Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

A Framework for Creating Distributed Online Games Powered by Apache Kafka

Mark Hanrahan

April 19, 2022

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MAI (Computer Engineering)

Abstract

Online gaming is more popular than ever before, with a consistently increasing player population and a trend to build large-scale online games. These large-scale games are growing in number with increasing virtual-world size, simultaneous player count and complexity.

The problem with building large-scale games with increasing virtual world size, number of simultaneous players, and complexity is the cost this requires. It is a financially expensive task to build and support distributed and scalable online games. In addition to this, it requires time and extensive research. Often, this research is completed by companies with no vested interest in sharing or distributing it. This results in a repeated effort to achieve similar goals. This cost can directly restrict the accessibility of development of these large-scale online games and experiences.

The goal of this work is to present and evaluate the design of a model and framework that can be used to create distributed, online games. The framework was implemented through software and used to build an example distributed online game. This approach was evaluated with respect to reusability, accessibility, performance and reliability. The results of this work show that this framework can support 100 concurrent players per partition with 90ms average event request latency and an event loss rate of 1.33

Acknowledgements

It is my genuine pleasure to thank Professor Vinny Cahill for providing guidance, corrections, welcomed criticism and his extensive knowledge in the field of distributed systems throughout the duration of this research. This work could not have been achieved without his support, and for that, I am deeply grateful.

To my colleagues and friends, Mayank, David, and Aditya it was my privilege to work academically alongside you for the past few years and I am grateful that you continuously pushed the bar higher for my work.

I'm incredibly thankful to my family for supporting me throughout my academic career, from providing me with my outstanding secondary school education to supporting me during College. To my mother, Vivienne and father, Edward, I am glad you believed in me. To my sister Chlöe, thank you for always challenging me intellectually and otherwise. To Claudia, I am thankful for your wit and dedication to the field of small rodents. To Daniel, I am thankful for so many reasons, namely your determination and kindness.

I owe a debt of gratitude to my good friend Lorcan who has always had my back and even especially during the past year. Thank you.

Contents

1 Introduction **2**

1.1 Problem 4

1.2 Motivation 4

1.3 Approach 5

1.4 Challenges 5

1.5 Contributions 5

1.6 Structure of Report 6

2 Related Works **7**

2.1 Consistency 7

2.1.1 History 7

2.1.2 Consistency Models 8

2.1.3 Data-centric Consistency Models 8

2.1.4 Client-centric Consistency Models 10

2.2 Online Gaming 11

2.2.1 Background 11

2.2.2 Latency Requirements 12

2.2.3 Single Server Architectures 13

2.2.4 Distributed Architectures 13

2.2.5 Distribution Strategies 14

2.2.6 Compensatory Techniques 16

2.3 Apache Kafka 18

2.3.1 Event Streaming 18

2.3.2 Automation 18

2.3.3 Architecture 18

2.3.4 Kafka Terminology 19

2.3.5 Kafka for Online Games 21

2.4 Summary 22

3 Model **23**

3.1 Game State 23
3.1.1 Local States 24
3.2 Objects 24
3.2.1 Local Objects 25
3.2.2 Global Objects 25
3.2.3 Diffuse Objects 26
3.2.4 Abstraction 26
3.2.5 Critical vs. Non-critical 26
3.3 State Transfer 27
3.3.1 Message Protocol 27
3.3.2 Event-driven Backbone 28
3.3.3 Event Streams 29
3.4 Partitioning 30
3.5 Consistency 31
3.6 Area of Interest 31
3.7 Archetypes 32

4 Implementation **33**
4.1 Framework 33
4.1.1 Event-driven Backbone 33
4.1.2 Framework API 33
4.1.3 Event Loop 34
4.1.4 State Transfer 35
4.1.5 Priority Streams 35
4.1.6 Partitioning 35
4.1.7 Replication 36
4.1.8 Architecture 36
4.2 Application of Model 37
4.2.1 The Game 37
4.2.2 The Requirements 37
4.2.3 The State 38
4.2.4 State Transfer 39
4.2.5 Partitioning and Replication 39
4.2.6 Consistency 39
4.2.7 Area of Interest 40
4.3 Development Process 41
4.3.1 The Environment 41
4.3.2 The Framework API 41
4.3.3 The Game 43

5 Evaluation **46**
5.1 Model . . . 46
5.1.1 Qualitative . . . 46
5.2 Framework . . . 47
5.2.1 Qualitative . . . 47
5.2.2 Quantitative . . . 48
5.3 Implementation . . . 50
5.3.1 Qualitative . . . 50
5.4 Limitations . . . 54
5.5 Summary . . . 55

6 Conclusion **56**
6.1 Summary . . . 56
6.2 Key Findings . . . 56
6.3 Limitations of Work . . . 57
6.4 Future Work . . . 57
6.5 Final Conclusion . . . 57

List of Figures

2.1 Consistency example described in (Garcia-Molina & Wiederhold 1982) (redrawn), 'red' describes inconsistency in the state of a node. 9
2.2 Zoning illustrated, this figure was extracted from (Glinka et al. 2008). . . 15
2.3 Instancing illustrated, this figure was extracted from (Glinka et al. 2008). 15
2.4 Zoning illustrated, this figure was extracted from (Glinka et al. 2008). . . 16
2.5 Architecture of an example Kafka cluster, with one zookeeper node, three brokers and two client nodes. 19
2.6 Illustration of an event in Kafka composed of key, value, and timestamp. 19
2.7 High level illustration of consumer and producer interaction with Kafka cluster. 20
2.8 Illustration of topics, partitions, and offsets within a cluster. 21

3.1 Game illustrated as a state machine with two states, a magnified view of the state machine is also displayed. 23
3.2 The state of the game divided into several "local" states. 24
3.3 Illustration of a Local object. 25
3.4 Illustration of a Global object. 26
3.5 Illustration of Critical and Non-critical objects. 27
3.6 Illustration of message protocol. 27
3.7 Illustration of multiple game clients producing events to the backbone. . 28
3.8 Illustration of priority streams. 29
3.9 Illustration of example partitioning strategy for figure 3.2. 30
3.10 Illustration of area of interest technique. 31
3.11 Illustration of game genres. 32

