the operation can be entirely rejected or rolled back, preserving the consistency of the game state.

In practice, atomic operations align well with a *tick-based simulation* model. Batching all incoming operations within discrete time steps (ticks) ensures that state changes are processed in a consistent, predictable order. This helps mitigate the effects of network latency by synchronising updates across players and, if necessary, rolling back to a prior tick to reapply missed or out-of-order commands. The ability to run ticks at varying frequencies also offers flexibility: fast-paced, real-time games may rely on high-frequency ticks for responsiveness, whereas single-player or slower-paced games can use less frequent ticks, reducing server load.

Overall, while **delta encoding** focuses on minimising the amount of data transmitted, the **atomic operations** paradigm emphasises correctness, predictability, and a clear order of operations. Each atomic operation represents a fundamental change in game state, processed authoritatively on the server, and then succinctly communicated to all participants. This high-level abstraction streamlines development, aids debugging, and helps maintain fairness by ensuring that all game-altering actions undergo consistent validation and execution in the same authoritative environment.

4.1.2 Reconciliation

To mitigate the effects of latency, techniques such as client-side prediction and server reconciliation are employed. Client-side prediction involves simulating the game state on the client using the inputs and actions received from the player, without waiting for the server’s response. This approach provides immediate feedback to the player and reduces the perceived effects of network latency. However, the client’s simulation may deviate from the actual game state on the server due to latency and other factors.

When the server sends the next update, the client adjusts its simulation to match the server’s authoritative game state. This process, known as reconciliation, involves comparing the client’s predicted game state with the server’s game state and reconciling any discrepancies. The client makes the necessary adjustments to align its simulation with the server’s, ensuring that the game state remains synchronised across all clients. The

client can use methods such as linear interpolation to smooth out the transition between the visuals and the correct logic state.

Under the paradigm explored, reconciliation not only ensures that the game state remains synchronised across all clients, but also supports the handling of actions originating from the past. Due to network latency or discrepancies in tick execution, it is possible for an action to arrive at the server after the corresponding tick has passed. In such a case, the paradigm allows the server to look back at the historical game state, apply the action, and reconcile the resultant state with the current one. This involves recalculating and applying all subsequent actions based on this new game state. This capability of the paradigm to "rewrite history" enhances the game's consistency, even under variable network conditions and latencies.

4.1.3 Error Handling

Error handling is a critical component of the paradigm, ensuring that the game remains fair and consistent for all players. When an error occurs, such as a network failure or a synchronisation issue, the server must handle the error and notify the client of the correct game state. To achieve this, game developers can add checks and error code return values to commands, catching errors and discrepancies with the game logic. For example, a command that moves a player's character might include checks for collisions with other objects, ensuring that the move is valid before executing. By using the Command Pattern with error handling, developers can write robust and reliable game logic code adhering to the paradigm.

The synchronous nature of game logic run on the server, according to the paradigm, brings an additional advantage: efficient and effective server-side logging and error handling. This setup allows the server to maintain a record of all game events, player actions, and system processes, making it easier to track, identify, and rectify errors or discrepancies in real-time.

In the event of an error, the server can immediately flag the issue and take appropriate corrective action, whether that involves resolving a conflict, rolling back an action, or updating the game state. Moreover, this synchronous approach allows for the collection of detailed server-side error reports, which provide valuable information to the development team about the exact circumstances and sequence of events leading up to an error. Such comprehensive logging and error handling capabilities significantly contribute to maintaining the robustness of the game simulation.

4.1.4 Latency

Latency, or the delay between a client sending a request and the server processing and responding to that request, is a critical factor in online gaming. It can significantly impact the player's experience, as delays can lead to sluggish responsiveness and disrupt the sense of real-time interaction. [19] To minimise the impact of latency on gameplay, developers employ various techniques such as client-side prediction, server reconciliation, and lag compensation (calculating latency to the server on the client and adjusting the predicted game state accordingly).

Client-side prediction is a technique used to provide immediate feedback to players while waiting for the server's results in a real-time multiplayer setting. Clients predict the game state based on their inputs and the last known state received from the server. This allows for smooth and responsive gameplay even in the presence of latency. The client continues to simulate the game state locally, displaying the predicted results to the player until the server's authoritative state update arrives.

Server reconciliation occurs when the server sends the updated game state back to the client. The client then compares its predicted state with the server's authoritative state. If discrepancies are detected, the client must reconcile these differences to ensure a consistent game experience across all players. Rollback strategies are often employed in this process.

Rollback strategies involve rolling back the client's game state to the last known authoritative state received from the server and re-simulating the game state using the corrected information. This synchronisation process resolves any conflicts between the client's predictions and the server's authoritative state. By reconciling these differences, the game state is aligned, and all players perceive a consistent game world.

Implementing client-side prediction and server reconciliation can be challenging. One key challenge is dealing with the uncertainty of network conditions, as latency can vary depending on the player's location, network infrastructure, and server load. Developers must carefully balance the client's ability to predict the game state accurately while also avoiding excessive rollback and re-simulation, which can lead to visual glitches and inconsistencies.

4.1.5 Embedding Actions

The paradigm's server-authoritative, tick-based architecture is well-equipped to incorporate external actions, rendering it a potent feature. It provides an organised and efficient mechanism to seamlessly blend in-game events with external inputs, such as LiveOps actions, to enrich the gaming experience. However, this embedding of actions originating outside the simulation requires careful management to ensure game consistency and player experience. These actions can be varied, from dynamic content updates, player reward distributions, or game event activations, and typically require immediate or scheduled implementation.

A common approach to embed such an action is by queueing it as a server-side command. As part of the normal game loop, each tick can include a step where the server checks for any new external actions queued for execution. This queue could be populated through the LiveOps dashboard, an API, or other server-to-server communication. If an action is present, the server treats it like a regular command within the tick timeline, processes it, and ensures its effects are reflected in the game state both on the server and the clients.

