Vrije Universiteit Amsterdam

Bachelor Thesis

Design and Evaluation of a Novel Virtual-World Architecture: Separating Simulator and State

Author: Shane C. Prent (2755970)

1st supervisor: ir. Jesse Donkervliet
daily supervisor: ir. Jesse Donkervliet
2nd reader: Prof. Dr. ir. Alexandru Iosup

A thesis submitted in fulfilment of the requirements for the VU Bachelor of Science degree in Computer Science

August 24, 2025

Abstract

Modifiable Virtual Environments (MVEs) such as Minecraft and Roblox are known for their immersive and continuous worlds, but they tend to hit limitations when attempting to scale beyond a few hundred players concurrently with traditional server-centric designs. This thesis explores the potential of using an in-memory key-value store database, specifically Redis, to completely replace conventional game servers.

We design and implement KeyVerse, a Unity-based prototype that maps player data, terrain blocks, and event streams onto Redis hashes, sets, and pub/sub channels. Each client operation is constrained by a dynamically moving 3 x 3 proximity window to manage message distribution effectively. Additional services like simulation, anti-cheat mechanisms, and analytics can be integrated through micro-services without altering the core system. The prototype supports gameplay for hundreds of concurrent users on a single Redis node, while maintaining update latencies under 100 milliseconds.

Our findings indicate that a key-value store can serve as the authoritative backbone for virtual environments, streamlining deployment and paving the way for serverless, horizontally scalable worlds. In testing, KeyVerse supports 1 200 concurrent players using a dynamic pub/sub system and 300 concurrent players on a single channel pub/sub system. We explore consistency trade-offs within the CALM framework and propose future research on sharded pub/sub mechanisms to expand the approach beyond a solitary broker.

Contents

1

Introduction

The gaming industry has undergone a dramatic transformation over the past decade or two, moving from single-player offline experiences to large, interconnected, online multiplayer worlds. Games like *Minecraft*, *Roblox*, and even *Fortnite* have pioneered the modern concept of Modifiable Virtual Environments (MVEs). MVEs are dynamic worlds where players can interact, build, and explore together. They are platforms for creativity, education, and social interaction, reflecting a broader societal shift toward digital and online spaces.

As MVEs continue to grow in popularity, they face a critical challenge: scaling to support a large number of concurrent players while maintaining low latency, consistency, and a seamless user experience. Traditional game server architectures, which rely on centralised servers to manage game state and player interactions, often struggle to meet these demands. These servers are responsible for critical tasks such as maintaining the game loop, enforcing security, and ensuring data consistency. However, the centralisation of these systems can lead to bottlenecks, especially as the player count increases.

This thesis explores an alternative approach: replacing traditional game servers with **key-value store (KVS) databases**. KVS databases, such as *Redis* or *Amazon DynamoDB*, are known for their simplicity, high scalability, and high performance, as well as their ability to handle large amounts of data efficiently. By leveraging these strengths of KVS databases, this project aims to create an architecture for MVEs that can handle large-scale player interactions without the limitations of traditional server-based systems. However, this approach is not without its challenges, as KVS databases lack native support for many game-specific functionalities like the game loop and security mechanisms mentioned previously. This thesis investigates the feasibility of this approach and proposes solutions to address these challenges.

1.1 Problem Statement

The traditional server-client game architecture for Modifiable Virtual Environments (MVEs) relies on centralised game servers to handle both simulation logic and state management. This approach simplifies the design but introduces scalability issues. As player numbers increase and world complexity increases, these servers are tasked with processing all simulation steps and state updates sequentially within a fixed time limit (tick). This architecture **(P1)** struggles to efficiently handle high workloads driven by the environment, such as terrain updates, activity of non-playable characters, and global effects. In these cases, the simulation load grows faster than the number of players. This leads to increased latency, slower tick rates, and ultimately to a diminished user experience at large scales.

On the other hand, Key-value store (KVS) databases exhibit a different performance dynamic. These databases excel in providing high-throughput, low-latency access to large volumes of data through parallel input/output and distributed data partitioning. Unlike traditional game servers, which perform all operations in a single-threaded loop, KVS databases can accommodate multiple independent read and write operations concurrently. This capability allows various clients and services to interact with shared world data without a centralised scheduling bottleneck. However, KVS databases **(P2)** lack inherent support for game-server specific functions such as a authoritative conflict resolution, ordered event processing, and cheat prevention, making them challenging to integrate as the foundation of an MVE backend.

1.2 Research Questions

RQ1 *How to design a system that uses a key-value store database to replace traditional game servers in MVEs?*

A successful design could remove central processing bottlenecks and allow parallel, distributed state management.

The challenge is that as no such systems exist, at least not at the scale of large MVEs, and the design will need to address challenges such as real-time state management, data consistency and scalability. The design must also find a way to use a database, that was not designed for the application of a real-time game, to address the game-specific functionalities needed for the MVE to run smoothly. This question is important as it lays the foundation for a new architecture that could overcome the limitations, centralised bottlenecks, single-threaded update loops, and limited parallelism, that restrict traditional server-based systems today.

RQ2 *How to implement such a system?*

An implementation can demonstrate the feasibility of the architecture and serve as a foundation for performance testing, iteration, and refinement. A functional prototype makes it possible to assess the performance under pressure and validate whether a KVS-based system can maintain the responsiveness and consistency expected in large MVEs.

The challenge lies in adapting the chosen KVS database to handle game-specific functionalities, such as the game loop and communication mechanisms, which are not natively supported. This question is critical as it bridges the gap between the theoretical design and practical application. This implementation will also be used as a base to evaluate the system on.

RQ3 *How to evaluate the performance and other non-functional properties of such a system?*

A comprehensive evaluation can highlight advantages and disadvantages if the proposed architecture, demonstrating whether a system supported by a KVS can achieve or surpass the scalability and responsiveness standards associated with traditional game server setups. Such findings offer substantive data to inform decision-making and future work.

The key challenge lies in crafting experiments that accurately represent the workloads and interactions of genuine MVEs, thus ensuring the relevance and reliability of the results.

1.3 Research Methodology

To address **RQ1** and **RQ2**, we follow the *AtLarge Design Process* [1], which consists of an iterative cycle of: (i) formulating requirements, (ii) design, (iii) implementation and (iv) testing and validation. We keep doing this until our design and implementation gives a satisfactory answer for our research questions.

For the design phase, we will analyse the requirements for MVEs and create a KVS architecture by designing data models, communication protocols, and conflict resolution methods.

For the implementation phase, we will build a simple working prototype based on the design. This involves selecting a specific KVS database, developing the core components, and creating game client adapters.

For the evaluation phase (**RQ3**), we will design test scenarios, set up testing infrastructure, perform controlled experiments with different loads, and analyse the results. These tests will be measured against the defined design requirements.

1.4 Thesis Contributions

This thesis presents three concrete contributions:

1. **Design (Section 3):** The introduction of a serverless MVE architecture designed to store the entire game state, comprising world data and player activity, within a standard key-value store, thereby removing the need for a dedicated game server. The source code of

2. **Prototype (Section 4):** The development of KeyVerse, an open-source prototype that servers as a tangible demonstration of this architecture. It involves a Unity client that communicates directly with a Redis backend, which concretely validates the proposed design. KeyVerse's source code is hosted in a private Github repository [1]. Due to organisational restrictions, the repository cannot yet be made public, but collaborators (including the first supervisor) have access. A public release is planned for after the thesis submission.

3. **Empirical Evaluation (Section 5):** The study offers the first publicly available dataset of analysis, highlighting the limitations of Redis's single-threaded event loop on the scalability of MVEs. It also addresses why simply adding more I/O threads fails to resolve this issue, a conclusion that, although disappointing, provides actionable insights for future research. The testing framework can also be found on Github [2].

1.5 Plagiarism Declaration

I hereby declare that the contents of this thesis is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

[1]`https://github.com/Goose-9/opencraft-unity-kvs`

[2]`https://github.com/atlarge-research/yardstick/tree/feature/redis-keyverse`

2

Background

This section introduces the core concepts that underpin this thesis: traditional game server architectures, key-value stores, and modifiable virtual environments (MVEs). Understanding these foundations is important to understanding the challenges and opportunities of using KVS databases as replacements for game servers.

2.1 Traditional Game Server Architectures

Traditional game servers are the backbone of multiplayer games and Modifiable Virtual Environments (MVEs). The main responsibilities of a game server are: managing game state, enforcing rules (security) and ensuring a consistent experience for all players. Game server architectures can differ in their implementation, but most typically consist of: a game loop, state management, networking, and cheat prevention/detection. In the remainder of this section, we discuss each of these elements in turn, followed by a description of the main limitations that game servers face.

The game loop is the core mechanism that updates the game state at regular intervals (ticks), the server processes player inputs, simulates the new game world and sends updates to all connected clients (players). An example of a game loop can be seen in Figure 2.1 on the right hand side. The server receives the input from the client, updates the game model (State Management) and sends updates back to the client. This will happen over and over, until a certain condition is met and/or the connection is terminated.

Additionally, the server maintains a centralised game state, which includes the positions of players, objects, and other dynamic elements in the virtual world. The server does this in order to have a reference to enforce rules and to help compensate for players with high latency (e.g., prediction algorithms).

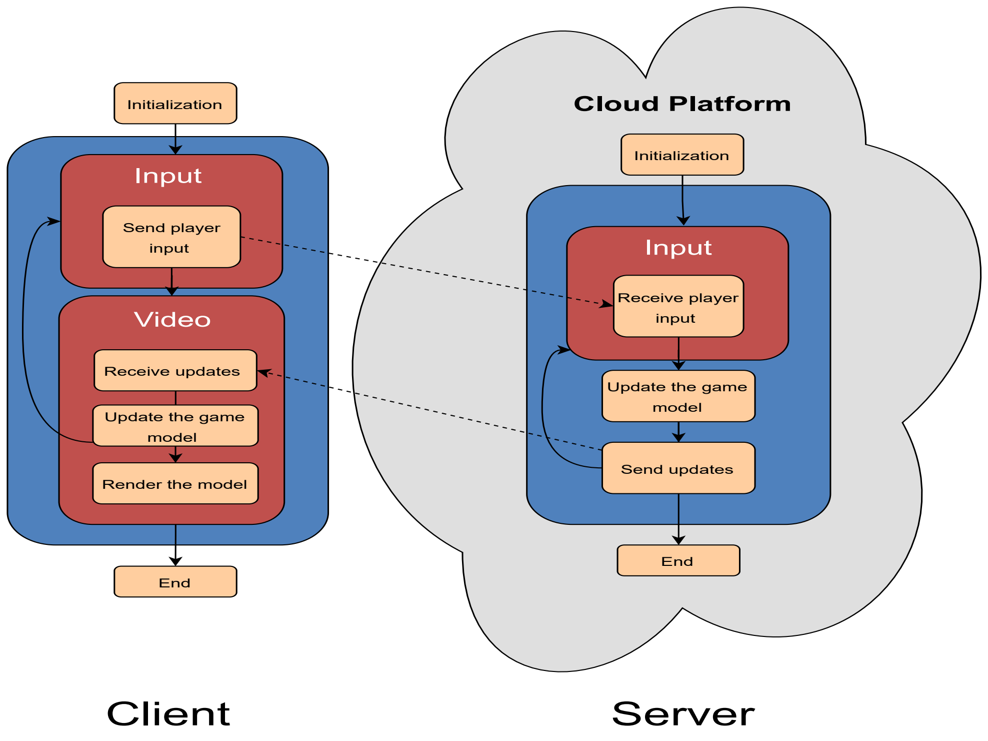

Figure 2.1: Simple Game Server in Client-Server Architecture.

The server handles communication between players by having all the clients connect to it and forwarding updates between the clients. This is essential to all clients receiving timely updates about the game state. This often involves using networking protocols like TCP and/or UDP to balance reliability and performance.

Finally, the game server enforces rules and prevents cheating by validating player actions and maintaining authority over the game state. This is especially important in competitive online games, as cheating can ruin the experience for other players.

Although game server implementations are simple and widely used, they face two main limitations: scalability and single point of failure. In terms of scalability, centralised servers become performance bottlenecks as the number of concurrent players increases, leading to increased latency and performance issues. This can occur for various reasons, such as insufficient network bandwidth preventing the server from receiving all the necessary data to complete a cycle of the game loop, or even that the server's CPU cannot keep up with the increase in player simulation updates within its time budget, causing the

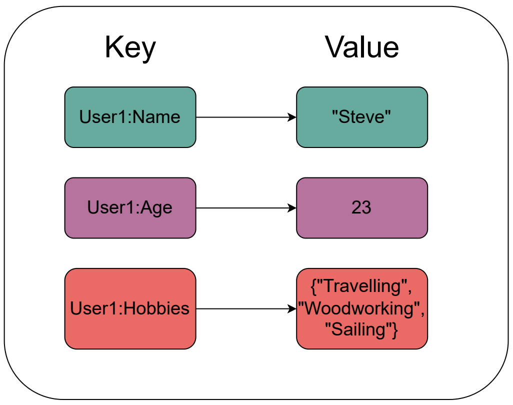

Figure 2.2: Simple Example of a Key-Value Store Database.

game to pause/freeze and stutter [2]. Additionally, the centralised nature of these servers creates a single point of failure, meaning that if the server crashes or experiences downtime, clients lose access to the game world entirely as all game logic, state updates, and player interactions depend on the server's availability.