4.1 Framework API described to connect the game with the event-driven backbone. 34
4.2 Illustration of event loop. 35
4.3 Illustration of two backbone deployment options, using players to host nodes (left), and using cloud services provider to host nodes (right). . . . 36

4.4 Illustration of the chosen game for implementation. 37
4.5 Illustration of the game state. 38
4.6 Illustration of how partitioning was implemented via horizontal sectioning. 39
4.7 Illustration of the area of interest (AOI) implementation. 40
4.8 Illustration of the event loop in final game code. 44

5.1 Average latencies for concurrent groups of players, with standard deviation as error bars, following the above experimental setup. 49
5.2 Density distribution of latencies for concurrent groups of players, following the above experimental setup. 49
5.3 Average latencies demonstrated for a data center deployment, with standard deviation as error bars, following the above experimental setup. . . 51
5.4 Closer look at average latencies. 51
5.5 Density distribution of latencies for concurrent groups of players (5 to 50), following the above experimental setup. 52
5.6 Density distribution of latencies for concurrent groups of players (50 to 100), following the above experimental setup. 52
5.7 Event loss rate (events/100 requests) expressed as an average compared with an increasing number of concurrent players. 53

List of Tables

2.1 This table displays the latency requirements specified in (Brandt 2009) . 12

4.1 Description of 'Objects' created in this game. 44

5.1 Average event loss rate in tabular form. 53

Nomenclature

DSM	Distributed Shared Memory
ISR	In-state Replica
FPS	First-person Shooter Game
RTS	Real-time Strategy Game
MMO	Massively Multiplayer Online Game
P2P	Peer-to-peer
RTP	Real-time Transport Protocol
IP	Internet Protocol
PDF	Probability Density Function
Frame	An iteration of the execution loop within a game
Player	The primary user of a game client, also referred to as a “user".

"There is an old network saying: Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed–you can't bribe God"

– David Clark, *MIT*

1 Introduction

In 2020 the global gaming population was estimated to be around 3.1 billion (DFC 2020). This number is expected to grow as we evolve into a more online-oriented generation. In 2020 alone, global gaming revenue grew by 23% (Williams 2022), thanks in no small part to the covid-19 pandemic. These statistics convey the popularity and growth of the general video game industry.

According to the popular PC game distribution platform Steam (Valve 2022*d*), of the ten most popular games played (Valve 2022*c*), nine are classified as "online" games. Of these, seven can only be played online. According to these statistics, for the 24 hour period (26/03/2022), the peak concurrent user counts for the top 3 games were 980 thousand, 720 thousand, 560 thousand, for *Counter-Strike: Global Offensive* (Valve 2022*a*), *Dota 2* (Valve 2022*b*) and *Elden Ring* (Valve 2022*b*) respectively. In this example, "concurrent" users refer to the total number of people who have launched the application for that game. These numbers offer an indication of the concurrent load and scale at which these games operate. However, these only include accounts logged into the game from the PC platform, Steam.

To further demonstrate the popularity scale of each game session, consider the industry leaders *Fortnite* (2017) (Epic 2022*a*) and *Roblox* (2006) (Corporation 2022). For background on these games, *Fortnite* is a competitive game where users, also referred to as "players", compete in game sessions. The terms users and players will be used interchangeably in this report. A "game session" consists of a single match that simultaneously supports up to 100 players in a large virtual space. The last player standing wins the game. Therefore, each game session is required to scale to 100 players in a performant and consistent manner as it is a competitive experience. *Roblox* is a game that can support up to 700 players per game session. The game session itself is a virtual space where players can create, chat with other players, and explore user-generated virtual environments. These numbers represent the scale of concurrent players within a session in these popular online games.

While these games maintain a high level of scale, with a large number of concurrent players within each game session performance also needs to be specified. The performance measure of focus in this work is latency. According to the 'British Esports Association', a latency of between 5 and 60ms is considered suitable for competitive play within first-person shooter games (FPS), while any higher than 100ms is generally undesirable for competitive play, but may be suitable for other game types (BSA 2022). *Fortnite* is a competitive third-person shooter (TPS) and requires a similar level of performance to an FPS. Relative to *Fortnite*, *Roblox* does not require the same degree of low latency, with community guides (Anon 2022*a*) suggesting a latency above 1000ms will demonstrate visible lag in the game client.

According to the Epic Games 2020 quarterly business review (Epic 2020), *Fortnite* has amassed over 350 million registered user accounts, with 80 million monthly active users. The all-time peak concurrent user count was 15.3 million (FortniteGame 2020) across all game sessions. This all-time peak came from the introduction of a so called "live-event", a virtual experience such as a virtual concert or combined effort activity. To reiterate, concurrent users refer to the number of users who have launched the application for that game simultaneously. Monthly active users refer to the number of users who have launched the game application within that month. *Roblox* has over 230 million registered user accounts, with an all-time peak concurrent user count of 5.7 million users (Dean 2022) across all game sessions. These games are massively ambitious and built using complex and expensive systems that scale to support large numbers of users. These numbers indicate the massive scale these online games can eventually reach. The key statistics that most relate to this work are the number of concurrent users. This data represent the current upper bound for cross-session scale in the industry.

While not the first to introduce large-scale games, *Fortnite* has popularised the idea of building bigger. *Fortnite* is a game that supports a capacity of 100 players in a game session, with a map size of 3.5km^2. This influence can be seen in many highly popular games today. Examples include EA's Apex Legends (EA 2022) with 60 player game sessions and a 4km^2, Activision's Call of Duty: Warzone (Ward 2022) with 150 player game sessions and a 9km^2 game world. Numerous other games continue this trend. These examples further demonstrate the influence and trend to build large-scale games with more simultaneous players in a single game session.

Online gaming has now become more ubiquitous than ever before. The games that support this industry are similarly growing in scale. In addition to this, the line between a game and a virtual experience is growing increasingly thin. These games and experiences are becoming more ambitious, with more concurrent players and large, complex worlds.

1.1 Problem

This work is concerned with increasing accessibility in the design and implementation of scalable and consistent online games and similar virtual world experiences. Accessibility in this work relates to the cost and difficulty in designing and building for scale in games.

1.2 Motivation

This work is motivated by the following factors:

Firstly, it is motivated by the growing demand for online, large-scale, shared-world experiences. It has been shown that these online games and virtual world experiences are increasing in popularity and scale. The increased demand catalyses and exposes weaknesses in the development process, such as the lack of publicly available standard practices. A standard approach and framework would greatly benefit the speed and development process.