One of the critical factors here is timing. Since the server operates on ticks, the server must handle the timing of these external actions. For example, if an action is scheduled to be executed at a particular time, the server can convert this time into the corresponding game tick. By doing so, these actions can be accurately integrated into the game's timeline, ensuring game state consistency.

Additionally, the server needs to handle edge cases. For example, what happens if an external action conflicts with an in-game action? Proper error checking and handling mechanisms need to be in place to deal with such situations. This might involve prioritising certain actions over others or providing a way for the LiveOps team to manually override certain in-game actions if necessary. It's important to note that since these actions come from outside the standard game client-server model, extra caution should be taken regarding validation and security to maintain the integrity of the game environment.

4.2 The Implementation

Developing a free-to-play mobile game that's developed and operated as a service requires a robust and scalable backend infrastructure. This holds true even for most single-player games, which are now server-run to enable balance adjustments, updates, and protective measures against cheating and hacking. To develop and operate these games, a robust set of technology, tools, and logic are required in the background, so the next section will explore the practical implementation of this paradigm in the context of Metaplay SDK.

Developing such technology from scratch is a significant undertaking, often requiring skills distinct from those used in game development. This is why game engines like Unity 3D and Unreal are so popular; utilising a pre-existing development environment with comprehensive tools is generally more efficient than building something from scratch.

The implementation of the paradigm by Metaplay utilises a collection of tools and libraries for developing online-enabled games using the Unity game engine. It caters to a broad spectrum of game types, from simple single-player games with server-persisted state, to full real-time games with intricate social interactions.

4.2.1 Logic Code

Programming free-to-play mobile games that incorporate sensitive features such as real-money transactions requires a unique architecture where every gameplay step is validated on the server. Metaplay's implementation of the paradigm through Actions, Models, and Configs facilitates the development framework to address these needs.

Under the paradigm, the server's version of the synchronous logic code holds the authority over the game state, effectively reducing the risk of client-side tampering. *Models* represent logic data for entities shared between the client and server, like the *PlayerModel* reflecting a player's state. *Actions* implement Command Pattern for applying change to these Models and can be triggered by player inputs, server events, or other occurrences. *Game Configs* specify the game's design data, configuration, and economy data, usually defined in spreadsheets like Google Sheets. *Ticks* track the elapsed time between Model updates on both the client and server, ensuring Actions are executed in correct order.

Metaplay SDK utilises C# attributes [20] heavily for helping developers

define Models, Actions, and Configs. Attributes are a way to add metadata to C# code, allowing developers to annotate classes, methods, and properties with additional information. This metadata can then be used by the Metaplay SDK to generate the necessary code for serialisation, deserialisation, and e.g. define the Actions that can be performed on the Models.

In single-player applications, actions are executed immediately on the client for maximum responsiveness and then sent to the server for validation. While running servers comes at a cost, it provides developers complete control over what happens on client devices. For instance, when a player makes a move in a single-player game, it is first executed on the client device, then mirrored on the server. This process safeguards the game's integrity, preventing altered states from hacked clients from propagating to the server.

4.2.2 Game Configurations

In addition to the identical code run on both client and server, the game's static configuration is also authored once and deployed to both client and the server. In order to keep the configuration in-line with both ends, various approaches are taken. Firstly, the authored configuration data is serialised in a format that's easy to process by the client and the server, even in the cases where different programming language is used. Secondly, versioning system is used to ensure client and server are running on the same configuration as the basis of their logic execution. Additionally over-the-air configuration is often implemented to update this basis separately from the compiled logic code, allowing greater control and flexibility in controlling the runtime environment.

The synchronous logic environment consists, therefore, of two separate layers: the configuration, and the logic code. The configuration defines various aspects of the game, such as gameplay rules, level design, character attributes etc., while the logic code defines the atomic actions (in the form of commands) that the player can perform in order to influence the game logic.

To facilitate the management and implementation of game configurations, developers often utilise spreadsheets, such as Google Sheets, as a convenient and flexible tool. These spreadsheets serve as a central repository for storing and organising configuration data.

The process of integrating game configurations into the game logic in-

volves several steps. Firstly, developers define the schema or structure of the configurations in the game's codebase. This schema defines the data types, fields, and relationships within the configurations.

Next, the configuration data is entered and organised in a spreadsheet format, often using tools like Microsoft Excel or Google Sheets. Each sheet in the spreadsheet corresponds to a specific category of configuration, such as enemy stats, level details, or in-game rewards. The spreadsheet allows designers, level builders, and other team members to input and update configuration data in a user-friendly and collaborative manner.

To transfer the configuration data from the spreadsheet to the game runtime, developers employ serialisation techniques. Protobuf (Protocol Buffers) is a commonly used serialisation format in mobile game development, while Metaplay implements a custom configuration system, purpose-built and optimised for the C# environments in Unity and .NET. Metaplay's configuration system is designed with C# in mind, making the data schema definition easy through C# attributes. The attributes are used to define the serialisable fields, types, and relationships within the configuration data, allowing for easy authoring and parsing of the data.

During runtime, the game logic utilises reflection or similar mechanisms to parse the encoded configuration data. Reflection enables the game logic to dynamically examine and manipulate the structure and values of the configuration data based on the predefined schema. This approach allows the game to adapt to changes in the configuration without necessitating extensive code modifications or recompilation. By generalising the configuration parsing code, the game can easily accommodate new fields in the data and columns in the spreadsheet. Adding new data or modifying the data structure only requires changes to the schema and the new data, eliminating the need for authorship of custom parsing code every time the data structure changes. Although reflection still requires recompiling the code, it greatly simplifies the process of integrating and updating game configurations.