2.2 Key-Value Store (KVS) Databases

Key-value store (KVS) databases are a type of lightweight, NoSQL database designed for high-speed data access and scalability. They store data as a collection of key-value pairs, where each key is unique and maps to a specific value. Examples of popular KVS databases include *Amazon DynamoDB* [1], *Apache Cassandra* [2] and *Redis* [3] (which will be used in this project). These systems excel in scenarios requiring fast read/write operations, making them a compelling alternative to traditional relational databases for certain use cases.

One of the primary strengths of KVS databases lies in their scalability. Unlike monolithic databases, KVS systems are inherently distributed, allowing them to handle large volumes of data and high request rates by partitioning data across multiple nodes. Additionally, KVS databases are optimised for performance, offering low-latency read and write operations. This efficiency makes them well-suited for real-time applications, where delays of even a few milliseconds can degrade user experience. Another key advantage is their flexibility in data modelling. KVS systems, such as the one seen in Figure 2.2, can

[1]`https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html`
[2]`https://cassandra.apache.org/_/index.html`
[3]`https://redis.io/docs/latest/`

store a wide range of data types, from simple strings to serialised objects or nested hashes, allowing developers to adapt the database to diverse requirements without rigid schema constraints.

However, KVS databases also present notable limitations, especially when applied to multiplayer virtual environments (MVEs). First, they lack build-in functionality for game-specific tasks, such as: managing the game loop, enforcing real-time state updates, or processing complex game logic. These responsibilities must be implemented separately, often requiring custom solutions. Second, consistency challenges can arise in distributed KVS architectures. This can lead to temporary discrepancies between nodes, which can be a critical issue for MVEs, where players expect instant and coherent updates. Finally, KVS databases do not natively support cheat prevention mechanisms, such as authoritative server validation. This absence necessitates additional security layers to detect and mitigate exploits, adding complexity to the system.

Despite these challenges, KVS databases offer a promising foundation for building scalable and efficient MVE backends. This thesis explores how their strengths can be leveraged to replace traditional game servers, while proposing solutions to address their limitations in consistency, game logic processing and security.

2.3 Modifiable Virtual Environments (MVEs)

Modifiable Virtual Environments (MVEs) are dynamic, interactive virtual worlds where players can create, modify, and explore content in real-time. Examples include games like *Minecraft*, *Roblox*, and *Fortnite*. MVEs typically use a client-server architecture, where the server simulates the changes to the world and sends the new state to the players/clients [3]. The player can interact with the world by performing various MVE-specific actions (e.g., mining blocks, crafting items, moving the avatar). The actions are simulated locally on the client and then sent to the server to be broadcast to all other clients. Unlike static game worlds, MVEs require technical support for simultaneous edits, low-latency synchronisation, and persistent state storage, all while scaling to accommodate large player bases. These demands strain traditional game servers, which centralise state management and struggle with scalability. This tension motivates exploring alternative architectures, such as decentralised key-value stores, which offer scalability but lack native support for game-specific logic as mentioned in Section 2.2.

3

Architecture of a KVS-Based Backend for MVEs

This chapter presents the architecture of the proposed key-value-store-powered architecture for Modifiable Virtual Environments (MVEs). The aim is to define the system's goals, outline the key requirements it must satisfy, and describe the design choices made to meet these requirements. Initially, we identify and motivate the requirements for the system, then we explain the overall architecture and its components. In addition, we explore specific architectural components, such as game simulators, conflict management, and scalability. These sections together provide a comprehensive perspective on how the architecture addresses the research requirements, particularly in relation to scalability, responsiveness, and consistency.

3.1 System Requirements

The system must meet the following requirements to function as a practical replacement for a traditional game server. Each of which introduces specific design challenges that are addressed in later sections of this chapter.

Requirements:

R1 Player Presence Management: Players must dynamically join/leave the shared environment, with their state recorded in the shared state store and changes broadcast to all clients. *Challenge:* Consistency in player state across clients must be maintained during joins and leaves, and presence notifications should be sent to all active clients.

R2 Real-Time State Synchronisation: Clients must exchange state changes (e.g., movements, object interactions) at high frequencies. These exchanges should happen more than 8.6 times per second (Hz), as this would coincide with the maximum playable latency of 116ms as described in [4, 5]. *Challenge:* Synchronising frequent updates across clients without a central authority risks inconsistencies (e.g., two players seeing different positions for the same object).

R3 Conflict Resolution: The architecture must resolve conflicts when clients concurrently modify the same game state (e.g., two players editing the same block or environment). *Challenge:* Without an authoritative server to arbitrate, conflicts must be detected and resolved deterministically, often requiring time-stamped writes and/or atomic operations.

R4 Low Latency: Mark Claypool [6] measures the accepted tolerable latency of: competitive first-person shooter games to be about 120-150ms, and Minecraft-like games to be up to 1s. If we compare the observation of 116ms in [7], the system must propagate quickly enough to avoid latencies up to about 120ms. *Challenge:* Key-value stores introduce latency trade-offs between consistency (strong guarantees) and responsiveness (weak guarantees). The system will need to find a balance between the two.

R5 Eventual Consistency: All clients must eventually converge to the same world state, even if temporary inconsistencies occur due to network delays or concurrent updates. *Challenge:* Divergence in views is limited to stale state (due to message delay) and conflicts (due to simultaneous edits). Staleness is mitigated by frequent state updates and client-side prediction, while conflicts are resolved deterministically using methods such as last-write-wins.

R6 Scalability to more than 100 Concurrent Users: The architecture must maintain acceptable performance with at least 100 concurrent players and scale to higher numbers without degrading responsiveness below the defined latency target. *Challenge:* As the player count increases, the architecture must continue to provide timely updates and maintain consistency in the shared world state across all players.

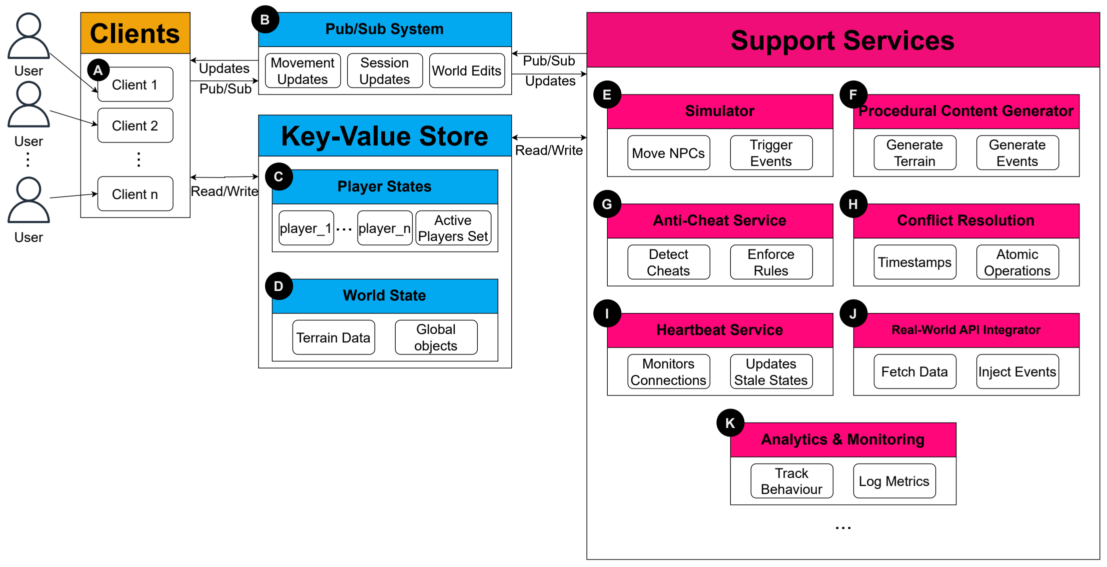

Figure 3.1: A High-level Architecture of a Key-Value Store-Powered Multiplayer System

3.2 Architecture Overview

The proposed architecture replaces the traditional game server with a distributed system that leverages an in-memory key-value store. By decentralising control and modularising responsibilities, this architecture enables the system to scale with workload rather than player count. Players and game simulator services interact directly with the shared world state in the key-value store, allowing for parallel updates, reduced contention, and flexible distribution of simulation tasks and responsibilities. The different components and how the system requirements are addressed will be shown below.

At its core, the architecture separates the system into two primary roles: the shared state and the simulators. The shared state, maintained in the key-value store, acts as the reference for the game environment, ensuring consistent data interactions and eliminating the need for a centralised processing loop. Both player clients (**A**) and game simulator services (**E**–**K**), discussed further in Section 3.3, interact with and modify this state, while concurrently exchanging events via a publish/subscribe system (**B**). This division allows scalability for each component: the shared state can be partitioned or duplicated to accommodate increased data and transaction volumes, while simulators can be distributed or adapted to address increased computational demands, such as updates for non-player characters or procedural content generation. By dividing state and simulation functions, the architecture flexibly adjusts to loads generated by either player activity or environmental changes without the need to overhaul its fundamental structure.

Each game client (**A**) maintains its own state and listens for updates from other clients via pub/sub (**B**). These updates include real-time information such as player movements, logins, logoffs, and edits to the virtual world, addressing the need for real-time state synchronisation (**R2**) and player presence (**R1**).

The shared game state is modelled as two primary components: one for per-player data **C** (e.g., position, orientation, and status under `player_{id}`), and one for world state **D** (e.g., terrain and global objects stored as `world:column_x_y`). This division allows access to spatially localised data, improving scalability and reducing contention (**R5**,**R6**).

Communication between clients is event-driven through pub/sub channels (**B**), such as `world:session` for login/logoff events and `world:movement` for movement updates. While the system uses timestamps and logical clocks to detect and resolve concurrent updates (**R3**) (as seen in component **H** of Figure 3.1), this approach is not without its limitations. In distributed environments, client clocks may drift, or system latencies may cause messages to arrive out of order. These issues can lead to incorrect conflict resolutions if timestamps are treated as globally accurate. As such, this architecture assumes loosely synchronised clocks and applies conflict resolution ideas with this limitation in mind. This trade-off and its implications are further discussed in Section 3.4.

To keep track of active users and session health, the system includes a pseudo-client called the heartbeat service (**I**). This service periodically pings connected clients and removes stale entries from the active player set. It may also optionally cache the latest state of disconnected players and help synchronise this state upon reconnect. This contributes to fault tolerance (**R1**).

To demonstrate the extensibility of the architecture, we consider several support or simulator services that can be integrated without altering the client codebase, such as simulators, procedural content generators, or analytics engines. These services can extend or enhance gameplay experience without requiring client modifications or server downtime. These capabilities are discussed in more detail in Section 3.3.

This decentralised model aims to preserve responsiveness while ensuring eventual consistency (**R5**, **R4**). Key value stores, particularly those in memory, offer low-latency read and write operations, which are essential for high-frequency state changes such as player movements (**R2**). However, challenges arise in ensuring that all clients maintain a coherent and agreed-upon world view despite potential network latencies or concurrent modifications (**R3**).

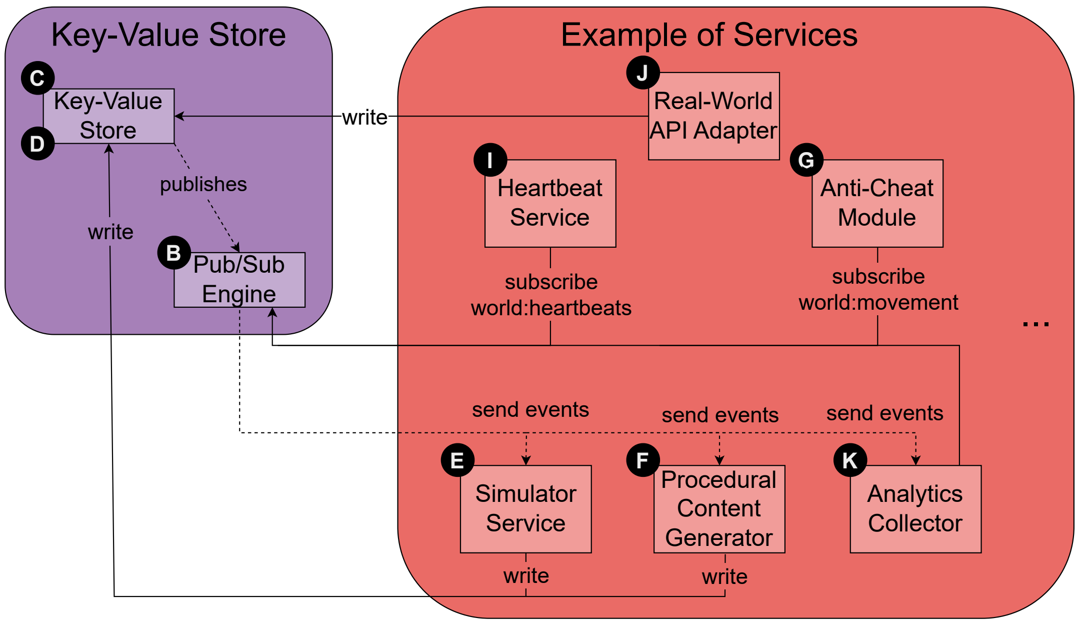

Figure 3.2: Support/Simulator Service Interaction Diagram. The labels match that of Figure 3.1

3.3 Game Simulators

One of the most compelling features of this architecture is its support for optional modular services that enhance the functionality of the game world without requiring changes to the clients or restarting the system. This section explores this idea and proposes several interesting use cases for these services. This section will reference Figure 3.1.