Secondly, it is motivated by the difficulty in building distributed online games and virtual worlds. As these are distributed systems, creating these systems is a non-trivial task. This is because specific problems begin to arise, such as maintaining consistency between different views of the game and ensuring the cost of consistency does not negatively impact the performance and scalability of the game. If there existed a standard model to define and build scale in online games, this would reduce the difficulty as many problems would already be addressed or highlighted for concern.

Thirdly, it is motivated by the high cost associated with building these games and experiences. This cost is both financial and time-related regarding research, design, implementation and maintenance. An example of cost can be seen from companies like Meta, which recently invested 50 million dollars (Clark 2021) in developing new Metaverse technologies and spending vast amounts of time and research into this area. If there were a set of standard methodologies, then the goal of designing and building these virtual experiences would be far less costly to reach.

This study is motivated by these three main concerns. Resolving these concerns would greatly benefit independent developers and organisations that lack the resources required to fund private research and development necessary to build scale in online games and other virtual experiences.

1.3 Approach

Accessibility was increased through the creation of a reusable model which can be adapted through abstraction to many game types to design online, distributed games and experiences. The model was applied through a low-cost framework by using open-source software.

The framework was event-driven and horizontally scaled. The backbone of this framework was Apache Kafka. This framework was concerned with permitting scale in virtual worlds while not compromising on consistency and performance; this was important to allow for competitiveness in games. With this in mind, performance concerning latency and throughput to support player load should be satisfactory for various games.

This work was evaluated based on the accessibility of the model and framework by designing and implementing a toy example using these methods. The toy example demonstrated the possibility of scale by increasing the number of concurrent players in a game session and measuring the performance and reliability change. The performance was measured with respect to latency. Reliability was measured with respect to the rate of lost requests (in the form of events).

1.4 Challenges

Below are some of the main challenges that this work is faced with addressing:

- Reusability: How the model and framework can be abstract enough to be applied to multiple types of games yet provide a standard approach.
- Consistency: How the model and framework achieve consistency and to what level of consistency it can provide.
- Performance: How performance can be achieved at scale and to what level.

1.5 Contributions

This work presents the following contributions:

- A standard model used to design distributed, scalable and consistent online games and virtual experiences.
- A software framework implemented through the combination of a Python API and event-driven backbone. The concepts of the model are realized in the API. This backbone is applied through Apache Kafka.

- An implementation of an online, distributed, scalable and competitive online game using the methods proposed in this work.
- A qualitative and quantitative analysis of the model, framework and implementation as well as recommendations for future work.

1.6 Structure of Report

The report is divided into six chapters:

- Chapter one provides an introduction to the reader to the field relevant to this study as well as outlining the problem, motivation, approach, challenges and contributions of this work.
- Chapter two describes the state of the art in this field of study as well as extended background information.
- Chapter three describes the model, and abstractions core to this work.
- Chapter four details the implementation of the framework, application of the model and development process of an example game.
- Chapter five describes the evaluation of the model, implementation and framework through qualitative and quantitative means.
- Chapter six provides the conclusion for this work.

2 Related Works

This chapter discusses the state of the art in the fields relevant to distributed online gaming. The objective of this chapter is to perform a review of the field to draw ideas and concepts which inspire the main contributions of this work. The major areas that are reviewed in this chapter concern Consistency in distributed systems, Online multiplayer gaming and the Apache Kafka software product.

2.1 Consistency

A major challenge of this work surrounds consistency. Consistency is the abstract idea that describes when two or more processes agree on the same value for an item of data. These processes can be located on the same computer system (i.e. inter-process communication) or, in the context of distributed systems, separate computer systems altogether. The agreement refers to both reading and writing to that data. There are multiple layers to consistency and these are described by consistency models. Consistency models describe in detail the relationship between processes and data items to provide a set of theoretical "guarantees" associated with that model. In short, when, how and to what level of consistency is achieved. However, in nature, there are rarely absolute guarantees. This section first describes a brief history of consistency followed by some of these consistency models.

2.1.1 History

In 1985 the study of the distributed shared memory (DSM) allowed for abstraction in the concept of shared data within a system of distributed computer systems (Cheriton 1985). The concept of inter-process communication was extended to create a theory for DSM. This was achieved through asynchronous message passing. This created the same problems encountered in the field of concurrency, inconsistency between what is written and read by independent processes. From this increasingly common issue, the idea of consistency models was born. (Steinke & Nutt 2004) phrased consistency models quite well as "It can be seen as a contract between the memory implementation and the program utilising memory". Consistency models also allowed developers to build programs for DSMs without requiring knowledge of the underlying memory implementation (Steinke & Nutt 2004). This benefit of abstraction was noted by this study and employed in this work through the model, which presents similar abstractions available to the game developer. (Cheriton 1985) had argued that at the time, distributed file systems were suitable to handle DSM for most applications, however, we feel this sentiment has aged poorly, as the requirements of applications have shifted dramatically with the rise in popularity of internet applications such as internet banking and card payment technology and generally any website that requires reasonable scale in modern-day.

2.1.2 Consistency Models

The problem with achieving and maintaining Consistency is notably timeless as over thirty years later it is one of the focal points of this review. With extensive research, models have been designed over the years and can largely be separated into two major categories, data-centric consistency models and client-centric consistency models. Data-centric Consistency models define a set of guarantees with respect to the data items, whereas client-centric describes the guarantees with relative to each client of the shared data.

2.1.3 Data-centric Consistency Models

This subsection describes various data-centric consistency models relevant to this work.

Strong and Weak Consistency

Data-centric models are defined along a spectrum from "strong" to "weak" consistency. (Garcia-Molina & Wiederhold 1982) described that strong consistency requires the schedule of all writes and reads to data items to be consistent across all systems. He further elaborates that "since a consistent schedule is equivalent to some serial schedule, all transactions in the schedule read consistent data." Therefore all updates to data items would need to be synchronised across all computer systems in the distributed system.