By utilising spreadsheets, serialisation formats like Protobuf, and reflection, developers can efficiently manage and integrate game configurations into the game logic. This approach separates the configuration data from the codebase, empowering designers and content creators to easily update and iterate on the game configurations. It also streamlines the process of configuring and fine-tuning the game without the need for extensive coding changes, enabling rapid iteration and optimisation.

4.2.3 Data Serialisation

Data serialisation is the process of converting data structures or objects into a format that can be transmitted over a network or stored in a file. In games employing the paradigm, data serialisation is essential for transmitting game state and command information between the client and server. For this purpose, Metaplay offers a powerful and easy-to-use built-in serialiser.

Designed to minimise boilerplate, Metaplay's serialiser operates by assigning C# attributes to types requiring serialisation. The game logic types (e.g. classes like PlayerModel) themselves are utilised for serialisation, eliminating the need for additional copies of types (e.g. Data Transfer Objects utilised generally in .NET development [21]). The serialisation code is programmatically generated, circumventing laborious and error-prone manual implementation. This automated approach ensures that the serialisation code is always in sync with the game logic types, reducing the risk of discrepancies and errors. Without such automation, developers would need to manually write and maintain functions for serialising and deserialising each type, which is error-prone and time-consuming.

Metaplay's serialiser is designed to be user-friendly. By simply tagging game logic types with the appropriate attributes, the serialiser takes care of the rest, eliminating the need for external tools or additional message specifications. Furthermore, it's multipurpose, using the same format to facilitate communication between client and server, as well as for data persistence in the database. Attributes can dictate which data is preserved in the database and which data is shared between the client and server.

In terms of data format, the serialiser utilises a binary format inspired by Google Protocol Buffers (protobuf), providing a representation that is 3-5 times more compact than JSON. This design choice results in reduced network bandwidth and storage requirements. Moreover, the serialiser is efficient, with all serialisation code generated and compiled using the Roslyn compiler. For Unity IL2CPP targets, the serialiser code is generated as a build step and included in the final build as a .dll file.

The serialiser ensures backward compatibility by tagging all types and members in the serialised data with metadata, enabling the addition or removal of members. While support for more complex changes does require hand-written migration code, the serialiser also allows stand-alone decoding, as the serialised data contains enough metadata to be traversed

and printed for debugging purposes.

Developers can also opt for standardised formats such as JSON or protobuf to ensure cross-platform compatibility and efficient data transmission. JSON and protobuf are both extensively supported across various programming languages and platforms, making them suitable choices for cross-platform compatibility in multiplayer games.

4.2.4 Fixed-Point Numbers

Game engines typically rely on floating-point arithmetic for computing the game simulation and rendering. However, floating-point arithmetic is can be non-deterministic across different compilers or platforms, making it unsuitable for the computation required for deterministic and synchronous game logic execution between the client and server platforms. In contrast, fixed-point numbers offer a deterministic and consistent way to represent decimal values, making them a more appropriate choice.

Fixed-point numbers are a representation of real numbers where a fixed number of digits are reserved for the fractional part, and the remaining digits represent the integral part. Unlike floating-point numbers, fixed-point numbers have a fixed number of decimal places, which ensures consistency in calculations and eliminates the rounding errors associated with floating-point arithmetic.

With the aim of achieving deterministic computations, Metaplay SDK uses the open-source FixPointCS [22] math library to provide fixed-point functionality. The library includes *F64* (a 32.32 signed fixed-point number) and *F32* (a 16.16 signed fixed-point number), along with vector types *F32Vec2*, *F32Vec3*, *F64Vec2*, and *F64Vec3*.

As a general rule, *F64* should be used as the default due to its larger range of values and accuracy. However, in situations where performance is of great concern, the *F32* family of types can be used. It should be noted that *F32* has a much smaller range and precision of values, so care should be taken to avoid any over- and underflows.

A noteworthy feature is that the fixed-point types are integrated into the game config parsing, which means they can be used in configuration spreadsheets just like floats. The file importing process automatically detects any fixed-point types and invokes the correct parsing function.

The primary advantage of using fixed-point numbers in game simulations is their deterministic nature. Given that fixed-point arithmetic adheres to strict rules and precision, the same set of operations performed on fixed-

point numbers will always yield the same result, a property crucial for ensuring consistent logic between the client and the server.

4.2.5 Deterministic Collections

In the paradigm, consistent game state data is critical, necessitating deterministic and order-preserving data structures. Standard C# collections like `HashSet<Key>` and `Dictionary<Key, Value>`, often used for storing game state data, use `HashCode` for internal organisation, impacting the iteration order. While not an issue within a single process where `HashCodes` remain stable, it can lead to subtle bugs in distributed computing scenarios.

Consider an algorithm iterating through a set of players and selecting the first fulfilling a specific condition. If multiple players fulfil the condition, different container orderings may cause clients and server nodes to select different players, potentially leading to an out-of-sync situation.

To enforce consistency, collections' iteration orders need to be explicitly or implicitly defined. For explicit order, .NET framework's `SortedSet<Key>` or `SortedDictionary<Key, Value>` can be used, iterating in sorted key order. However, it requires the `Key`s to be `IComparable`, which can be tedious and error-prone to implement for types that are not naturally comparable. [23]

For implicit ordering, the Metaplay SDK provides drop-in replacements for .NET native collections. These collections iterate items in the order of their insertion into the collection. For instance, inserting elements A, B, and C in order results in the iteration order of A, B, C. This mechanism allows maintaining the synchronicity of the game state across the client and the server, an essential aspect of the paradigm.