These support services or simulators are stand-alone programs or background processes that interact with the key-value store (**C, D**) and pub/sub (**B**) messaging system to enhance consistency, interactivity, and responsiveness in the virtual world.

One branch of services is the Simulators (**E**), which can drive in-game events or simulate non-player characters (NPCs). These services act as independent processes that update entity states, such as NPC positions, animations, or terrain conditions, directly in the data store. Clients can then react to these updates in real time, ensuring a consistent world view across all participants. This removes the burden, simulating global behaviour, from individual clients and allows complex logic to be managed centrally, improving trust and coherence in the decentralised system.

Procedural Content Generation (**F** services form another valuable category. These services can create or modify world data, items, quests, or events without interrupting player sessions. For example, new terrain or interactive objects can be generated when a player moves into unexplored area. Likewise, procedural systems can add narrative content, generate seasonal events, or populate the world with dynamically crafted challenges. Because all clients draw their view of the world from the same shared store, this content becomes immediately available to players without requiring updates or manual restarts.

Another powerful support service is an anti-cheat system (**G**) or some sort of enforcement system. Traditionally, cheat detection relies on server-side logic, but in this architecture, an anti-cheat service can run independently and monitor for suspicious updates or patterns in the shared state. For example, it can subscribe to movement or action events and flag impossible behaviour, such as teleporting across the map, acting during cooldowns, or moving faster than is allowed. Since all client actions are ultimately written to the shared store and visible via pub/sub, the service can analyse this pub/sub stream, recording violations or even triggering corrective actions such as rolling back a change, kicking a user, or broadcasting a warning. Since this logic is separate from the game clients, the service can continue to evolve independently of the client or game logic, and allows for deeper analysis without affecting gameplay latency. It also makes it easier to experiment with advanced techniques as the game matures.

This architecture also opens up possibilities for integrating real world data (**J**) into the virtual environment. Support services can fetch and reflect external information, such as live weather conditions, financial markets, or current events, and add the corresponding changes into the world state. This creates opportunities for hybrid experiences, where in-game content is shaped by live data feeds, offering not only immersion, but also creative and educational applications.

Analytics and monitoring services (**K**) can be introduced to observe client behaviour and system health. By subscribing to relevant event channels, a service can log movement trends, measure activity levels, or track user engagement patterns. These insights can inform design improvements or drive adaptive systems, such as dynamic difficulty scaling. Similarly, moderation services can listen for behavioural patterns, such as abusive language, spamming, or abnormal interaction rates, and respond automatically, helping to ensure a safe and enjoyable player experience. Research shows that game analytics are vital to improve player engagement, system performance, and safety [8].

The architecture's modular nature allows these services to be added, updated, or removed without affecting core functionality. Since they operate independently and rely only on standardised interfaces, key-value reads/writes and pub/sub messaging, they can be written in different programming languages, deployed separately, and scaled according to their workload. This separation of concern promotes system resilience and enables developers to experiment with new ideas without risking regressions in core game features.

In summary, support services demonstrate how this architecture enables developers to build richer, more reactive game worlds. They reduce complexity on the client side, promote consistent behaviour across users and offer a platform for live content, automation and real-world integration to be added to complex worlds. Whether used for game logic, content expansion, data analysis, or system authentication, these services turn the architecture into a flexible platform that supports ongoing evolution far beyond its initial design.

3.4 Conflict Resolution and Consistency

In decentralised architectures such as the one proposed in this thesis, consistency is one of the central challenges. With no traditional server to act as the single source of truth, clients and support services must coordinate their actions indirectly through the shared data store and messaging layer. This model opens up many possibilities for scalable and modular design but introduces the risk of conflicts between simultaneous updates or out-of-order events. For example, two players might both believe they have picked up the same unique sword, or one might “teleport” through a wall by sending an illegal update.

Another example is highlighted in Figure 3.3, where two players are both trying to place a block (Stone or Dirt) on an empty block/cell (`block:5_5`). In this example, Player A places the dirt block first, followed by Player B placing the Stone block. If we use “last-write-wins” here, it is not fair or correct as Player A still loses their dirt block while the block is replaced by stone. In these cases, it is difficult to ensure consistent states across clients (**R2**).

Ensuring consistency in such a system is not about achieving perfect agreement at all times, but about making careful decisions about where coordination is needed, how conflicts are resolves, and where inconsistency can be tolerated without harming player experience (**R3**, **R5**).

To reason about these situations, we can draw insights from distributed systems theory, particularly the CALM theorem, short for "Consistency as Logical Monotonicity," as defined in [9]. The CALM principle states that programs that produce outputs using only monotonic logic (i.e., their outputs never need to be retracted or revised as new inputs arrive) can be executed without coordination and still be consistent. This idea is particularly relevant to systems like ours, where reducing the need for coordination directly leads

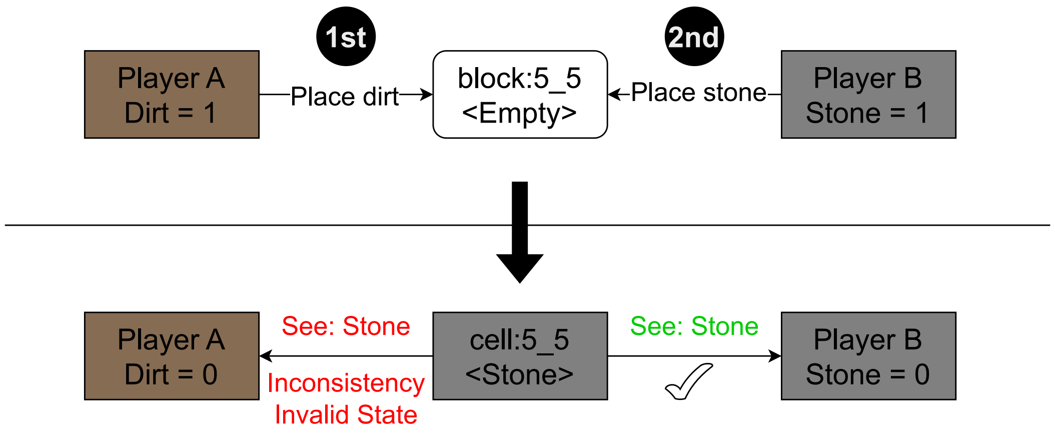

Figure 3.3: A visual example of two clients issuing conflicting "place" commands to the same block/cell, resulting in one client placing the block and the other losing theirs.

to better scalability, lower latency, and simpler design (**R4**, **R6**). Basically, the CALM theorem encourages developers to design parts of the system in a way that avoids conflicts using monotonic logic.

Applying this to our architecture, certain categories of game state can be made monotonic and therefore do not require coordination. Player logs, chat messages, and event histories are examples of data that can be appended without risk of conflict. Similarly, some types of world progression, such as marking a quest as completed or recording that a player has entered a zone, can be treated as monotonic additions. In contrast, data that is inherently mutable and exclusive, such as positions of shared objects or ownership of a tile, is non-monotonic and cannot be made consistent without some form of conflict resolution (**R3**).

For monotonic data, we simply append entries and rely on eventual consistency. For non-monotonic keys, we provide several strategies. For most data types, a simple "last-write-wins" (LWW) policy is enough, especially when updates are relatively frequent and precision is not critical. In this model, each write is tagged with a timestamp, and the system accepts the most recent one. However, this approach has well-known limitations, particularly in systems where client clocks may drift or become unsynchronised. As a result, timestamp-based ordering is only a best-effort solution and should be used with caution. For critical interactions, such as claiming ownership of an object or confirming a world change, a more robust solution involves atomic operations, such as compare-and-swap (CAS), which can ensure that only one update succeeds even under contention (**R3**, **R5**). Figure 3.4 compares LWW and CAS and shows why LWW does not always work well and why CAS is needed for critical interactions.

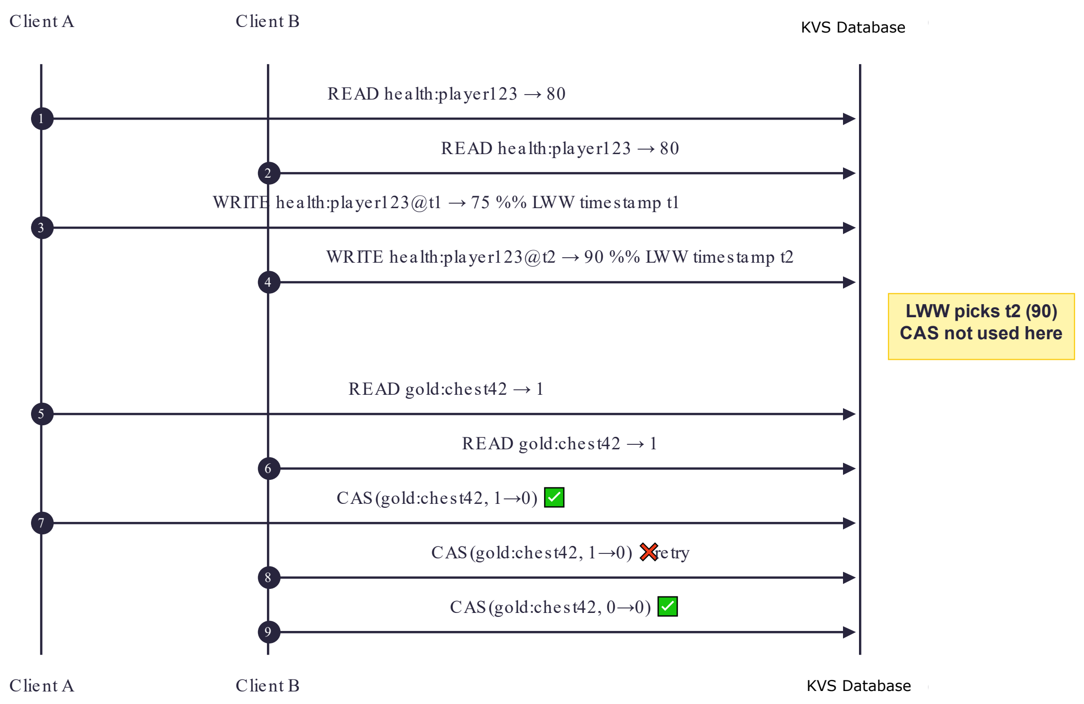

Figure 3.4: On the top, two clients race with LWW: the later timestamp wins. On the bottom, atomic CAS ensures only one client drains the last gold.

An additional method to reduce conflicts is through isolation and scoping. By assigning ownership or responsibility for certain keys to individual players or systems, we can ensure that only one process is ever expected to write to a particular subset of data. For example, each client might exclusively control its own namespace, updating its position, state, and view without risk of collision. Shared objects or world data can be partitioned spatially, where only one writer is active in a given region, or logically, with support services taking ownership of specific domains like procedural generation or simulation. These design choices make it possible to avoid unnecessary contention, rather than resolving it after the fact (**R3**).

Of course there are cases where perfect coordination is neither achievable nor necessary. In a fast-paced game, momentary inconsistencies, such as two players seeing slightly different NPC states, may be acceptable if they resolve quickly and do not disrupt gameplay. In these cases, the system favours eventual consistency (**R5**): updates may arrive in different orders across clients, but the shared state converges over time. To support this model, clients may employ interpolation and prediction techniques to ensure a smoother experience for the user. This is especially useful for data like movement, where players care more

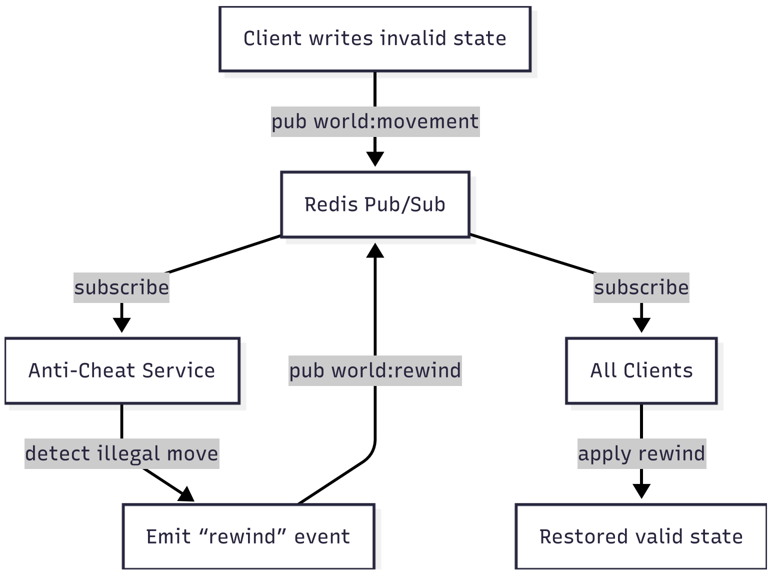

Figure 3.5: Compensation flow: invalid updates are flagged by Anti-Cheat, a "rewind" event is published, and all clients revert to a valid state.

about responsiveness than strict ordering.