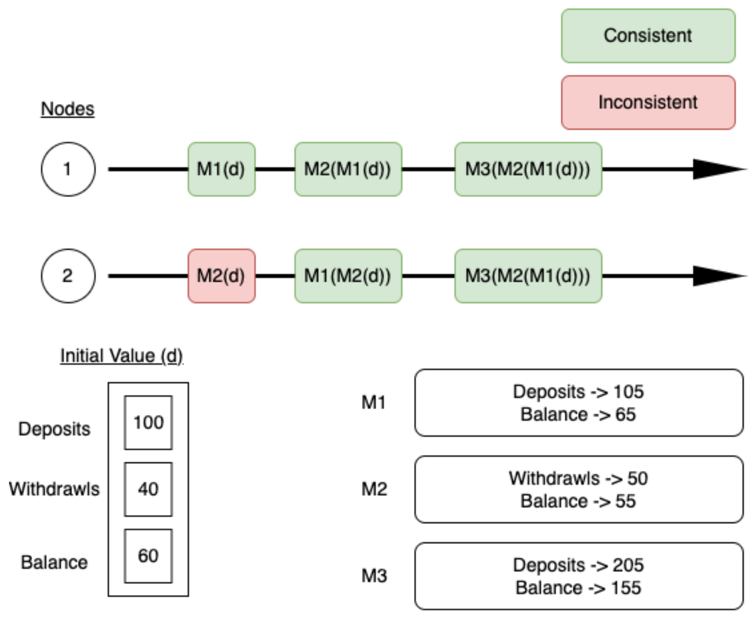

Figure 2.1: Consistency example described in (Garcia-Molina & Wiederhold 1982) (re-drawn), 'red' describes inconsistency in the state of a node.

Weak consistency is described by (Garcia-Molina & Wiederhold 1982) through a banking example. The example imagines two nodes of shared data with initial value d, and three messages M0, M1, M2, where M0 arrives first and M2 arrives last. According to this definition, weak consistency allows the reading query to read any values that are consistent, i.e. the deposit and withdrawals result in the balance. Comparing this to strong consistency, strong can only read the latest consistent value.

Causal Consistency

Causal consistency, first described by (Hutto & Ahamad 1990) as "Causal Memory", describes a stronger guarantee than Weak consistency. (Hutto & Ahamad 1990) built from the consistency model from the ideas laid by (Lamport 1978) on partial ordering. Partial ordering assumes there is some order between events of a process, which means there is a sub-sequence of events (updates in this case) that might relate a set of events. If a partial order exists between two updates, then those updates must happen in the sequence of that partial order. This can be seen as a causal relationship, also referred to as "happens-before". In the banking example, M0 needs to happen before M2 so that the state within each node is consistent. In this work, partial ordering is employed as the basis for our hypothesis that there is a partial order between events in a partition of the game and for consistency in gameplay, only those partitions of interest need to be kept consistent.

Sequential Consistency

Sequential consistency is a stronger requirement than causal consistency and builds on this idea. This model requires a total order across all updates within the system. This was described by (Dubois et al. 1986) through (Lamport 1979) as a solution to achieving consistency in shared memory between multiple cores in a multi-core processor. This idea was abstracted and carried over to distributed systems, where it describes the "Sequential Consistency" model. This model requires the “total order” of events described in (Lamport 1979) to be recognised by all systems. In the banking example above, in sequential consistency, each node would process the received events in the same total order as all other nodes, thus achieving consistency. Sequential consistency is implemented later in the methods proposed in this work, more specifically in the model chapter within streams.

Eventual Consistency and CAP

First proposed by (Vogels 2009), "Eventual Consistency" assumes that if there are no more updates to data items, then all replicas will hold the same consistent values at some point in the future. Eventual consistency is a stronger requirement than Weak consistency as it guarantees that the consistent values read will eventually also be the up-to-date/current values. (Vogels 2009) presented the concept of eventual consistency as means to enable a “worldwide scale”, and we agree with his hypothesis. This is because while Eventual Consistency is limited in its use-cases, it demonstrates the relationship between availability and consistency as defined by the CAP theorem from (Brewer 2000). The CAP theorem was a conjecture presented by Brewer without evidence and is heavily citepd in this field. While his observations are considered by many as fact, the theorem was proved in detail by (Gilbert & Lynch 2002). The CAP theorem suggests that of the three objectives in distributed systems: “Consistency”, “Availability”, and “Performance” only two can be achieved. While this theorem has a proof detailed in (Gilbert & Lynch 2002), the CAP theorem has been observed in the evaluation of our work.

2.1.4 Client-centric Consistency Models

Client-centric consistency models describe how each client of the system interacts with the data of the system to gain client-level consistency. In other words it acts a the contract of consistency between an individual client and the data of the system. Various kinds of client-centric models are described below.

Monotonic Reads Consistency

“Monotonic reads”, proposed by (Tanenbaum & Van Steen 2007) is a consistency model that describes how each client of the system interacts with the data. This model guarantees that if a client reads a data item, that client will never read a previous instance of that item. As this is a client-centric model, this guarantee would only stand for that client process.

Monotonic Writes Consistency

“Monotonic writes”, proposed by (Tanenbaum & Van Steen 2007) is a client-centric consistency model that guarantees that a client process will partially order their own updates to a data item in the order they were declared. For example, if a client writes the value ‘x’ to data item 1 and then writes the value ‘y’ to that same data item, the client would complete the initial write before starting the second write.

Read-your-writes Consistency

“Read-your-writes” consistency proposed by (Tanenbaum & Van Steen 2007) is a client-centric consistency model that guarantees the availability of the previous update of a data item made by that client process. This means that if a client process assigns a value of ‘y’ to data item 1 then that client process will at least be able to return that value on a read to data item 1.

Writes-follow-reads Consistency

“Writes-follow-reads” consistency proposed by (Tanenbaum & Van Steen 2007) is a client-centric consistency model that guarantees a write by a client process will take place on the last read value or more recent value of a data item.

2.2 Online Gaming

This section represents the research portion in online gaming. First a general background will be discussed, then latency requirements for various games, current architectures, distribution strategies, and finally compensatory techniques.

2.2.1 Background

Online gaming is an internet application and has been a growing area of academic and industrial interest over the last 20 years (Diot & Gautier 1999). Due to a combination of factors, namely rapid hardware advancement and the mass adoption of the internet, gaming formed one of the largest revenue shares of the entertainment media industry.