4.2.6 Message Passing

Message passing is a fundamental mechanism for exchanging information between different parts of a distributed system [24], such as the client and server components in an online game. In Metaplay's implementation of this paradigm, **Akka.NET** [25] serves as the foundation for message handling, providing a robust actor-based infrastructure that encapsulates communication logic and manages concurrency.

At its core, a *message* is a container of data transmitted between actors (e.g., between the client and server, or among server's services). These messages are intentionally lightweight, designed to carry only the information necessary for the recipient to interpret and process the request. They

are not responsible for executing any game logic; rather, they act as the transport mechanism.

In contrast, *commands* encapsulate both the parameters of an action and the logic needed to execute that action. In this paradigm, commands are written in a shared code domain, meaning that both senders and receivers know how to instantiate, serialise, and execute each command. When the client issues a command (e.g., “move character” or “perform attack”), it packages that command into a message, calculates checksums, and sends the package to the server. The server then deserialises the command, executes it, and compares checksums.

To accommodate the paradigm’s broad scope, messages are subclassed into specific types that clarify their purpose and origin:

- **Client-to-Server Messages:** Carry player inputs or actions (in the form of commands) from the client to the server.
- **Server-to-Client Messages:** Relay the authoritative outcomes of commands, along with state updates or acknowledgments, back to the client.
- **Action (Command) Messages:** Contain the core logic for modifying game state (e.g., moving characters, updating resources), along with any parameters required for execution.
- **Model Tick Updates:** Convey progress in the simulation timeline, ensuring that the server and clients remain synchronised regarding which tick (or timestep) they are processing.

A notable feature of this paradigm is the tight integration between messages and the tick timeline. Each tick represents a discrete update interval in the game simulation. Commands that arrive during a tick are batched, validated, and then executed atomically at the end of that tick. By pairing each message with a specific tick identifier or timestamp, the system guarantees the correct order of operations even in the presence of network latency or asynchronous processing. Consequently, actors can roll back to a previous tick and replay any missing commands if necessary, helping maintain a coherent game state across all nodes.

To safeguard data integrity and mitigate cheating, messages may include checksums of the execution results. Upon receiving a message, the server (or client, depending on the direction) verifies that the checksum aligns with the expected result of the operation. This prevents tampering, since any alteration to the command’s parameters or serialised data would invalidate the checksum and prompt the system to reject or request a

retransmission of the message.

By leveraging Akka.NET's actor model, the implementation benefits from encapsulated concurrency and fault tolerance. Each actor handles messages asynchronously, processing them in a thread-safe manner, which simplifies both scalability and maintenance. The inheritance structure of message types streamlines development: developers can clearly differentiate client-to-server commands, server-to-client updates, and synchronisation signals related to tick progress.

4.2.7 Actors Running the Show

The Metaplay SDK leverages Akka.NET actors [25] to run game models on the server.

An actor in Akka.NET is an autonomous, stateful object communicating exclusively through message-passing. It operates on its own lightweight thread, shielded from the rest of the system, eliminating the need for synchronising access using locks. This isolation makes it possible for developers to write actor code without worrying about concurrency. Each actor encompasses State, Behavior, a Mailbox, Children, and a Supervisor Strategy, all represented via an Actor Reference.

An Entity Actor, a Metaplay SDK implementation of Akka.NET actor, responsible for implementing the behavior of a Game Entity on Metaplay, can be categorised based on their lifecycle into Ephemeral and Persisted entities.

- **Ephemeral Entity:** This entity lives for its runtime, with its state and unique EntityId discarded upon the termination of the entity instance.
- **Persisted Entity:** In contrast, a persisted entity retains its state in persistent storage when idle. Upon creation, it receives an immutable EntityId, which refers to this particular entity thereafter.

Entities interact through message exchange. While sending a message, the sender specifies a destination entity id and payload. The system delivers the message to the destination entity, activating it if necessary. However, communication is inherently unreliable; messages are dropped with no retries if the destination entity doesn't exist or if an error occurs. To ascertain if the message was processed, the recipient must return an acknowledgment message.

Metaplay enhances this fundamental messaging feature with a PubSub system, allowing entities to subscribe to another entity's topic. When a

message is published to a topic, all subscribing entities receive it.

4.2.8 Models For Synchronisation

The discussion on atomic operations in chapter 4.1.1 provides an important context for understanding the role of models within the game server architecture. In particular, models facilitate the implementation of these atomic operations, called *Actions* in Metaplay SDK, and the maintenance of a server-authoritative game state.

Models in the Metaplay architecture represent the structural aspects of the game entities and their interactions. They provide a framework that defines and contextualises the data that represents each entity within the game. This could include data such as a player character's health, location, inventory, and so on.

Actions correspond directly to manipulations of these models. For instance, a *move* operation might alter the *position* attribute of a player character model. The result of these operations, or the *deltas*, represent changes to the game's models. These actions, in turn, are transmitted between the server and the client to keep the game state synchronised.

Each action is tied to a tick in the game server's tick-based simulation. This arrangement ensures that actions are executed in a deterministic and consistent manner across all parties, and it allows for efficient processing and validation of player actions. By grouping several operations within one tick, the game server can manage its resources more efficiently, reducing the frequency of state updates sent to clients and thereby decreasing network bandwidth usage.

The ticks also provide a framework for handling issues like packet loss or out-of-order delivery of operations. If such issues occur, the game state can be rolled back to the previous tick, and operations can be replayed to ensure consistency. The deterministic nature of these operations, governed by the models, enables this functionality.

In this way, models serve as the fundamental building blocks for the game state and the atomic operations that manipulate it. They define the parameters within which game interactions occur and provide a consistent framework for maintaining and synchronising the game state across multiple devices.

4.2.9 In-Memory Player State Management

In the Metaplay's implementation, a model is loaded into memory for (Akka) Actor to operate on. A significant benefit with this is the loading of player state into memory whether a client connects into the game, or when the player's profile is opened on the online dashboard.