So in practice, we default to LWW when the cost of inconsistency is low. For instance, if two players drop loot in the same location, the order of the drops does not affect the gameplay outcome, and thus it does not matter which player "wins" last. Conversely, in scenarios where the cost is high, such as both clients trying to consume the last "Key of Destiny", LWW would lead to incorrect state (both clients think they have the key). In that case, we wrap the write in a CAS check against the key's remaining count, ensuring only one client succeeds.

For illegal or out-of-bounds updated detected by a Anti-Cheat service, discussed in Section 3.3 and shown in Figure 3.5, we employ **compensation logic:** upon flagging an invalid state change, the service publishes a "rewind" event that each client applies to restore a valid state. This approach is commonly used in distributed systems where strict consistency would be too big of a performance cost, and it is particularly relevant to game environments where occasional conflicts can be handled through in-world logic (**R3**, **R4**).

Overall, the architecture balances the need for consistency with the realities of decentralised, low latency multiplayer systems. It uses coordination sparingly, avoids it where possible through monotonic design, and applies resolution policies where necessary. By acknowledging which parts of the system require coordination, the architecture offers a scalable and flexible platform for interactive multiplayer games. The CALM principle servers as both a theoretical guide and a practical tool for organising game logic in a coordination-free way wherever feasible.

3.5 Scalability and Performance

A core requirement is to support a large number of concurrent players while maintaining interactivity and consistency **R6**. The use of a key-value store naturally supports horizontal scalability: as the number of keys or clients grows, the store can be partitioned or sharded accordingly. Since each client mostly writes to a localised portion of the world state, contention on shared resources is minimised, allowing the system to scale without introducing bottlenecks.

Furthermore, pub/sub channels may also be segmented by topic or region to reduce congestion. For instance, the movement channels can be split up into a grid system, where clients subscribe to and unsubscribe from channels as the player moves around. The player is only interested in frequent movement updates by nearby players and thus can subscribe to a 3x3 or 5x5 grid around their current column in the world. This means that the updates from players far away, will be less frequent and thus lower the total number of messages on each channel leading to greater performance and higher throughput. This is explored more in Section 4.3.

3.6 Player Presence and Session Management

Each player joining the world is assigned a unique key prefix (e.g. `player_{id}` where id is a unique number) under which session-specific information is stored, such as current position, orientation, and online status. A global set tracks all currently active players. When a client connects, it registers itself by adding its identifier to this set and publishing a login event to the pub/sub channel (**R1**). Other clients listen to this channel and update their local state accordingly. This decentralisation of presence tracking allows the system to remain responsive and maintain player awareness even in the absence of a central server.

3.7 Real-Time Communication via Pub/Sub

To synchronise real-time events such as movement and interactions, the architecture uses pub/sub channels. Clients publish their own state changes, and simultaneously subscribe to receive events from others. This mechanism replaces traditional server-side dispatch logic, providing a direct path for client-to-client communication mediated by the database. Given the system requirement of supporting interactions at over 8.6 Hz, low-latency message propagation through pub/sub is essential (**R2**, **R4**).

However, pub/sub does not guarantee delivery or ordering. To manage inconsistencies caused by message delays or drops, each movement update includes metadata such as a timestamp or logical clock. Clients use this metadata to reconcile potentially conflicting updates and discard stale information (**R2**, **R5**).

4

Implementation

This chapter translates the design of Section 3 into a working Unity + Redis prototype. Section 4.1 summarises the overall technology stack.

4.1 Implementation Overview

In this section, we give an overview of all the frameworks and library used to implement our design presented in Section 3. Additionally, we give context and reasoning for these choices. A visual representation of the implementation stack can be seen in Figure 4.1.

The prototype, *KeyVerse*, is developed using Unity for the client engine and Redis for the backend datastore and communication broker. We chose Unity as it allows for fast prototyping of game clients and gives the benefit of using existing Redis `C#` client libraries (**B**).

The backend is built entirely on Redis, an in-memory key-value store known for its low-latency performance and flexible data structures. In this system, Redis is used not only to manage state (such as player data and world changes) but also to facilitate real-time communication between clients using its publish-subscribe (pub/sub) system. This choice addresses the following core functional and non-functional requirements:

First, the system fulfils the need for low-latency communication (**R4**) which is critical for synchronising player movement and world interactions within the sub-120ms range established in the design. The in-memory nature of Redis provides low data access (e.g., reads, writes) delays and pub/sub (**C**) provides an efficient mechanism for real-time broadcasting. Redis also allows for the execution of atomic Lua scripts (**E**) within the server, which can help with the more complex requirements of replacing a traditional game server.

Secondly, Redis plays a key role in player presence management (**R1**) and real-time state synchronisation (**R2**). The `active_players` (**D**) set provides a centralised query-

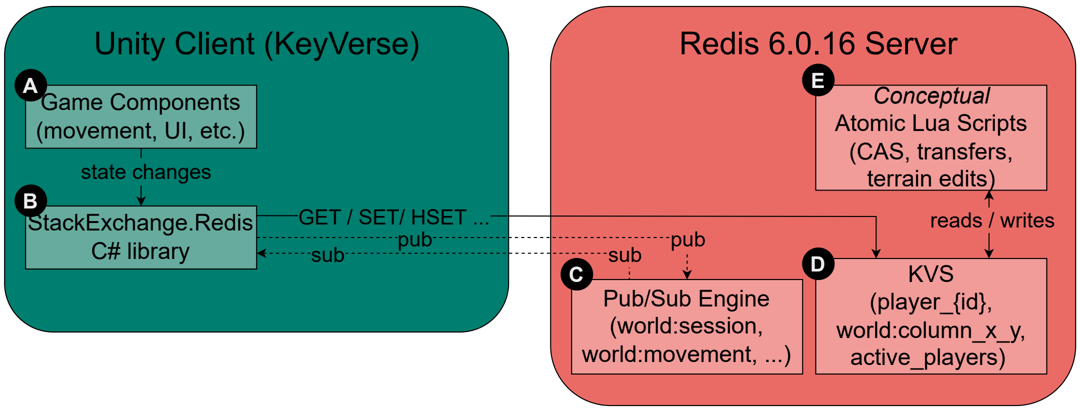
Unity Client (KeyVerse)
A
Game Components (movement, UI, etc.)
state changes
B
StackExchange.Redis C# library
GET / SET/ HSET ...
pub
sub
Redis 6.0.16 Server
E
Conceptual Atomic Lua Scripts (CAS, transfers, terrain edits)
reads / writes
pub
sub
C
Pub/Sub Engine (world:session, world:movement, ...)
D
KVS (player_{id}, world:column_x_y, active_players)

Figure 4.1: Implementation Stack Overview for the KeyVerse prototype.

able view of currently connected users, while pub/sub channels like `world:auth` and `world:movement` (**C**) ensure that join, leave and movement events are propagated to all clients. Third, these channels support scalability (**R6**) by enabling client updates without the need for polling or custom distribution methods.

Fourth, Redis's support for atomic operations (**E**) and data versioning primitives lays the groundwork for conflict resolution (**R3**), such as timestamp-based overwrites or compare-and-swap semantics.

In summary, the use of Redis addresses key architectural challenges in building a serverless multiplayer environment. Many of Redis's innate functions address and/or solve major portions of the requirements in the system. It supports scalable, low-latency communication, state persistence, and real-time coordination between distributed clients.

4.2 Implementation of the Communication Sub-system

From the client's point of view there is two types of exchanges happening between the client and Redis. Firstly there is the exchange of state data, mainly as reads and writes to the keys in the database, which is explained in Section 4.4. Secondly, explained in this section, is the exchange of game-specific commands aimed at other clients. This is essentially client-to-client communication facilitated by Redis Pub/Sub.

To satisfy the requirement for player presence (**R1**) and to handle the login/logoff of players in the virtual world, the prototype uses a dedicated `world:seesion` channel. When a player joins the environment, first their client subscribes to this channel in order to receive login commands from other new clients and logoff commands from clients leaving.

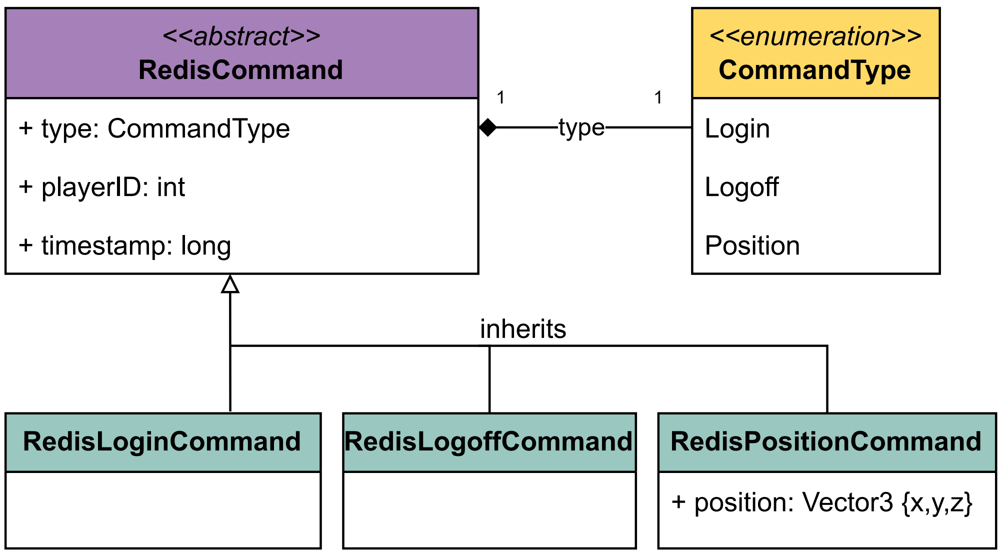

Figure 4.2: Class hierarchy of the custom Redis communication protocol showing inheritance from base RedisCommand and specialised command types

Following this subscribe, the client also publishes a login message to this channel to prompt other clients to start displaying a new player. Through this mechanism, all clients are kept aware of other players entering or leaving the shared world.

Another Pub/Sub channel `world:movement` is responsible for relaying player positions updates. Clients publish to this channel whenever movement is detected locally, and they listen for movement updates from others. This pattern allows for real-time state synchronisation (**R2**), which is essential to delivering a fluid and responsive multiplayer experience. Since all updates are timestamped, it also becomes possible to implement conflict resolution mechanisms (**R3**) such as a "last-write-wins" policy by comparing update times and discarding stale data. However, as described heavily in Section 3.4, this is only a best-effort solution and should be used with caution. Since this is a simple prototype to test the viability of a very simple Redis-based game, consistency is not the main focus. To improve this, more robust solutions can be used like designing updates to be monotonic and to use atomic operations.

A single global channel, `world:movement`, is easy to reason about but can scale poorly. So we needed to find a way to broadcast high-frequency movement without flooding clients with updates (**R2**. Messages average 60 bytes, so 500 players sending messages at 10 Hz would deliver 300 KB/s to every subscriber. This can take some time for the client to work through for every frame, so we implemented a Dynamic Proximity-based Pub/sub system in Section 4.3 that preserves low latency will cutting fan out overhead.

Messages sent over Redis Pub/Sub follow a unified format based on a custom command protocol (see Figure 4.2). All command messages inherit from a base `RedisCommand` class, which includes fields such as a `type`, `playerID`, and `timestamp`. Specific subclasses like `RedisLoginCommand`, `RedisLogoffCommand` and `RedisPositionCommand` encapsulate the semantics of each message type. This abstraction allows for easy additions to the protocol and enforces a consistent structure for deserialisation and interpretation by clients. Serialisation currently uses JSON via a custom converter, allowing messages to be easily read and debugged. While this incurs some overhead, the format is flexible and may be replaced with a more compact binary format in the future.

4.3 Dynamic Pub/Sub System

A single channel for all client movement updates wastes bandwidth and processing time on clients receiving the updates. Players only really need to receive updates from other players that they may have contact with or see in the game, such as players that are nearby.

The dynamic publish/subscribe system replaces the single movement channel with a proximity-aware scheme that scales better as the player count increases (**R6**). Instead of sending every movement update to every client, the world, as described in Section 4.4, is partitioned into columns, and each column has its own movement channel (e.g., `world:movement:3_5`). Figure 4.3 visualises this system. A client publishes movement updates only to the channel that matches its current column in the world grid (solid filled blocks with player number inside in Figure 4.3) and subscribes to the channels inside a set radius, typically a 3x3 or 5x5 grid centred on its position, so it receives only the updates that happen close to it.

This subscribe radius is shown by the faded squares around the filled square in Figure 4.3. An important note is that even though Player 1 (red) and Player 2 (blue) have overlapping subscription areas (cell `2_3`), they do not receive messages from each another. This is due to the fact that their respective publishing squares fall outside the subscription areas of the other player. However, Player 3 (yellow) and Player 1 (red) are visible to each another and will send movement updates to each other by publishing to their own channels (yellow filled square and red filled square) and this update will be picked up by the other player's subscription area.