This, in turn, catalysed the development of new technology and innovation, specifically inspired advancement in the field of multi-server and distributed internet applications. Online gaming can be performed over many communication capable mediums such as WiFi, LAN, Bluetooth, 3G etc. This work focuses on the internet-based form of online gaming which can come in different architectures, mainly single-server and multi-server.

2.2.2 Latency Requirements

An interesting quirk of online gaming is that, while different online games can function in similar ways they can require different performances by their audiences. It should be noted that there is a distinction between noticeable performance and tolerable performance. Noticeable latency is the threshold at which latency begins to be felt by the player. Tolerable latency is the threshold at which the player begins to have an unpleasant experience within the game. (Brandt 2009) collated the responses of several research papers on the subjective performance requirements of players. While that work acts as a good indication of the kinds of requirements these games can have, it is far from an exhaustive list and as it was published in 2007, gaming demands, in general, have also changed since then.

Genre	Noticeable Latency	Tolerable Latency
RTS	250ms	800ms
Sports	n/a	500ms
Driving	50ms	100ms
FPS	75ms	100ms
MMORPG	n/a	1250ms

Table 2.1: This table displays the latency requirements specified in (Brandt 2009)

To explain the above table 2.1, For each genre above, either one or two games were analysed in research papers. The genres and research papers are outlined below.

- *RTS* stands for real-time strategy; this is a game where players command a group of units to battle the other player's units. The games surveyed were *Age of Empires* and *Warcraft III* were surveyed in (Bettner & Terrano 2001) and (Sheldon et al. 2003) respectively.
- The *Sports* game *NFL* (2004), a real-time American football game was surveyed in (Nichols & Claypool 2004).
- The *Driving* game was custom built for research in (Pantel & Wolf 2002).
- *FPS* stands for first-person shooter which is a game that requires fast reflexes

from the player and therefore, lower latency. The game *Unreal Tournament* (2003) was researched in (Beigbeder et al. 2004).

- *MMORPG* is a combination of the genres “Massively Multiplayer Online” and “Role-playing Game”. These games are large in scale with many concurrent players but typically do not require fast reactions from the player. The game researched was *Everquest II* in (Fritsch et al. 2005).

2.2.3 Single Server Architectures

Initially, multiplayer games of the early 2000s followed a traditional client and server model, games such as: *Halo: CE*, *Counter Strike*, and *TF2*. This approach was appropriate to its scale as online games were small and relatively simple. In the case of *Counter-strike*, one of the most influential online games of the early 2000s, up to 22 players would communicate to a single centralised server. However, gaming in recent years appears quite different in comparison to the early 2000s and game worlds, in general, are becoming increasingly more significant. With more concurrent players in large shared spaces, scalability becomes the apparent issue. In *Battlefield 2042* (2021) , a game session can support up to 128 players on a single server (Dev Team 2021). The game *MAG* (2010) could support up to 256 players per server (Anon 2022*b*). This is an impressive feat, especially for 2010. However, there is no publicly available information regarding how this was achieved.

2.2.4 Distributed Architectures

In recent years, however, distributed server architectures have become increasingly popular (Waldo 2008). This is clear from the increased prevalence of Massively Multiplayer Online games (MMOs). Games such as *World of Warcraft*, *FFXI*, *Destiny 2*. These types of games cannot achieve a “massive” scale from single server architectures, and so evolved to use multi-server approaches(Webb et al. 2006). (Waldo 2008) makes the important point that developing online games requires a different approach than other internet applications. This is because online gaming is a part of the entertainment industry and the primary objective is that it needs to be fun. (Waldo 2008) remarks that “Latency is the enemy of fun” and this is why latency is one of the main evaluation criteria of the work.

In a broader sense, the type of architecture is causally linked to the type of game. First Person Shooter (FPS) games are notably smaller in scale, prioritising communication over complicated logic (Glinka et al. 2008). MMOs require large shared worlds to be maintained, prioritising scale and complex logic over highly accurate and real-time communications. This distinction is important as it provides some general guidelines on the overall architecture design.

Distributed server architectures can be divided into 2 general groups, multi-server and peer-to-peer. Multi-server approaches are commonly used today to achieve scale in online games (Chambers et al. 2010). Multi-server in this context means that the servers are communicating to create the virtual world experience for the client, while the servers are privately owned by that organisation.

P2P gaming architectures are inherently enticing as a concept. In theory, they offer scalability, low overhead, distribution and decentralisation at no service cost to the game developers. However, as we have already discussed, CAP limits the performance here. P2P gaming has been explored since the 1990s, (Diot & Gautier 1999) claimed the validity of scalable distributed P2P architectures to host an online game synchronised using RTP and IP multicast techniques, however, there were several issues with this work. For one, the solution was not tested rigorously enough to give conclusive results, the game application was also quite simple compared to games at that time, additionally, there was no comparison to the traditional client-server implementation. However, it was a valid step in the direction of P2P online gaming and works as a proof of concept.

2.2.5 Distribution Strategies

Distribution strategies are the building blocks of constructing distributed online games. They describe partitioning and replication of the virtual environment. Distribution strategies are abstract techniques meant to manage the load on a server or group of servers. The primary examples of such strategies include Zoning, Instancing and Sharding (Glinka et al. 2008).

Zoning

Zoning is the concept that a single virtual world 'map' is divided into specific sub-spaces named 'Zones'. Each Zone is assigned to a single server or server process. Any communications must go through the server responsible for that zone. These are effectively area-based partitions of the virtual world within the game. The zoning concept was adopted by this work to create 'streams' found within the model and implementation. This is because a single leader with multiple followers is similar is the same model offered by Kafka in topic partitions, discussed later in this chapter.

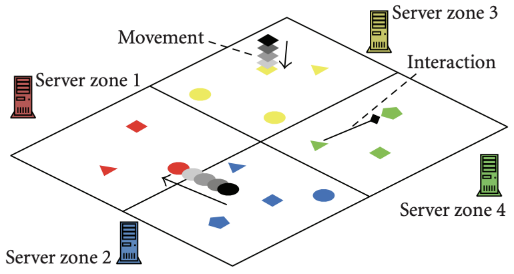

Figure 2.2: Zoning illustrated, this figure was extracted from (Glinka et al. 2008).