By having the player state readily available in memory on the server, game administrators and support staff can easily modify or update the state using player support tools in the dashboard. This streamlines the process of making adjustments to a player's progress, inventory, or other game-related data, eliminating the need for complex database queries or direct database access.

The in-memory player state management also allows for a *migration when loaded* approach with the database. When the player state is loaded into memory, any necessary database migrations or schema changes can be applied immediately, ensuring that the state is always up to date with the latest version of the game. This simplifies the deployment of updates and maintains data consistency across different versions of the game, reducing the complexity of managing database migrations. It also eliminates the need for expensive and time-consuming migrations for the entire database at once, as the migration can be performed on a per-player basis when their state is loaded.

With the player state accessible in memory, scan jobs or batch operations can be performed on the states using the same code path as the migration and other actions. This enables consistent and reliable execution of tasks such as analytics, data aggregation, or applying global changes to all player states. By leveraging the same code path, the system maintains coherence and robustness in these operations.

Loading player state into memory also contributes to improved performance by reducing the need for frequent database queries or disk I/O operations. With the state readily available in memory, the server can quickly access and manipulate the data, resulting in faster response times and a smoother gameplay experience for the players.

4.3 High-Level Example: A Space Shooter

To illustrate how this paradigm can be applied and how Metaplay SDK operates, consider a simple multiplayer space shooter game. In this setting,

multiple players control ships in a 2D environment, firing projectiles and attempting to eliminate each other in real time. The following subsections provide a description of how the game's logic and components are organised using the Metaplay SDK.

4.3.1 Shared Code

The shared code encompasses the core data structures and actions that both client and server can access. This arrangement ensures consistent logic and data validation across all parts of the system.

Game configurations are also serialised into the shared codespace, defining gameplay rules, ship attributes, and other parameters. This shared configuration is used by both the client and server to maintain a consistent game state.

Player Model Defines the attributes and state of a player in the game's meta layer (the state that persists across matches). This includes data such as the player's username and accumulated rewards from battle sessions.

`Player Model` also contains information for LiveOps events, such as A/B testing segments, promotional offers, in-game events or other dynamic content.

Player Actions A set of actions that the client can invoke, including actions modifying their `Player Model`, joining the matchmaker, and performing actions in the battle session.

- `Join Match Action` Represents the player's request to join a match. This action is sent to the server, which assigns the player to an appropriate battle session based on matchmaking criteria.
- `Player Input Action` Represents user inputs encapsulated in a command object. This action mirrors a traditional real-time multiplayer game running the physics simulation, with the client sending its inputs on each simulation tick. The server then processes these inputs and solves the game simulation, which is later sent to the client. While each client can visualise their own inputs immediately, the server's authoritative state is the final arbiter of the game's state. Other players' inputs are not processed, but their state is updated based on the server's simulation.
- `Shoot Action` Encapsulates the firing logic and parameters of a projectile, such as its origin, direction, and velocity. The projectile's movement and collision detection are handled by the server during

`Battle Model`'s simulation tick. The server then sends the updated game state to all clients, ensuring consistency across all players.

- `Leave Match Action` Represents the player's request to leave a match. This action is sent to the server, which removes the player from the battle session and updates the game state accordingly.

When authoring the actions, Metaplay SDK's utilisation of attributes comes in extra handy. By tagging the action properties with the appropriate serialisation attributes, the SDK can automatically generate the serialisation code for the actions. This feature significantly reduces the amount of boilerplate code required to implement the actions, streamlining the development process and ensuring consistency between the client and server implementations. The attritubes also designate intended targets for the actions, ensuring that the actions are only processed by the appropriate actors.

Battle Model Maintains the overarching state of an ongoing match or battle session, including metadata about active players, remaining time, and any relevant scoring or victory conditions.

Each player client maintains a local copy of the battle model, which is periodically updated by the server to ensure consistency across all clients. Server actor is responsible for managing the authoritative state of the battle model, processing player actions and running simulation ticks, and sending updates to all relevant clients. Client's local copy of the battle model is used to calculate the validity and outcome of player actions, and to render the game state visually.

Simulation tick loop is the core of the battle model, where the server processes player actions, simulates the game state, and sends updates to all clients. Projectile movement, collision detection, and player interactions are all handled within this loop, ensuring that the game state remains consistent and synchronised across all clients.

In order to keep the battle state as compact as possible, only the necessary attributes for players and projectiles are included in the model. For this, the `Battle Model` implements an internal `Player Battle Avatar Model`, which contains only the essential player attributes required for the battle session. This is a crucial optimisation, as Player Model can contain much more data than what is needed for the battle session, such as player statistics, inventory, and other game progress data.

4.3.2 Server Side

On the server, Akka.NET actors in a .NET environment handle authoritative logic, concurrency, and data integrity. These actors enforce the rules of the game and communicate results back to clients.

Session Actor Manages the connection between the client and the backend, handling the initial handshake, authentication, and session management.

The `Session Actor` is responsible for establishing PubSub channels between the `Player Actor` and other server-side actors. It also facilitates the communication between the client and the server, ensuring that messages are delivered and processed correctly. The routing of messages (and therefore also actions) to the correct actors is facilitated through Metaplay SDK's built-in attributes, which designate the intended target for each message.

Player Actor Manages the in-memory state of the corresponding `Player Model`. It processes incoming `Player Actions` and updates the player's state accordingly.

The `Player Actor` also persists the `Player Model` state to database, ensuring that player progress is saved and can be retrieved across sessions.

In Metaplay SDK, the `Player Actor` is loaded into memory when a player connects to the server, as well as when the player's profile is accessed on the dashboard. This approach streamlines the process of updating player data for player support tools, applying database migrations, and performing batch operations on player states.