Blindly resubscribing on every frame would generate a large overhead and stall the client. Instead, the system computes the new channel set only when a client crosses a

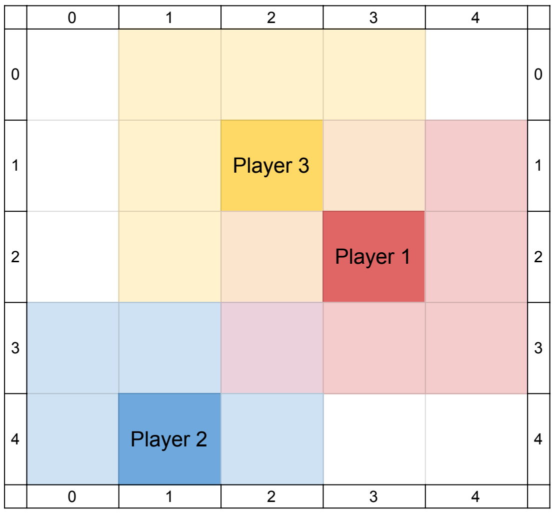

Figure 4.3: A Dynamic, 3x3 Proximity-based Publish/Subscribe Layout.

column boundary. The system then calculates the $2 * RADIUS + 1$ new channels and subscribes to them. Then it unsubscribes from the same number of channels to keep the total of subscribed channels at $RADIUS^2$. This keeps the subscription overhead low even at high tick rates.

The DynamicPubSub component orchestrates the entire workflow: it first computes the set of channels that fall within the player's subscription radius, the issues subscribe and unsubscribe commands whenever the player crosses column boundaries. It also immediately forwards any movement updates it receives to the client's handler, and finally republishes the client's own movement to the channel that matches its current column.

By lowering the subscribers for each message sent, this implementation reduces the traffic on a single channel and help to alleviate the handlers on clients from processing too many messages. The modest overhead of managing subscriptions is outweighed by the drop in deliveries on a single client, especially when hundreds of clients share the same world.

4.4 Data Model and Consistency

The data model supports a minimal but functional multiplayer environment, focusing on fast positional updates. Each Unity client renders the game world and remote players based on shared data stored in Redis. No single server holds authority.

Each player is assigned a unique identifier, generated by an atomic Redis counter, avoiding conflicts during simultaneous logins. Once assigned, a player's state is stored under a namespaced key (`player_{id}`), which holds relevant information. For now it contains the player's current position (x,y,z), but the structure can grow to include additional game-specific metadata like orientation or status effects. This structure makes it easy for clients to retrieve or update a single player's data (**R1**), without much contention, and allows for the easy addition of new attributes.

The world itself is represented as a collection of "columns", where each column corresponds to a specific grid coordinate. These are stored under the `world:` namespace using the pattern `column_x_y` where x and y are integer coordinates. The value associated with each key is a base64-encoded string representing the chunks serialised data. In this implementation, the chunk data is only required when first rendering the world, as terrain editing is outside the prototype's scope.

To support login logic and global player tracking, the system uses a Redis set named `active_players`. When a player logs in, their ID is added to this set and removed upon logout. Clients can query this set when joining to determine which other players are already online and render each player according to their position stored in their namespaced key (**R1**).

State changes, such as movement or logout, are reflected both through Pub/Sub messages as described in Section 4.2, and through updates to the Redis state store (to allow late joiners to recover state).

Because multiple clients can write overlapping keys, the prototype falls back on a simple **last-write-wins (LWW)** rule based on client-side timestamps. Every message carries the local clock value at send-time and when a recipient processes an update, it overwrites its cached value only if the incoming timestamp is later than the one it already holds. This keeps the implementation lightweight, with no sequence numbers or coordination rounds, yet still lets all the sessions converge over time (satisfying **R3** and **R5** in a basic form). This approach is trivial to implement, though it is susceptible to clock skew and described in Section 3.4.

As this prototype only focuses on movement updates, operations that truly need exclusive ownership, such as chest looting or terrain edits, are not fully protected in the prototype. Adding atomic operations via commands like `SETNX` or Lua compare-and-swap scripts is therefore identified as future work (see Section 4.6) and would be required for production-grade integrity.

Although Redis is an in-memory store, it can be configured to persist data to disk periodically. However, in the current implementation, this is not critical, as the system does not yet persist long-term state between sessions.

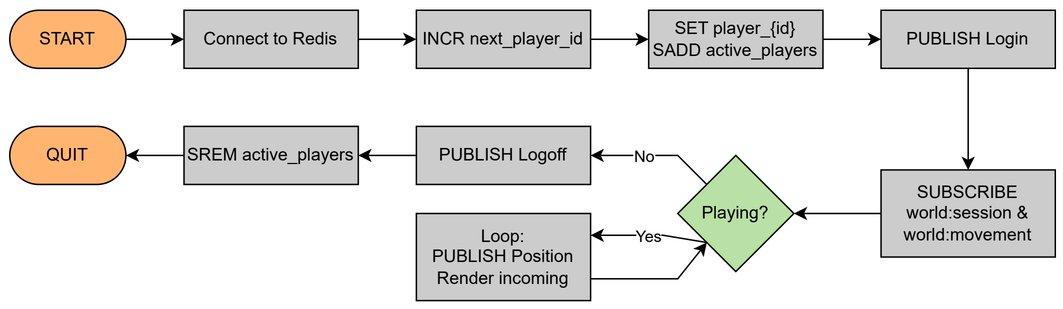

Figure 4.4: Client Lifecycle in KeyVerse.

4.5 Client Lifecycle

The client lifecycle explains how a player connects to, interacts with, and eventually disconnects from the shared game environment. This flow is a critical part of the system's architecture, as it governs how state is initialised, synchronised, and maintained across clients without a central game server. Figure 4.4 shows a visual representation of the client lifecycle.

The first challenge is to guarantee unique identifiers for the players and to keep consistent presence tracking (**R1**). Two players logging in at the same time can cause a race condition on ID assignment, which would give two players the same identifier. To combat this, we store an increment in the database that can be increased atomically using `INCR next_player_id`. This will give each player a unique ID without coordination (**R5**). This identifier is used to initialize the player's state and to namespace their corresponding Redis keys (e.g., `player_29`). The client can also use this identifier to rejoin the virtual world after disconnecting.

The client then inserts itself into the `active_players` set and publishes a `LoginCommand` as shown in 4.2 to the `world:session` Pub/Sub channel. This message informs all other subscribed clients that a new player has joined the session. At the same time, the joining client also subscribes to this channel to listen for future login or logoff event from other players.

To render the current multiplayer state, the new client queries the `active_players` set in Redis. This provides a complete list of player IDs currently online. For each entry, the client reads the corresponding `player_{id}` key to retrieve and render that player's position.

When a player moves, the client publishes a `PositionCommand` to the `world:movement` Pub/Sub channel. This command contains the new position data (x,y,z). Other players (subscribers) can use this message to render the player at the new position **R2**.

The final challenge in the life cycle is avoiding orphan records when players disconnect. Unexpected terminations could leave stale keys in the database. On graceful quit, the client publishes a `Logoff` to the `world:session` channel. This is picked up by all the other subscribed clients who remove the player from their rendering list. Following this, the client removes its ID from `active_players` in one atomic step. Additional support or maintenance services, such as a Heartbeat Service explained in Section 3.1 can come in to clean up stale keys.

4.6 Future Work

The current prototype is deliberately lightweight, focusing on core networking and state-sharing to verify whether a system like this would be plausible at the basic level first. Several additions or extensions would strengthen correctness, enrich gameplay, and move the system toward a more production-ready version.

Redis, the key-value store used in this prototype, embeds a lightweight Lua interpreter. A client can send a Lua script which the server executes atomically: it may read multiple keys, perform conditional logic, and write results, all in one uninterrupted operation. Every other client sees the scripts changes as if they happened at the same time, so race conditions disappear without the cost of multi-round coordination.

Specifically, these Lua scripts can be used for stronger consistency. With Lua available, atomic terrain edits, inventory transfers, or chest claims can be guarded by short scripts. For example., a compare-and-swap script would read a block's owner field, verify it is empty, and set the new owner in a single round trip. Building a small library of Lua scripts would close the gap between this prototype's last-write-wins and production-grade consistency, directly strengthening **R3** and **R5**.

The design already mentions optional micro-services that plug into the same pub/sub and key-value layer. Three high-impact candidates are:

- Heartbeat service - pings clients every few seconds and removes stale IDs from `active_players`
- NPC simulator - drives non-player characters by writing position keys, providing authoritative AI movement.
- Anti-cheat analyser - subscribes to movement channels, detects velocity spikes and flags suspicious behaviour.

As each runs independently, they can be deployed, scaled, or updated without touching client code.

To take the prototype past movement updates, more interesting mechanics can be implemented to test more parts of the Redis backend. Current column keys only store static terrain. Future versions could include crafting stations, interactive blocks, or dynamic weather, each protected by Lua scripts for safe concurrent edits.

Collectively, these extensions could elevate the prototype into a fully featured platform for large-scale, modifiable virtual worlds.

5

Evaluation

This section evaluates the performance characteristics of the Redis-based multiplayer architecture prototype described in Chapter 4. The goal is to empirically verify whether the system can handle large-scale client communication under varying loads, and to identify potential bottlenecks or trade-offs inherent in the design.

5.1 Main Findings

We summarise the following set of Main Findings (MF):

MF1 With the dynamic proximity-based pub/sub system from Section 4.3, the system can support ≈1 200 concurrent players under **R4** (≥8.33 Hz), with the recommended 10 Hz operating point being 1 000 players. Above this the system becomes limited by the single-threaded Redis PUBLISH loop, which caps throughput at ≈12 000 ops/s (see Section 5.3).

MF2 A single-channel, Redis 6.0.16 broker can support ≈300 concurrent players at 10 updates per second (10 Hz) on one pub/sub channel. Beyond that, the performance is limited by the single channelled pub/sub system which limits the throughput (see Section 5.4).

MF3 Expanding the view distance from 3×3 to 5×5 significantly diminishes player capacity, dropping from ≈1 200 players to ≈400 players (≈66.67% less). This drop is expected, but the experiment shows how much the scalability is reduced, highlighting the trade-off between immersion and player count (see Section 5.5).

MF4 Increasing the Redis `io-threads` configuration setting from 1 to 8 has no measurable effect on publish throughput or memory stability under high pub/sub load. While this result is expected, the experiment is necessary to rule out the socket I/O as the bottleneck and confirm that the true limit lies in the single-threaded `PUBLISH` loop. (see Section 5.6).

5.2 Experimental Setup

All experiments presented are performed on the VU cluster of the DAS-5 distributed supercomputer [10]. The DAS-5 is a project of the Advanced School for Computing and Imaging (ASCI), and is funded by NWO/NCF. Each of the 68 nodes (58 were available at the time of testing) of the VU cluster has two 8-core Intel E5-2630v3 CPU @2.4 GHz with 64 GB RAM.

The experiments are launched with **Yardstick** [11], a lightweight framework that reserves exclusive CPU cores on the DAS-5 cluster and deploys the workload as ordinary userland processes. Yardstick takes a small declarative inventory file that lets us label each slice/node as *server* or *worker*. A Yardstick-supplied Ansible play-book then

- **compiles and starts Redis 6.0.16** on the designated server node;
- **installs Python 3.9.23 + redis-py** on every worker node;
- **copies the bot workload and Telegraf collectors**.

Because the executables run directly on the host Linux installation on DAS-5, no custom VM images are required.

Telegraf [1] is used to collect all the metrics on the different nodes for analysis and plotting. Telegraf collects the metrics of the Redis server (equivalent to `INFO` command), hardware utilisation metrics on the nodes (CPU and Memory usage), as well as custom metrics measured by the workers like bot throughput.

All experiments were carried out using the implemented Redis-based backend system described in the implementation chapter, Chapter 4. The backend is powered by a locally hosted Redis server (v6.0.16), running on a single DAS-5 node through Yardstick. The purpose of the experiments was to evaluate the system's messaging throughput and scalability under increasing load.

[1] `https://github.com/influxdata/telegraf`

To generate load, we have developed a custom Python-based system, using the `redis-py`, library that simulates headless clients (bots) by directly publishing to Redis channels (`world:movement`) and measuring the number of messages sent and received, as well as the latency of different operations in the system. These bots replicate core client behaviour: they send login events and continuously publish position updates at a determined fixed rate (Hz). These simulated clients also listen for updates and record how many messages they receive. This allows us to measure the amount of messages that are lost and/or missed.

Unless otherwise noted, all experiments follow this environment and setup. Deviations or alternative setups are explicitly detailed in the relevant subsections.

5.3 Dynamic Pub/Sub: How many players can we handle? (MF1)

The goal of this experiment is to determine the maximum number of concurrent players that the dynamic proximity-based pub/sub system (see Section 4.3) can support at a 10 Hz update rate without exceeding the 120 ms (8.33 Hz) latency requirement (**R4**). In **MF2**, the single-channel pub/sub was limited to ≈300 players before hitting CPU and memory bottlenecks. The hypothesis is that lowering fan-out, by partitioning updates into channels based on in-game region data, will delay the CPU and buffer saturation limits observed in **MF2**, thus increasing the total number of concurrent players supported by the system.

We reuse the experimental setup from Section 5.4, replacing only the single global `world_movement` channel with proximity-based channels corresponding to a 3×3 grid around each player's current column. Each simulated player publishes position updates at a sending rate of 10 Hz or 10 msg/s to its own column's channel and subscribes to channels in its view radius (see Figure 4.3 in Section 4.3 for a visual representation of the system). We increase the number of players from 250 to 1 500 across multiple runs of 600 seconds each. We, again, gather Redis-specific metrics, player listener/sending metrics, and hardware utilisation metrics using Telegraf.