Instancing

Instancing (Glinka et al. 2008) is the concept of hosting multiple versions of the same virtual environment within the game, called an ‘instance’. An instance is only created for a purpose i.e. a client/player enters a specific area that needs to be separated from the global space. The instance is assigned to a server and destroyed when all clients/players leave that instance. This is a form of replication on a comparatively small scale.

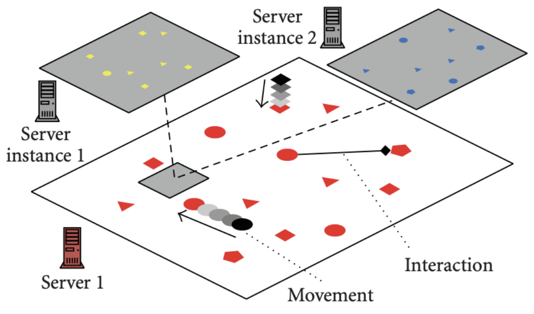

Figure 2.3: Instancing illustrated, this figure was extracted from (Glinka et al. 2008).

Sharding

Sharding (Waldo 2008), in this context, means representing the same area on multiple servers and holding each server responsible for that set of data. Data that a server is responsible for is called an entity, entities are replicated on the other servers and called shadow entities. Therefore a client can theoretically request data from one server and build an image of the game at that time. While the term ‘Sharding’ is used frequently in the field of distributed systems and distributed databases, substantial evidence points to the etymology of Ultima Online (Team 2022), where the term was used to describe replicas within the MMO game.

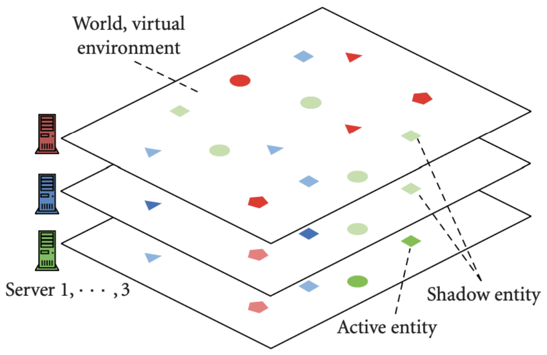

Figure 2.4: Zoning illustrated, this figure was extracted from (Glinka et al. 2008).

These orthogonal techniques can also be combined to create complex distributed models for game worlds (Waldo 2008), for example, one large map can be maintained while smaller instanced areas (interiors) can be instanced for each player. Zones can also be overlapped to allow for seamless traversal for the client. This was implemented successfully in (Waldo 2008) through the distributed system named 'Darkstar' from Sun Microsystems Laboratories. It was a privately funded research project aimed at designing "server-side infrastructure to exploit the multi-threaded, multi-core chips being produced" while horizontally scaling over many server nodes. The goal was to provide the developer with the illusion that they are developing a single-threaded application. While the Darkstar project was advanced in its multi-threading performance, it still relied on server nodes to process the state and distributed this state to clients. However, the proposed methods in this work move the state computation to the client to reduce the overall cost of such a system. Therefore, purpose-built, computationally advanced server nodes would not be necessary, especially as they might be provisioned and underutilised.

2.2.6 Compensatory Techniques

We now focus on the client itself and 'Compensatory Techniques' (Smed et al. 2001). There exist architecture agnostic techniques to reduce the amount of data required by each client to build a picture of the whole game state. Dead-Reckoning, Tick Frequency, Interest Management, Message Compression.

Dead-reckoning

Dead-Reckoning is a technique to mitigate the absence of data, such as packet loss. Originally used in aircraft navigation (Burbeck et al. 1954), it is now applied in some form to virtually every online game involving movement. If data for a necessary entity is missing on the client it will make a best-effort guess as to where the entity could be, where the data is missing. Various algorithms exist, the trivial solution is to just use the previous positions. However, some may use a combination of the previous state and linear/angular speed, with some advanced solutions performing physics simulations (Epic 2022*b*).

Tick Frequency

Online games usually involve some form of execution that occurs in a loop, each loop is referred to as a tick. Tick frequency relates to the rate at which the execution loop within the game processes updates on the server, this is measured in Hz (Glinka et al. 2008). Tick frequency, therefore, influences the rate at which the client will receive updates from the server. Increasing the ticks increases network traffic. For most competitive play 60-120Hz is the ideal range of tick frequency, while 20hz is suited for more casual experiences. It is also relevant to mention here that clients also use interpolation techniques to show entities moving smoothly through the world between ticks (Diot & Gautier 1999).

Interest Management

Interest management is a technique of managing the client's "Area of Interest" (AOI). Reducing the AOI of a client is an effort to reduce incoming network traffic (Roehl 1995). Clients express the data they are interested in, i.e. only data for the 'Zone' (Roehl 1995) or subspace they are situated in. (Glinka et al. 2008) mentioned this idea but did not exploit this compensatory technique in their work; this will be explored in our work, with an example in the implementation chapter. Pub/Sub models theoretically suit this concept of AOI, where clients can subscribe to the data they are interested in. This will further be explored in the model of this work in the section regarding streams.

Message Compression

Message Compression techniques can be employed to reduce bandwidth within the distributed architecture. This occurs by reducing the size of individual packets of data. Data compression algorithms can be used (e.g. 'gzip', 'snappy', 'lz4') to reduce packet size. (Brandt 2009) identified that compression techniques were used in the games *Quake II* and *Eve Online* and also commented that "compression is an expensive operation", and this adds protocol compression lag to the packet while it travels through the presentation layer. Another approach called 'bundling' requires multiple messages to be aggregated to reduce overall header length. Compression can be an effective solution to reducing bandwidth. However, we felt this would negatively impact the model and framework as the frequency of events is so high that any additional lag to individual event packets could be multiplicative and detrimental. This was not implemented in our work as it would require extensive evaluation and is not the primary focus of our work.

2.3 Apache Kafka

This section will act as a brief introduction to Apache Kafka to convey the important ideas necessary to understand this work.

2.3.1 Event Streaming

Kafka is an event-stream processing platform developed by Linkedin in 2011. The Kafka documentation describes event streaming as “the practice of capturing data in real-time”, forming this data into an event and “storing these events durable for later retrieval” (Authors 2022). Event streaming effectively allows continuous streaming of data which can be dealt with in any arbitrary way. Examples of data sources listed by the (Authors 2022) are databases, sensors, mobile devices and cloud services.