Matchmaker Actor Coordinates matchmaking logic to place players into appropriate battles, creating or joining instances of `Battle Actor` as needed.

When a player's `Session Actor` establishes a connection to the `Matchmaker Actor`, the latter assigns the player to a battle session based on their skill level or other criteria. A connection between the player's `Session Actor` and the `Battle Actor` is then established, allowing the player to participate in the battle session.

Battle Actor Oversees the battle session itself, executing commands that affect multiple participants (e.g., collisions between projectiles and ships). It maintains the authoritative state of the `Battle Model` and sends periodic updates to all relevant `Session Actor` instances.

The `Battle Actor` is responsible for persisting the `Battle Model` state, ensuring that a battle can be resumed or replayed if necessary. It also handles

the reconciliation of player actions and the simulation of game events such as projectile movement and collision detection.

At the end of the battle session, the `Battle Actor` is responsible for persisting the final result of the battle, adding the necessary results from each `Player Battle Avatar Model` to their respective `Player Model`. This happens via each player's `Session Actor`, which routes the results to each `Player Actor`. The `Player Actor` then updates the player's `Player Model` with the results and saves it to the database.

4.3.3 Client Side

On the client, Unity is responsible for rendering visuals and handling local user interactions. The client renders the game state from its local copy of the `Battle Model` and sends user actions to its `Session Actor`, which forwards them to either the `Player Actor`, the `Battle Actor`, or the `Matchmaker Actor` on the server.

Connection Handling The client establishes a connection to the server via the `Session Actor`, which manages the session and communication between the client and the server. The `Session Actor` also handles the initial handshake and authentication process, ensuring that the client is connected to the correct server instance. In case of connection loss or logic mismatch, the `Session Actor` can also handle reconnection and resynchronisation.

Game Config The client loads the game configuration data either from serialised data in the game build, or from the server via an over-the-air configuration update. This data includes game rules, player attributes, and other parameters that define the game's behaviour. The server is responsible for resolving A/B testing segments, promotional offers, in-game events or other dynamic content before sending the configuration data to the client.

Game Menus Provides the player with options to navigate the game, access their profile, and invoke `Player Actions`. The client visualises the player's state based on the local copy of the `Player Model`.

Battle Visuals Translates state data from the local `Battle Model` into on-screen graphics, including projectiles and any other simulated game elements.

Player Visuals Renders each player's visuals based on their state in the `Player Battle Avatar Model` in the local `Battle Model`.

4.3.4 Logic Flow

The following steps outline the typical flow of events in the game:

1. **Player Connection:** A player connects to the server, server starts a `Session Actor` that manages the player's connection and session. This triggers a `Player Actor` to be started and `Player Model` loaded into memory. The player can then access their profile and join battles. Client visualises the player's state based on the local copy of the `Player Model`. The player can invoke actions related to `Player Model`, which can be immediately processed by the local model. The `Player Actor` processes these actions and updates the player's server state accordingly, persisting it to the database periodically.

2. **Matchmaking:** The player client executes a `Join Match Action`, which is routed to the `Matchmaker Actor` by the `Session Actor`. The `Matchmaker Actor` assigns the player to an appropriate battle.

3. **Battle Session:** The `Matchmaker Actor` invokes creation of a `Battle Actor` with a fresh `Battle Model` to manage the battle session. The player's `Session Actor` connects to the `Battle Actor`, so player's battle-related actions are routed correctly. Once the battle starts, the `Battle Actor` starts to execute ticks on the `Battle Model`, processing player actions and updating the game state.

4. **Gameplay:** The player sends inputs to the server, which are processed by the `Battle Actor` to update the in-memory `Battle Model` (and persists the state periodically to database). The `Battle Actor` sends the state of the `Battle Model` to all relevant clients, ensuring consistency across all players. The client visualises the game state based on the local copy of the `Battle Model`. This process continues until the battle session concludes.

5. **End of Battle:** The battle concludes, and the `Battle Actor` processes the final result of the `Battle Model`. The `Battle Actor` sends the results to the `Session Actor` of each player. The `Session Actor` updates the player's `Player Model` via the `Player Actor` with the results, persisting it to the database.

6. **Players Exit the Battle:** Each players' `Session Actor` terminates the connection to the `Battle Actor`, and the `Battle Actor` is stopped once

all players have disconnected. The clients get their updated `Player Model` state from their `Player Actor`.

Considering different scenarios such as network latency, packet loss, or client disconnections, the system is designed to handle these issues gracefully. Let's consider a few examples:

- **Network Latency:** The server's tick-based simulation ensures that all clients' actions are processed in the correct order. The server's authoritative state is the final arbiter of the game's state, ensuring consistency across all clients. Client-side prediction and server-side reconciliation can be implemented to reduce the perceived effects of latency.
- **Packet Loss:** If a client's message is lost due to packet loss, the server can push a full state re-sync, or a client-side roll back logic can be implemented to allow replaying the missing ticks and actions.
- **Client Disconnection:** If a client disconnects during a battle session, the `Session Actor` will handle the reconnection and resynchronise the `Battle Model` to the client. If the battle session was concluded while the client was disconnected for a long time, the persisted `Battle Model` can be loaded into a `Battle Actor`, and the results can be sent to the client's `Session Actor` for updating the `Player Model`.

5. Conclusion

This thesis investigated the synchronous execution paradigm for modern mobile game development, a concept that has emerged in multiple forms within the Finnish mobile game industry over the past couple of decades. By enabling distributed yet synchronised execution of game logic across both client and server, this paradigm addresses the fundamental challenges of maintaining secure, scalable, and continuously evolving free-to-play mobile games. Throughout the preceding chapters, multiple facets of this paradigm were examined, spanning from server-authoritative logic validation to the technical infrastructure supporting Games-as-a-Service operations.