Looking at Figure 5.1, we can see that the performance of the system scales with an increase in player count, maintaining throughputs of ≈10 messages per second (Hz) per player for up to 1 000 players. Beyond this threshold, the total operations per second (ops/s)—seen in Figure 5.2—continued to increase (e.g., ≈10 120 at 1 100; 10 560 at 1 200; 10 920 at 1 300; 12 300 at 1 500), but the effective update rate per player began to decline below 10 Hz (≈9.2, 8.8, 8.4, and 8.2 Hz respectively). Table 5.1 summarises the same runs, re-

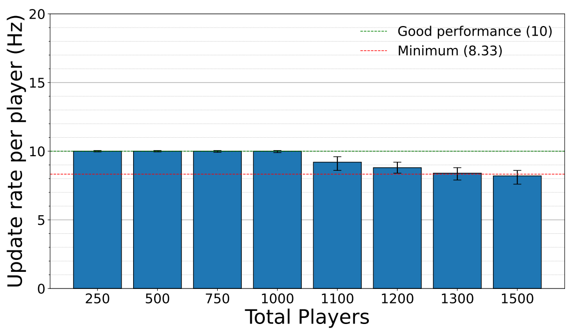

Figure 5.1: Dynamic Pub/Sub: Update Rates as the Player Count Increases.

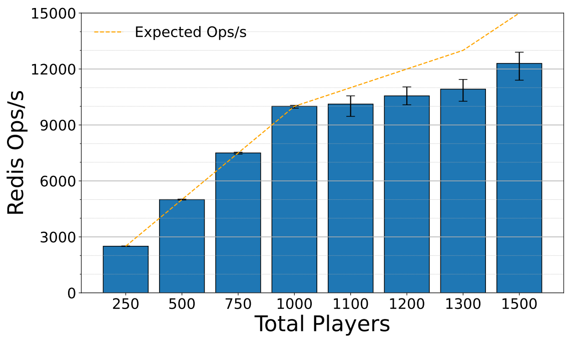

Figure 5.2: Redis Ops/s as Player Count Increases, excluding Sub/Unsub operations.

Table 5.1: Results of the dynamic multi-channelled test at a 10 Hz publish target.

Players	Target publishes/s	Redis ops/s	Measured vs target	Sending Rate (Hz)
250	2 500	2 500	±0%	10
500	5 000	5 000	±0%	10
750	7 500	7 500	±0%	10
1 000	10 000	10 000	±0%	10
1 100	11 000	10 120	−8%	9.2
1 200	12 000	10 560	−12%	8.8
1 300	13 000	10 920	−16%	8.4
1 500	15 000	12 300	−18%	8.2

porting target publishes/s, measured Redis ops/s, deviation from target, and the resulting sending rate (Hz).

To highlight the degradation that occurs toward the end of the run rather than masking it with an average, Figure 5.1 includes 5th–95th percentile error bars (the range within which 90% of the per-second values fall). These error bars are calculated over the entire test window, including the degradation tail for ≥1 100 players. Reading from Figure 5.1:

- From 250 to 1 000 players: There are tight error bars (p5–p95 approximately [9.9, 10.05]), indicating a consistent publish rate around the mean of 10 Hz.
- At 1 100 players (9.2 Hz mean, error bars at [8.6, 9.6]): The lower bar reflects a brief late run dip; however, the p5 value of ≈8.6 is above the 8.33 threshold, so it is still within the requirement of 120 ms in **R4** (degraded but acceptable).
- At 1 200 players (8.8 Hz, [8.4, 9.2]): The lower p5–p95 error bar extends further, with

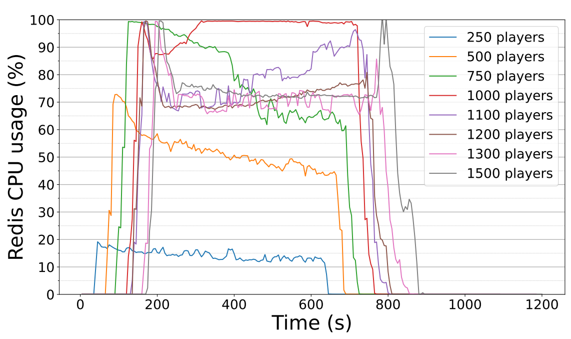

Figure 5.3: Redis Single-Core CPU Usage over Time.

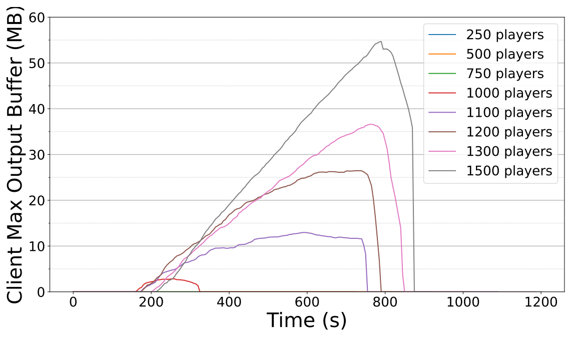

Figure 5.4: Max Client Output Buffer recorded by Redis over Time.

p5 ≈8.4≥8.33, so this is the maximum limit under **R4**.

- At 1 300 players (8.4 Hz, [7.9, 8.8]): The p5 error bar exceeds the threshold limit, placing this outside of the supported range.
- At 1 500 players (8.2 Hz, [7.6, 8.6]): The p5–p95 error bars show a large lower tail far below the supported range. In addition, the mean is also below the supported range.

CPU utilisation: In Figure 5.3, the utilisation climbs toward saturation as the player count increases, maintaining a nearly constant 100% utilisation at 1 000 players. Above this, the average CPU declines despite the increase in player numbers.

This behaviour could be attributed to the time spent in network I/O blocking (such as socket read/write), which could reduce the observed user-space CPU activity under overload conditions. However, more data is needed to confirm this hypothesis, so this remains a hypothesis for future work.

Max Output-buffer growth: When Redis cannot flush/distribute messages as quickly as it generates them, the messages accumulate in per-client output buffers. Figure 5.4 tracks the maximum buffer supported by Redis for each run. For player counts 250–750, the buffer stays at 0 MB for the whole run. The buffer briefly increases at 1 000 players, but is soon resolved. The 1 100 and 1 200 player runs increase quite drastically at the start and then flatten out about half way through the test. From ≥1 300 players, the curves have a large initial slope and no plateau, signalling that the buffer is out of control and will likely fill up and exceed memory limits if the test continues for a longer time. This timing aligns with the extended lower p5–p95 error bars in Figure 5.1 and Figure 5.2, indicating that the late-run dips occur in the same region where the buffers begin to saturate.

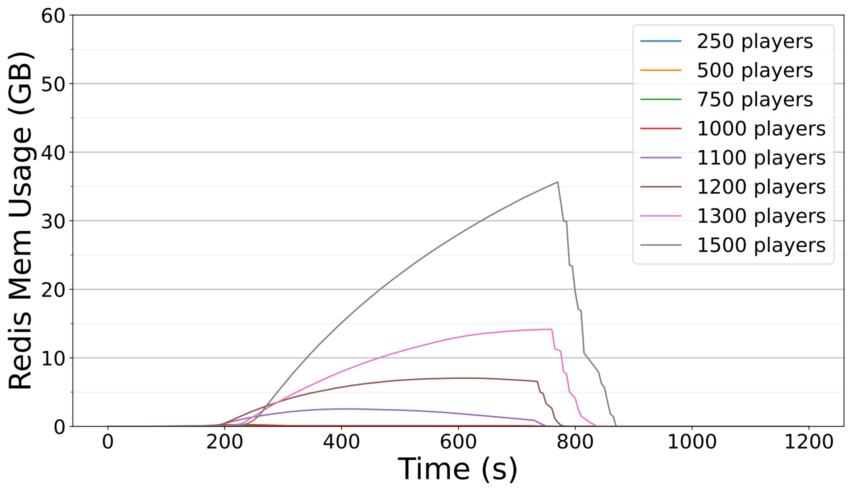

Figure 5.5: Redis Memory usage over the Test Time.

The rising buffers are consistent with back-pressure (Redis enqueuing messages faster than it can be drained). An example of how this back-pressure is created and relieved is that, during the 1 000 player run in Figure 5.3 and Figure 5.4, the output buffer starts to increase at the start but is soon resolved when the CPU utilisation increases to 100%. This means that the CPU is initially stalled, causing a build-up in the buffer, and when the CPU is free to work again, the buffer is emptied. This means that rising buffers could be consistent with CPU utilisation decreasing, which in turn could be caused by the CPU stalling or waiting for processes to finish (I/O sockets, for example). This hypothesis will need more research to be proven, so it will be set as future work.

Memory Usage: From 250 to 1 000 players, memory usage remains stable and minimal (less than 1 or 2 GB). With 1 100 players, there is a short increase to ≈3 GB at the beginning and then consistently decreases throughout the rest of the test. At 1 200 players, memory usage rises quickly to about 8 GB but then levels off and decreases slightly later in the test, indicating no prolonged increase. For 1 300 players, memory usage increases to ≈14 GB but does not stabilise or flatten, indicating that memory may exceed the limitations given enough run-time. Similarly, with a group of 1 500 players, there is a continuous increase in memory usage with no indication of levelling out, indicating that it is not sustainable for longer testing/playing periods and will eventually exceed memory limits.

The decline observed at 1 100 and 1 200 players implies that the client buffers are being drained and freed by the end of the run. As mentioned above, the memory usage at 1 300 players indicates slight stabilisation, but the residual positive slope could lead to a slow increase on longer runs. The persistent increase in memory usage observed at 1 500 players indicates that given additional time, memory consumption would likely continue to increase, potentially exceeding memory limitations. These interpretations are deduced from the behaviour of the observed data, as longer runs were not conducted to verify these hypotheses.

Combining Figures 5.1–5.5 and Table 5.1, it is evident that a population of 1 000 players comfortably achieves the 10 Hz target within the test window. This conclusion is supported by the tightly clustered p5–p95 error bars around 10 Hz, alongside stable output buffers and consistent memory usage. When the population increases to 1 200 players, the mean rate decreases to ≈8.8 Hz, with the p5 value ≈8.4 Hz, which is above the 8.33 Hz requirement (**R4**). We therefore treat 1 200 as the maximum supported number of players under these requirements, with the output-buffer growth monitored. At ≥1 300 players, the p5 metric falls below 8.33 Hz and both output-buffer ad memory usage indicate sustained expansion, thus categorising these scenarios as beyond the acceptable limits.

In conclusion, **MF1** enhances single-broker capacity to accommodate approximately 1 200 players under **R4**, maintaining 1 000 players as the conservative operating point when a 10 Hz sending rate must be preserved.

5.4 Single-channel Redis: How many players can we handle? (MF2)

The goal of this experiment is to measure how many concurrent players a **single channel Redis Pub/Sub setup** can support without exceeding the 120 ms (8.33 Hz) latency requirement (**R4**). To give the system a safety buffer and reflect gameplay needs, each client aimed to publish **10 updates/s**. Additionally, it gives us two metrics for the sending rate or throughput performance: *Good* at 10 Hz and *Minimum* at 8.33 Hz.

To test this, we incrementally increase the number of publisher bots (players) from 100 to 500 in steps of 100. The total offered load of each test is equal to players count × 10 messages per second. Every run lasts 600s after a short warm-up (varies as nodes increase), during which we record Redis-specific metrics, bot listener/sending metrics, and hardware utilisation metrics using Telegraf.

Firstly, when looking at Figure 5.6, we can see that the system supports up to around 400 players before going below the latency threshold of 8.33 Hz. The Redis broker delivered the full load up to 300 players. At 400 players the tick-rate dipped by 5% to 9 Hz, which is still compliant with **R4**. With 500 players the deficit widened to 20%, bringing the tick

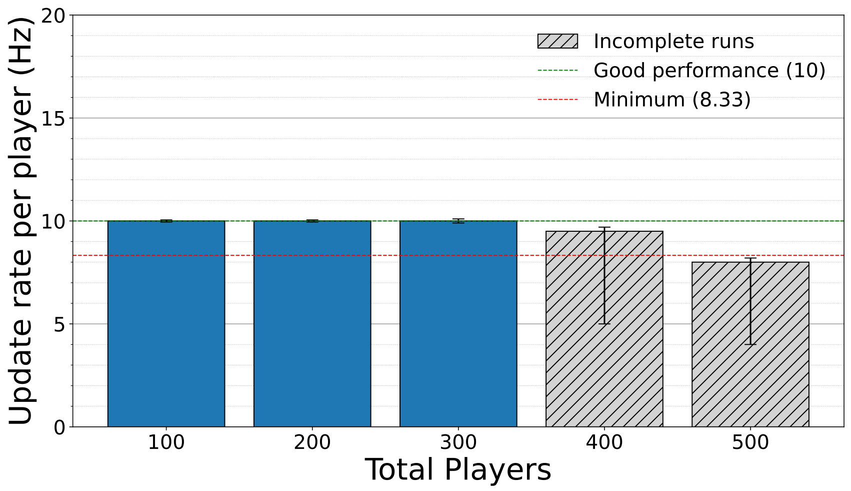

Figure 5.6: Update Rates as the Player Count Increases.

rate down to 8 Hz and exceeding the requirement. The gray cross on the runs with 400 and 500 players means that those runs did not complete and ran out of memory (`OOM`). This is discussed later in this section under Figure 5.10 and is the reason for 300 players being the max supported, even though we can handle 400 player throughput.