2.3.2 Automation

In an age where business is more software-reliant than ever before, real-time automated data processing is crucial. The users of these event streams are commonly not human users but machines and software programs. To this end, the open-source and relatively mature nature of Kafka has led to its popularity in the tech industry with Airbnb, The New York Times, Goldman Sachs and Shopify all using Kafka (Authors 2022). This popularity has resulted in numerous helpful APIs all constructed for several computer languages such as ‘Kafka-python’(Python) ‘librdkafka’(C/C++), Java and more.

2.3.3 Architecture

Kafka is a distributed system in and of itself, consisting of servers known as brokers and clients known as producers and consumers. Brokers contain distributed data and are orchestrated by a Zookeeper node. This grouping of broker and Zookeeper nodes is called a cluster. Brokers can be geographically distributed or contained within the same data centre. Each broker has a unique ID within the cluster which is used to perform identification and health checks, to identify broker outages.

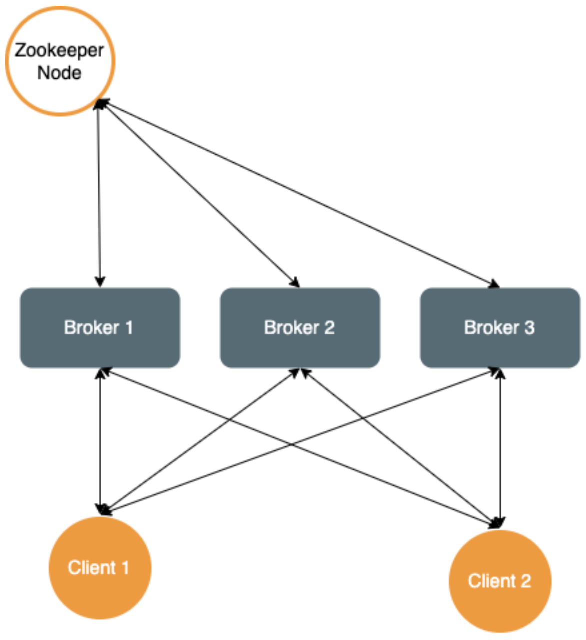

Figure 2.5: Architecture of an example Kafka cluster, with one zookeeper node, three brokers and two client nodes.

2.3.4 Kafka Terminology

Key to understanding how Kafka works lie in its terminology. This section will explore the main concepts through light explanations and illustrations.

Event

An 'Event' is the unit of data, also called a message read or written within Kafka. It is a simple key-value data object with a timestamp. It may also include optional meta-data headers.

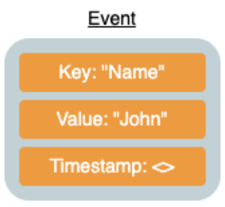

Figure 2.6: Illustration of an event in Kafka composed of key, value, and timestamp.

Producer and Consumer

Producers are clients who send (publish) events to Kafka, while Consumers are clients who read these events (subscribe). Therefore Kafka follows a Pub/Sub model. Producers and consumers are naive to each other, allowing scaling through abstraction as additional nodes can be added without a complex multicast configuration of IP addresses. Kafka also provides certain 'guarantees' such as "the ability to process events exactly-once" (Authors 2022) similar to a traditional Queue platform. Kafka is agile as it can operate in either Pub/Sub or Queue model simultaneously.

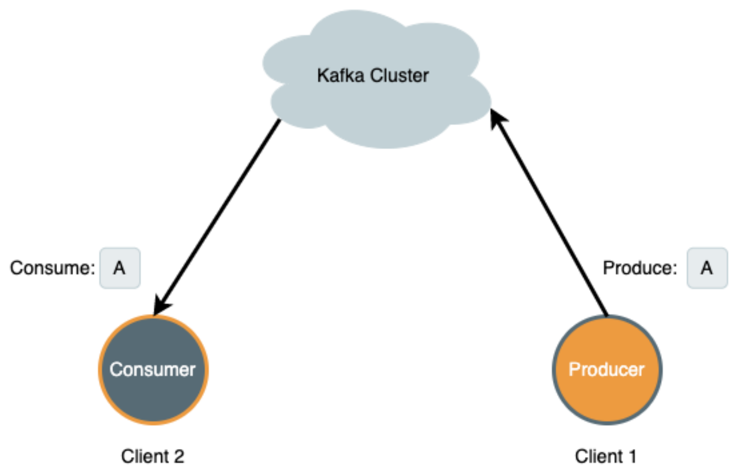

Figure 2.7: High level illustration of consumer and producer interaction with Kafka cluster.

Topics

Events are stored in 'topics'; simply put, these are groupings of events. Topics can have any number of producers or subscribers. Unlike similar messaging queues such as RabbitMQ, events are not deleted after being read. Instead, they are stored durably. The developers of Kafka claim that performance remains constant with respect to data size contained within the topic (Authors 2022). Each topic can be separately configured with a wide range of parameters such as "message retention period", which determines how long messages are kept before deletion.

Partitions

In Kafka, topics can be partitioned, meaning divided into pieces that can be stored on separate Kafka brokers. In Kafka, these pieces are called partitions. When an event is produced to a topic, it is appended to one partition of that topic. The number of partitions is described by a topic-level configuration value called "num.partitions". Kafka guarantees a total order of events for the events within one topic partition, meaning that all clients and consumers will read the same order for that topic's partition. When an event is committed to a topic in Kafka, the event within the partition is assigned an offset value. This offset value is an identifier that points to the location of that event. Offsets can be saved for each client, allowing clients to save their read position.