A key contribution of this work lies in formally documenting and analysing a paradigm that, despite widespread adoption in the Finnish mobile game industry, had not been comprehensively studied. The synchronous execution approach diverges from both traditional client-server multiplayer architectures and microservice-based mobile app development by offering a unified framework that bridges these two approaches. At its core, the paradigm employs server-authoritative logic to maintain consistency and fairness while allowing localised client-side execution, ensuring smooth gameplay even in environments where connectivity may be intermittent. This structure reduces the burden on network resources while enhancing scalability through the use of atomic operations that facilitate secure, structured, and efficient gameplay execution.

The paradigm's foundation in addressing the challenges of less reliable mobile networks has proven valuable even as mobile networking has improved. By not requiring constant server connection for fluid game experiences, it accommodates the complex interconnected nature of game state that makes traditional microservice architectures less suitable for video games, while providing the horizontal scaling capabilities that dedicated game servers struggle to achieve at mobile platform scale.

Several core principles define this synchronous execution paradigm. **Unified Logic Layer** provides a monolithic but modular approach with shared codebase execution on both client and server, reducing version fragmentation and operational overhead compared to managing numerous disconnected microservices for core gameplay features. The one-code approach, where identical logic code is authored once and executed on both ends,

streamlines development and maintenance while ensuring consistency. **Tick-Based Simulation** employs fixed-length update frames that emulate frame-by-frame execution, ensuring consistent gaming experiences while decoupling logic ticks from GPU-rendered update frames. This approach allows for greater server scalability by reducing computational demands while maintaining deterministic progression. **Command Pattern Implementation** encapsulates gameplay actions as discrete, self-contained command objects, facilitating validation, replay, and rollback capabilities essential for state synchronisation and robust error handling. This pattern enables atomic operations that treat each action as an indivisible unit, ensuring consistent game state management. **Deterministic Data Management** employs fixed-point arithmetic and deterministic collection types to ensure identical outcomes across client and server executions, eliminating the non-deterministic behaviour inherent in floating-point arithmetic. **LiveOps Integration** supports continuous content delivery, over-the-air configuration changes, and comprehensive player behaviour tracking through the paradigm's server-centric approach to game state management. This enables dynamic content updates, A/B testing, and real-time economic adjustments essential for free-to-play business models. **Cheat Prevention Architecture** leverages server authority and checksum verification to detect discrepancies between client and server outcomes, maintaining fair competition and economic integrity through what closely resembles a client optimistic server authoritative model.

The paradigm's implementation by Metaplay demonstrates its practical viability in real-world game development. By utilising technologies such as Akka.NET actors for message handling and in-memory state management, the paradigm achieves the concurrency, fault tolerance, and scalability required for modern mobile games. The integration of C# attributes for serialisation and configuration management, combined with efficient data formats inspired by Protocol Buffers, provides developers with powerful tools for implementing complex game features while maintaining code clarity and maintainability.

The paradigm's ability to thread the needle between traditional game server approaches and microservice architectures makes it particularly valuable for mobile game developers facing the demands of creating games that must remain relevant for years to decades. By embracing this integrated approach, studios can more effectively maintain the complex, interconnected game states required for sophisticated free-to-play experiences

while achieving the scalability necessary for mobile platform success. The framework supports diverse multiplayer experiences, from asynchronous and turn-based interactions to real-time gameplay, while also enabling compelling single-player experiences enhanced by online connectivity.

5.1 Future Work

Several avenues for future work emerge from this initial documentation and analysis of the synchronous execution paradigm.

Technical Implementation Analysis: while this thesis focused on articulating the paradigm's conceptual framework and industry adoption, future research could provide detailed technical evaluations of specific implementation approaches. Comparative performance studies of different synchronisation techniques under various network conditions, in-depth analysis of tick-based simulation scalability patterns under diverse load conditions, and examination of atomic operation optimisation strategies would complement the foundational work presented here. Such studies could explore the trade-offs between tick frequencies and server resource utilisation, investigate the effectiveness of different checksum verification approaches, and analyse the performance implications of various deterministic data structure implementations. These technical deep dives would offer developers concrete guidance for optimising their implementations while maintaining the paradigm's core benefits.

Historical Industry Context Research: Although this thesis documented the paradigm's emergence within the Finnish mobile game industry, a more comprehensive historical analysis would provide valuable institutional context. An interview-based study with founders and technical leads from pioneering game companies could illuminate the decision-making processes, evolutionary pressures, and technical innovations that shaped the adoption of synchronous execution approaches. Such research would capture important institutional knowledge about how technical paradigms emerge and spread within specific industry contexts, particularly during the era of less reliable mobile networks that shaped the paradigm's initial development. This historical perspective could reveal patterns of technology adoption that might inform future paradigm evolution and help identify the conditions that favour such unified approaches over traditional microservice architectures.

Educational Integration and Professional Preparation: Given the

widespread adoption of synchronous execution paradigms in the Finnish mobile game industry, there is significant value in systematically introducing these concepts to computer science and game development students. Future work could develop comprehensive curriculum modules that bridge the gap between academic game development education and industry practices. This might include practical exercises implementing tick-based simulations, hands-on experience with command pattern applications in game contexts, and case studies examining real-world implementations of server-authoritative logic. Such educational initiatives would ensure that new graduates entering the Finnish mobile game industry understand the paradigm's principles and can contribute effectively to its continued evolution, while also preparing them for the unique challenges of games-as-a-service development and LiveOps management.