To make variability over the run visible rather than hidden by a single average, Figure 5.6 includes 5th–95th percentile error bars (p5–p95) calculated across the test duration for each player count. The bars represent the average update rate in Hz, while the error bars depict where 90% of the per-second values occurred. Since the percentiles are calculated for the entire duration of the test, the error bars also capture any decline encountered towards the end of the test.

For 100–300 players, the error bars remain close to 10 Hz, indicating stable delivery. However, at player counts of 400 and 500, the error bars display a lower skew due to the drop-off observed toward the end of the run. Initially, the runs adhere closely to the target, but as they progress, the performance decreases, dragging down the lower percentile, even though the upper percentile remains near the initial steady state.

Figure 5.7 and Table 5.2 summarises Redis-side publish throughput versus the target or expected load. Figure 5.7 uses the same p5–p95 error bars as Figure 5.6 on total ops/s. The data points lie exactly on the ideal line up to 300 players. At 400 players, the ops/s bends slightly before the run crashes with (`Out_Of_Memory`) and at 500 players, the ops/s is even further below the expected. The reduction in performance at the 400/500 mark cause the drop of the lower percentile, mirroring the lower percentile decrease seen in Figure 5.6. To explain this bend we examined the process-level CPU metrics that Redis makes available

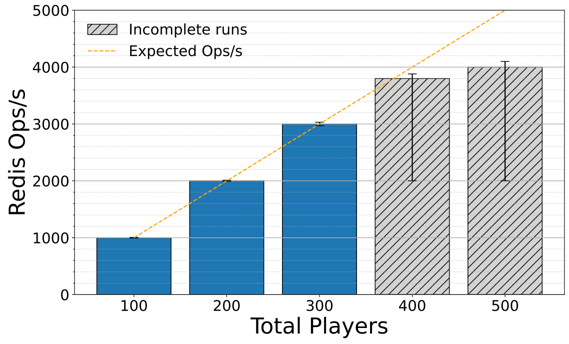

Figure 5.7: Redis Operations/s as Player Count Increases.

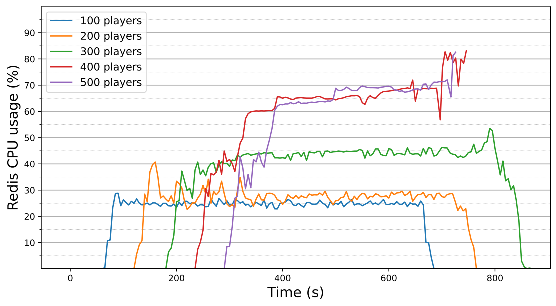

Figure 5.8: Redis Single-Core CPU Usage over Time.

Table 5.2: Results of single-channelled test at a 10 Hz publish target.

Players	Target publishes/s	Redis ops/s	Measured vs target	Outcome
100	1000	1000	±0%	Stable (600s)
200	2000	2000	±0%	Stable (600s)
300	3000	3000	±0%	Stable (600s)
400	4000	3900	−5%	OOM at ≈400s
500	5000	4000	−20%	OOM at ≈300s

through its `INFO` command. To understand the bend we inspected two additional metrics: Redis CPU utilisation and the size of the client output buffers. And to understand the `Out_Of_Memory` failure, we will look at Redis's memory usage.

CPU utilisation: Figure 5.8 shows the aggregate user + system CPU usage for the Redis process during each run. Since Redis has a single thread event loop, we plot the data of a single core of the CPU on the node. The lines for 100, 200, 300 players hover well below saturation, peaking near 30%, 40%, and 55% respectively. At 400 players, the curve climbs steadily until it plateaus at 65%, and then peaks at above 80%. The 500 player climbs faster and plateaus at about the 70% mark, and similarly peaks at above the 80% point. This behaviour indicates that the CPU of the single event-loop thread inside Redis becomes the bottleneck right around the 400-player mark, but more data is needed to confirm this. To this end we also consider the Max Output-buffer growth and Memory Usage below.

Max Output-buffer growth: When Redis cannot flush/distribute messages as quickly as it generates them, the messages accumulate in per-client output buffers. Figure 5.9 tracks the maximum buffer size reported by Redis for each run. For 100–200 players the buffer stays at 0 MB for the whole run. During the 300 player run, the buffer increases

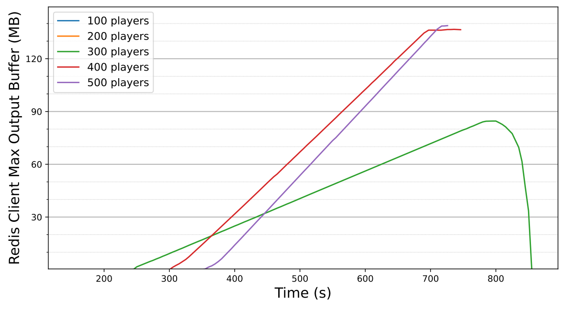

Figure 5.9: Max Client Output Buffer recorded by Redis over Time.

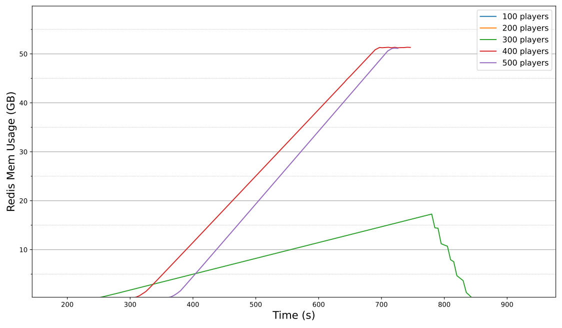

Figure 5.10: Redis Memory usage over the Test Time. *400 and 500 players did not complete the test due to OOM.*

.

gradually as a few slow clients struggle to empty their buffer quickly. However, the buffer does not get full, thus the messages are delivered in time and the throughput stays at the maximum. At 400 and 500 players the buffers grow linearly to about 140 MB before the process ultimately exits.

Memory Usage: Figure 5.10 shows the consequence of the output buffer being too full. All of the buffered messages are written to memory as it waits for the client or Redis's event loop to process it. When all the clients messages start getting buffered, the memory usage explodes to past 50 GB for the 400 and 500 player runs, at which point the kernel's OOM-killer or Redis itself terminates the process (there is no output log from Redis so likely the former). The CPU and Output Buffer plots stop writing at the same timestamp.

Increasing the player count raises the fan-out cost quadratically. The main thread can handle around 3 000–4 000 `PUBLISH` operations per second (≈300 players). Beyond that, CPU saturation slows socket flushes, messages queue in client buffers, leading to big memory increases, and the process is killed. The last population that avoids this chain for the full test duration is **300 players**, which we conclude as the safe single-channel limit.

5.5 Dynamic Pub/Sub: How does View Distance impact Player Count? (MF3)

The goal of this experiment is to assess the effect of varying view distances within the dynamic proximity-based pub/sub system on scalability. While **MF1** sets the subscription radius to a 3×3 grid (radius 1), in practice, game developers may want to expand view distance to enhance player immersion and player awareness. However, increasing the

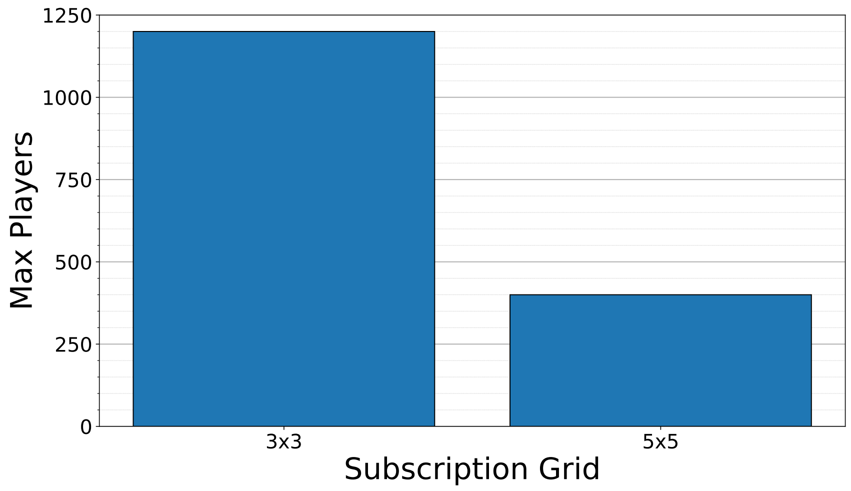

Figure 5.11: Max Players supported at varying Subscription Grids (View Distances).

subscription radius leads to an increase in the number of channels each player subscribes to, consequently increasing the fan-out costs of every published message. This experiment measures the trade-off between immersion (larger view distance) and throughput (maximum supported players).

The setup is identical to that of **MF1**, as detailed in Section 5.3. However, the subscription radius is increased, extending from 1 ($3\times3 = 9$ channels) to 2 ($5\times5 = 25$ channels). Each simulated player publishes messages at a frequency of 10 Hz to their current column channel and subscribes to all channels within their 5×5 view area. The number of players is scaled up to a maximum of 500. Metrics collected includes publishing throughput, CPU utilisation, growth of the client output-buffer, and memory usage.

Moving from a 3×3 view distance (radius 1) to a 5×5 view distance (radius 2) increases subscriptions from 9 to 25 channels per player. Consequently, system capacity diminishes, shown in Figure 5.11, from ≈1 200 supported players (**MF1**) to ≈400 players, marking a 66.67% decrease. This demonstrates a clear trade-off between enhanced immersion and the maximum number of supported players (scalability).

As shown in Table 5.3, the system's throughput becomes saturated at ≈4 000 operations per second when utilising a 5×5 view distance, which limits scalability to around 400 players.

At 250 players, the system is stable at 10 Hz. At 350 and 400 players, there is a slight decrease in the average update rate to approximately 9.9 and 9.8 Hz, though the range between the 5th and 95th percentiles remains compliant with the 8.33 Hz requirement (**R4**).

Table 5.3: Results of the dynamic 5×5 multi-channelled test at a 10 Hz publish target.

Players	Target publishes/s	Redis ops/s	Measured vs target	Sending Rate (Hz)
250	2 500	2 500	±0%	10
350	3 500	3 480	−1%	9.94
400	4 000	3 920	−2%	9.8
500	5 000	4 000	−20%	8

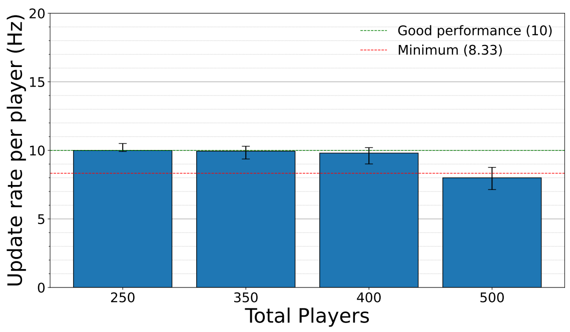

Figure 5.12: Dynamic Pub/Sub (5×5): Update Rates as the Player Count Increases.

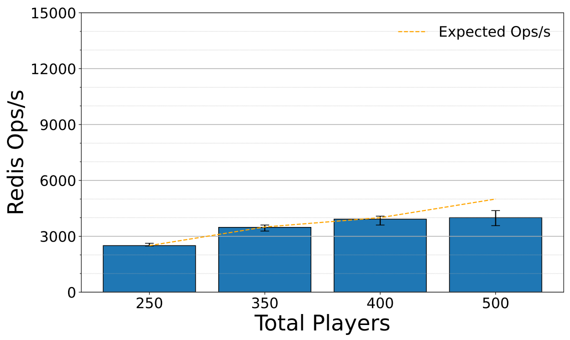

Figure 5.13: Redis Ops/s as Player Count Increases, excluding Sub/Unsub operations (5×5).

Upon reaching 500 players, throughput remains at the upper limit of 4 000 operations per second, and the average rate decreases by 20 % to 8 Hz, with the lower percentile dropping to roughly 7.1 Hz, which exceeds the latency requirement.

Figures 5.12 and 5.13 visualise this decline, showing narrow error margins for players numbering 350 or fewer, a broader variation at 400 players, and sustained declines at the 500 player mark.

In conclusion, increasing the view distance radius from 1 to 2 (3×3 to 5×5) decreases the maximum supported player count from approximately 1 200 players to around 400 players, resulting in a 66.67% reduction. This shows that the view distance is a crucial factor in scalability, as larger radii improve immersion as the cost of fewer concurrent players, while smaller radii maintain scalability at the cost of visibility in the game.

5.6 Effect of I/O Threads on Single-channel Redis: (MF4)

The goal of this experiment is to assess whether enabling additional Redis I/O threads can relieve the bottleneck identified in Section 5.4.

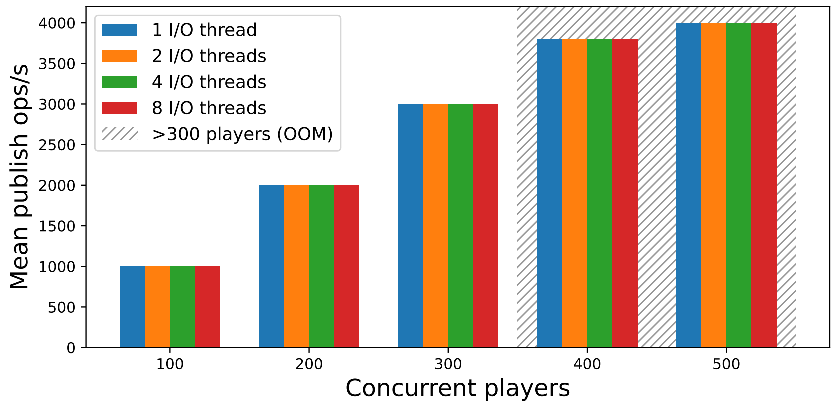

Figure 5.14: Effect of Redis I/O threads on the single pub/sub channel throughput.

For each player count (100–500) we repeated the 600s run with `io-threads` set to 1,2,4 and 8. All other parameters, including the single pub/sub channel and 10 Hz publish rate, were left unchanged. The only metric presented is publish throughput (ops/s) as seen in Figure 5.14. There was absolutely no change to the ops/s as the number of I/O threads were increased. Additionally the output buffer and memory behaviour match those already documented in Section 5.4. Runs at 400 and 500 players still terminate early due to Out-Of-Memory, at nearly the same timestamps as in Section 5.4.

Why did the extra threads not make a difference? I/O threads offload kernel `write()` calls but cannot parallelise the `PUBLISH` loop, which remains single-threaded. Only the core event loop can fan-out or distribute the `PUBLISH` commands. As the bottleneck is in this core event loop, increasing I/O threads provides no benefit. This negative result does not mean that the experiment is a waste, as it helps us locate the true cause of the bottleneck by removing the idea of it being socket-based. This result confirms that further optimisation must come from increasing the number of pub/sub channels (**MF1**) or having multiple Redis brokers.

6

Related Work

Research on scalable online multiplayer game architectures has long been a central topic in computer science, spanning areas like centralised servers, distributed peer-to-peer systems, cloud gaming platforms, and more recently, database-backed and serverless approaches. Each of these systems offers distinct advantages and limitations regarding scalability, fault tolerance, and consistency. This section provides an overview of prior work within four specific areas: (i) centralised and replicated server, (ii) distributed and peer-to-peer architectures, (iii) cloud gaming platforms, and (iv) adaptive, database-driven consistency models.

Centralised Servers and Replication: Early research on large scale multiplayer games centred around increasing the scalability of dedicated servers via replication and load balancing techniques. For instance, Lin et al. [12] developed *MiddleSIR*, a framework that applied replication protocols like primary-copy and eager update-everywhere to manage multiplayer states. Their findings showed that while replication could improve fault tolerance and scalability, it occasionally resulted in constraints affecting gameplay, such as batching or aborting updates. These findings illustrate how strongly consistent replication protocols, which generally perform well in databases, might adversely impact the responsiveness crucial to interactive gaming experiences.

Research conducted by Abdelkhalek et al. [13] examined the performance of early commercial game servers, identifying CPU and network demands as primary limitations as player numbers rose. Although their findings were based on early 2000s hardware, the underlying observation remains valid today: centralised, single threaded event loops tend to max out on CPU and network usage with increased concurrency. This architectural constraint continues to drive the exploration of alternative architectures, such as the database-centric approach discussed in this thesis.

Distributed and Peer-to-Peer Architectures: Another line of research explored distributed and peer-to-peer approaches. *Colyseus* [14] managed to distribute both the game logic and its state, blending strong consistency for crucial actions with weaker guarantees for frequent updates such as player movement. *Donnybrook* [15] focused on large-scale shooter games, utilising aggressive interest management techniques to manage hotspots involving thousands of players. Surveys conducted by Yahyavi and Kemme [16] provide comprehensive classifications of consistency strategies in multiplayer gaming, covering a spectrum of dead-reckoning to optimistic and exact methods. These explorations emphasise the importance of flexibly easing consistency guarantees to achieve scalability while maintaining interactive latencies, a principle also reflected in this thesis's treatment of movement and transactional state.

Cloud Gaming Platforms: Cloud gaming offers a distinct architectural approach by offloading rendering and simulation processes to data centres. Shea et al. [17] analysed systems such as OnLive and Gaikai, highlighting that while cloud offloading reduces the demands on clients, it introduces strict latency budgets: $\leq$100 ms for first-person games and up to 1 000 ms for strategy games. These findings establish clear benchmarks for responsiveness in any distributed system, encouraging techniques such as local prediction to hide network delay. Although this thesis does not specifically address cloud rendering, the latency limitations provide an external baseline for assessing interactive responsiveness.

Adaptive and Database-Backed Consistency: More recent work explores adaptive consistency models and database-centric architectures. Vector Field Consistency (VFC) [18] reduces communication demands by relaxing guarantees based on spatial distance. Complementing this, Donkervliet et al. [19] propose Dyconits, a middleware that dynamically groups the game state into units with configurable bounds on staleness and numerical inconsistency. These units adapt at runtime to player interests and workload patterns, reducing unnecessary updates and preserving interactivity. Karsai's 2024 BSc thesis [20] provides a modern evaluation of Dyconits within a pub/sub context, demonstrating their ability to dynamically adjust consistency in exchange for enhanced performance across various topics.

In parallel, research in distributed systems has highlighted how key-value stores like Dynamo, Cassandra, and Redis facilitate operations with low latency and high throughput at scale. Building on this foundation, Donkervliet et al. [3] have introduced serverless MVEs that utilise databases as the authoritative state store. Similarly, Eickhoff et al [7] introduced Meterstick, a benchmarking tool designed for Minecraft-like virtual worlds, aimed at assessing performance inconsistencies in both self hosted and cloud environments.

These works collectively support the feasibility of database-driven and serverless models for MVEs while highlighting the need for further empirical evaluations, such as those presented in this thesis.

Summary: This collection of research offers a comprehensive perspective on design possibilities, extending from centralised replication to distributed peer-to-peer systems, cloud architectures, and adaptive database-driven consistency models. Prior studies show: the trade-offs between consistency and scalability, the need for adaptive or flexible guarantees on consistency, and the latency limitations imposed by interactive games. This thesis contributes to this by developing and evaluating a prototype based on Redis, which integrates a database-managed state with publish/subscribe messaging. It illustrates that this methodology can support over a thousand concurrent users while maintaining responsiveness.

7 Conclusion

In today's expansive digital realms, the constraint is not the size of the map or the rendering capabilities, but rather the limitations of the servers that keep them coherent. This thesis explores whether a key-value store could take over the role traditionally assigned to monolithic game servers.

7.1 Answering Research Questions

In this section we will address the Main Research Questions proposed in Section 1.2.

RQ1 - Design: Chapter 3 highlighted the essential components present in modern key-value systems, such as: hashes, sets, publish/subscribe channels, and atomic compare-and-swap scripts. We also demonstrated their adequacy in addressing four functional requirements: player presence, real-time state synchronisation, conflict resolution, and durable world data. By dividing state into monotonic logs versus non-monotonic keys and applying the CALM principles, most updates can occur seamlessly without requiring coordination.

RQ2 - Implementation: Chapter 4 translated the design into KeyVerse, a Unity prototype directly leveraging a Redis 6 backend. It employs a sliding 3 x 3 proximity window to sustain constant message loads for each client. Additionally, micro-services explained in Chapter 3 like simulation, anti-cheat, and analytics can integrate without needing to modify the client code.

RQ3 - Evaluation: Chapter 5 revealed that with the dynamic pub/sub system (**MF1**) and a single Redis node, the system can support approximately 1 200 concurrent players under **R4** ($\geq$8.33 Hz), with the recommended 10 Hz operating point being approximately 1 000 players. Beyond 1 200 players, the Redis single-threaded publish loop capped throughput at ≈12 000 ops/s, reducing the per-player update rate below the target. This represents a nearly fourfold improvement over **MF2**, showing that dynamically splitting pub/sub traffic substantially delays CPU saturation. A single Redis node supports approximately 300 concurrent players on a single pub/sub channel at a sending rate of 10 Hz until the pub/sub fan-out loop becomes saturated (**MF2**). Providing additional I/O threads did not alleviate this issue, identifying the event loop, not the socket output, as the critical bottleneck (**MF4**). **MF3** shows that increasing the view distance to a 5×5 grid decreases the capacity to approximately 400 players, representing a reduction of roughly 67%. This decrease is expected and provides a measure of the trade-off between immersion and scalability. The larger the view distance, the lower the supported player count, and the lower the view distance, the higher the supported player count.

7.2 Limitations and Future Work

This section will bring light to some limitations faced during this thesis and possible future projects.

Single-broker ceiling: The experiments revealed that the while the baseline single-channel design could accommodate approximately 300 players (**MF2**), the dynamic pub/sub system (**MF1**) increases this limit to approximately 1 200 players. This confirms that a single broker can only scale so far, and future work should explore: the deployment of multiple brokers, or alternative mechanisms for event distribution to surpass this limitation.

Best-effort Consistency: Most keys can rely on the last-write-wins approach, but unique items sill need CAS or transactional pipelines.

Scope of workload: While benchmarks have been concentrated on movement updates, other activities such as crafting, terrain modifications, and high-frequency combat deserve dedicated stress tests.

7.3 Closing Remark

KeyVerse suggests a simple paradigm shift: let the database become the server. By integrating authoritative state and real-time messaging within the key-value store, we successfully separate Simulator and State, paving the way for serverless , horizontally scalable virtual worlds. While the prototype exposes the limitations of a single-broker design, it also highlights the opportunities of database-backed architectures as a foundation for future large scale MVEs and hints that this future may lie one well-placed **SET** command away.

References

[1] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin van Eyk, Lucian Toader, Vincent van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The AtLarge Vision on the Design of Distributed Systems and Ecosystems**. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776, 2019. 3

[2] Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos. **Behavior and performance of interactive multi-player game servers**. *Cluster Computing*, **6**:355–366, 2003. 7

[3] Jesse Donkervliet, Animesh Trivedi, and Alexandru Iosup. **Towards Supporting Millions of Users in Modifiable Virtual Environments by Redesigning Minecraft-Like Games as Serverless Systems**. In Amar Phanishayee and Ryan Stutsman, editors, *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2020, July 13-14, 2020*. USENIX Association, 2020. 8, 46

[4] Matthias Dick, Oliver Wellnitz, and Lars Wolf. **Analysis of factors affecting players' performance and perception in multiplayer games**. pages 1–7, 10 2005. 10

[5] Valentin Forch, Thomas Franke, Nadine Rauh, and Josef Krems. **Are 100 ms Fast Enough? Characterizing Latency Perception Thresholds in Mouse-Based Interaction**. pages 45–56, 05 2017. 10

[6] Shengmei Liu and Mark Claypool. **The Impact of Latency on Navigation in a First-Person Perspective Game**. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery. 10

[7] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. **Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games**. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 147–149, 2022. 10, 46

[8] Günter Wallner. **A brief overview of data mining and analytics in games**. *Data analytics applications in gaming and entertainment*, pages 1–14, 2019. 14

[9] Joseph M. Hellerstein and Peter Alvaro. **Keeping CALM: when distributed consistency is easy**. *Commun. ACM*, **63**(9):72–81, August 2020. 15

[10] Henri E. Bal, Dick H. J. Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank J. Seinstra, Cees G. M. Snoek, and Harry A. G. Wijshoff. **A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term**. *IEEE Computer*, **49**(5):54–63, May 2016. 32

[11] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. **Yardstick: A Benchmark for Minecraft-like Services**. In *Proceedings of the International Conference on Performance Engineering, Mumbai, India, April, 2019*, 2019. 32

[12] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. **Applying database replication to multi-player online games**. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, page 15–es, New York, NY, USA, 2006. Association for Computing Machinery. 45

[13] Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos. **Behavior and performance of Interactive Multi-player Game Servers**. *Cluster Computing*, **6**(4):355–366, Oct 2003. 45

[14] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. **Colyseus: a distributed architecture for online multiplayer games**. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, page 12, USA, 2006. USENIX Association. 46

[15] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. **Donnybrook: enabling large-scale, high-speed, peer-to-peer games**. In *Proceedings of the*

ACM SIGCOMM 2008 Conference on Data Communication, SIGCOMM '08, page 389–400, New York, NY, USA, 2008. Association for Computing Machinery. 46

[16] Amir Yahyavi and Bettina Kemme. **Peer-to-peer architectures for massively multiplayer online games: A Survey**. *ACM Comput. Surv.*, **46**(1), July 2013. 46

[17] Ryan Shea, Jiangchuan Liu, Edith C.-H. Ngai, and Yong Cui. **Cloud gaming: architecture and performance**. *IEEE Network*, **27**(4):16–21, 2013. 46

[18] Manuel Cajada, Paulo Ferreira, and Luis Veiga. **VFC-RTS: Vector-Field Consistency para Real-Time-Strategy Multiplayer Games**. *Master of Science Disertation*, 2012. 46

[19] Jesse Donkervliet, Jim Cuijpers, and Alexandru Iosup. **Dyconits: Scaling Minecraft-like Services through Dynamically Managed Inconsistency**. In *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*, pages 126–137. IEEE, 2021. 46

[20] Martin Karsai. *Dynamically Managed Inconsistency in Distributed Systems.* PhD thesis, Vrije Universiteit Amsterdam, 2024. 46