In summary, this thesis has provided a comprehensive framework for understanding the synchronous execution paradigm and its utility in modern mobile game development. By formally documenting this approach to game logic execution, state management, and live operations, the work addresses many of the challenges facing contemporary mobile game developers. The paradigm facilitates the creation of games that are robust, fair, and adaptable, offering solutions for maintaining game integrity, supporting complex monetisation strategies, and driving player engagement in an increasingly competitive market. The unified logic layer approach, combined with tick-based simulations, atomic operations, and deterministic data management, provides a methodical foundation for developing scalable games that can adapt to the ever-changing landscape of mobile gaming. while the concepts and principles discussed are broadly applicable across different tools and technologies, as evidenced by the multiple forms this paradigm has taken in the Finnish mobile game industry, the framework presented here offers a solid foundation for future mobile game development. The suggested research directions provide clear paths for expanding both theoretical understanding and practical application of these approaches, ensuring the paradigm's continued relevance in the evolving landscape of online game development.

References

[1] *Mobile Gaming Market - Report, Trends & Growth*. URL: https://www.mordorintelligence.com/industry-reports/mobile-games-market (visited on 09/04/2024).

[2] Debadatta Mishra et al. "Clouds + Games: A Multifaceted Approach". In: *IEEE Internet Computing* 18.3 (May 2014), pp. 20–27. ISSN: 1089-7801. DOI: 10.1109/MIC.2014.20.

[3] *Source Multiplayer Networking - Valve Developer Community*. URL: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking (visited on 03/27/2025).

[4] *About Netcode for GameObjects | Unity Multiplayer*. URL: https://docs-multiplayer.unity3d.com/netcode/current/about/ (visited on 03/27/2025).

[5] *Introduction to Iris in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community*. Epic Games Developer. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/introduction-to-iris-in-unreal-engine (visited on 03/27/2025).

[6] *High-Level Multiplayer - Godot 4.4*. Godot Engine documentation. URL: https://docs.godotengine.org/en/4.4/tutorials/networking/high_level_multiplayer.html (visited on 03/27/2025).

[7] Will Luton. *Free2Play: Making Money from Games You Give Away*. New Riders, 2013. 217 pp. ISBN: 978-0-321-91901-4.

[8] LLC Treeform's Games. *Don't Use Lockstep in RTS Games*. Oct. 1, 2016. URL: https://blog.istrolid.com/blog/dont-use-lockstep-in-rts-games.html (visited on 03/27/2025).

[9] *Pyramid: PBM: The Infancy of Cyberspace*. URL: https://www.sjgames.com/pyramid/sample.html?id=363 (visited on 03/27/2025).

[10] Robert Nystrom. *Command - Design Patterns Revisited - Game Programming Patterns*. URL: https://gameprogrammingpatterns.com/command.html (visited on 02/09/2023).

[11] Louis-Etienne Dubois and Johanna Weststar. "Games-as-a-Service: Conflicted Identities on the New Front-Line of Video Game Development". In: *New Media & Society* 24.10 (Oct. 2022), pp. 2332–2353. ISSN: 1461-4448, 1461-7315. DOI: 10.1177/1461444821995815.

[12] Oskari Teittinen. “Analysis of Cheat Detection and Prevention Techniques in Mobile Games”. June 18, 2018. URL: https://urn.fi/URN:NBN:fi:aalto-201806293894 (visited on 02/09/2023).

[13] Jussi Prokkola. “LiveOps Product Management for Mobile Games”. 2022. URL: https://urn.fi/URN:NBN:fi:amk-2022061017416.

[14] *What Is Feature Flag-Driven Development?* URL: https://www.geteppo.com/blog/feature-flag-driven-development,%20https://www.geteppo.com/blog/feature-flag-driven-development (visited on 05/10/2025).

[15] Scott Hunter [MSFT. *.NET Core Is the Future of .NET*. .NET Blog. May 6, 2019. URL: https://devblogs.microsoft.com/dotnet/net-core-is-the-future-of-net/ (visited on 06/09/2025).

[16] Rich Lander [MSFT. *Announcing .NET 5.0*. .NET Blog. Nov. 10, 2020. URL: https://devblogs.microsoft.com/dotnet/announcing-net-5-0/ (visited on 06/09/2025).

[17] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Reading, Mass.: Addison-Wesley, 1995. ISBN: 978-0-201-63361-0.

[18] Glenn Fiedler. *Fix Your Timestep!* June 2004. URL: https://gafferongames.com/post/fix_your_timestep/ (visited on 04/04/2023).

[19] Mark Claypool and Kajal Claypool. “Latency and Player Actions in Online Games”. In: *Communications of the ACM* 49.11 (Nov. 2006), pp. 40–45. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1167838.1167860. URL: https://dl.acm.org/doi/10.1145/1167838.1167860 (visited on 06/09/2025).

[20] BillWagner. *Attributes and Reflection - C#*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/ (visited on 06/09/2025).

[21] Rick-Anderson. *Create Data Transfer Objects (DTOs)*. URL: https://learn.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5 (visited on 06/09/2025).

[22] Jere Sanisalo. *FixPointCS*. URL: https://github.com/XMunkki/FixPointCS (visited on 03/30/2025).

[23] *Deterministic Collections | Metaplay Docs*. URL: https://docs.metaplay.io/game-logic/utilities/deterministic-collections.html (visited on 06/14/2025).

[24] Cosmin Arad, Jim Dowling, and Seif Haridi. “Message-Passing Concurrency for Scalable, Stateful, Reconfigurable Middleware”. In: *Middleware 2012*. Ed. by Priya Narasimhan and Peter Triantafillou. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 208–228. ISBN: 978-3-642-35170-9. DOI: 10.1007/978-3-642-35170-9_11.

[25] *Actor Systems | Akka.NET Documentation*. URL: https://getakka.net/articles/concepts/actor-systems.html (visited on 03/28/2023).