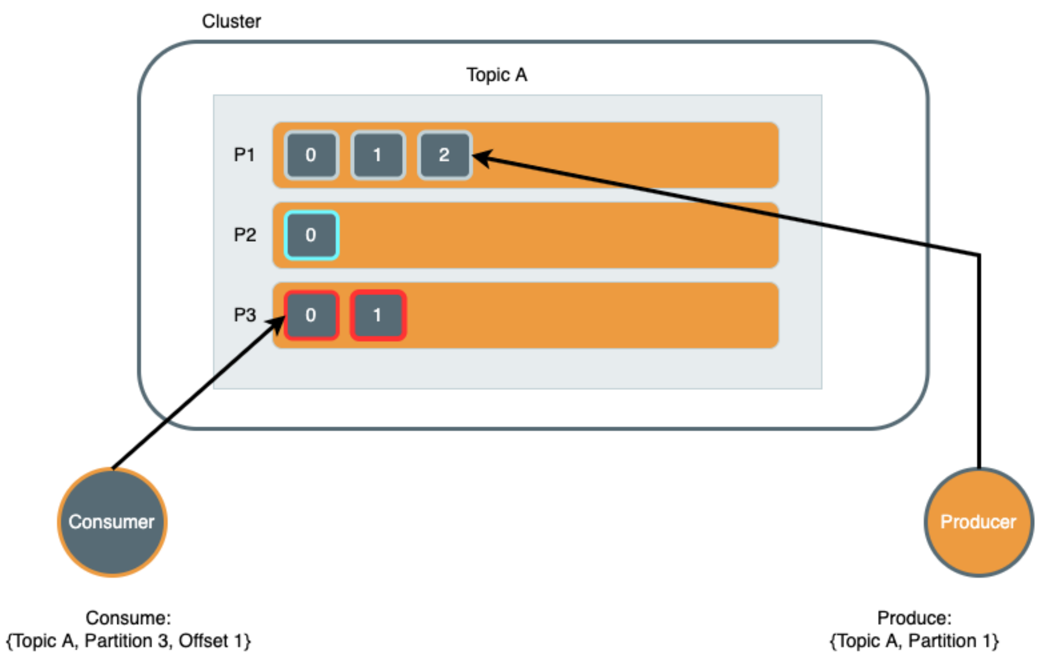

Figure 2.8: Illustration of topics, partitions, and offsets within a cluster.

Replication

Topics can also be replicated to several brokers. The unit of replication is a partition within a topic. The "replication factor" topic-level configuration defines the number of replicas for each partition within that topic. Kafka guarantees that each replica of a partition of a topic will be stored on different Kafka brokers. Replication in Kafka aims to support fail-over and availability, as, without this, a broker outage would lose all data on that broker. As several partitions would exist at once, to preserve consistency, a single partition controls read/write access to events within that partition, called the "partition leader". Events can only be read from and written to the partition leader, preserving the total order (Lamport 1978) of events.

Lastly, there is the concept of in-sync replicas (ISRs). These are replicas that are currently up to date within a certain threshold with the partition leader. This threshold is defined in the topic-level configurations. The goal of ISRs is to prevent writing to the leader of a partition if there are no valid replicas to receive these updates. Thus writes are paused until the minimum number of ISRs is reached.

2.3.5 Kafka for Online Games

User actions as well as other events in a game display similar properties to sensor readings in a distributed system implemented in Kafka. These similarities include typically small packet sizes, a mass quantity and high frequency. Additionally, online gaming requires consistency and this is already achieved in banking and e-commerce applications that Kafka already serves.

2.4 Summary

This chapter investigated the relevant areas to this work, exploring key concepts, related work, and current technology. In summary, this chapter explored consistency and consistency models pertinent to this work, online gaming and current architectures in addition to related distributed current architectures and distribution strategies. Finally, the core concepts of Apache Kafka needed to understand the methods presented in our work were provided. Our work employs these ideas as the core foundation from which to build the model and implementation of an event-driven and consistent distributed online framework and game. In the following chapter, the model for this work will be defined.

3 Model

The “Model” refers to the definitions and abstractions presented to the programmer to simplify the creation of distributed online games and similar virtual experiences in this work. It outlines a set of features implemented within the framework. This model is deliberately abstract to allow for compatibility with a range of games, designed with several archetypes of games in mind. These abstractions are provided to increase reusability. However, this does not mean it will necessarily perform well in implementation. Through using this model, a standard client-server game can be modelled using an event-driven distributed system.

3.1 Game State

A game is a state machine, given a set of inputs, for example, user actions, it stores this information and reacts based on the rules of the game. The state is used to construct a view for the player, this allows the player to understand the current state of the game.

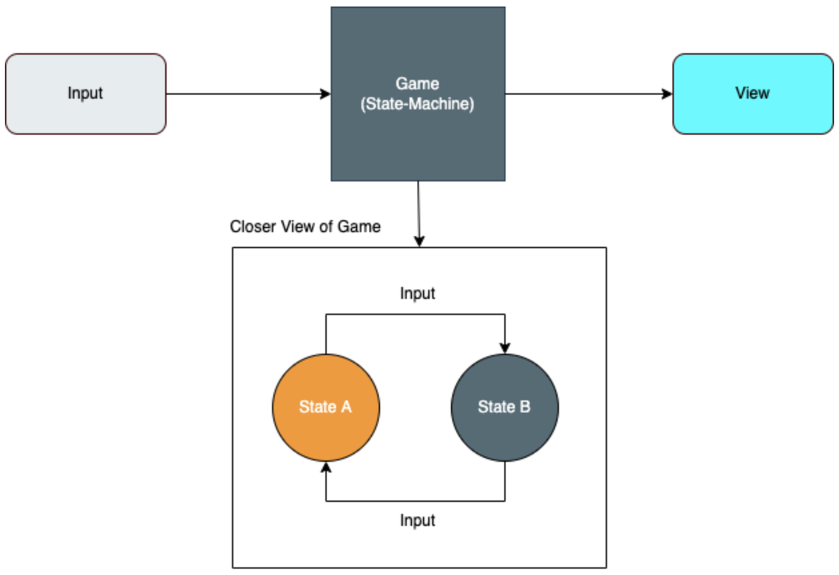

Figure 3.1: Game illustrated as a state machine with two states, a magnified view of the state machine is also displayed.

3.1.1 Local States

The state can be divided into a set of sub-states. The model will refer to these states as “Local” states. Each game will have a different approach to breaking down the game’s state into Local states, it may be based on sectioning the virtual environment such as a 2-dimensional grid, or in a poker game, the games’ state could be divided by players in the card game. Each Local state will be some piece of the Global state that contains all information relevant to that locality.

To demonstrate this concept, consider a game set in a large virtual environment. The game’s state represents the data required to rebuild that game at that point in time. This state is then divided into a set of Local states. This is achieved by dividing the virtual environment into regions and assigning a Local state to that region. Each Local state would be used to show an isolated view of the virtual environment but combined, the Local states contain the information required to show the virtual environment. This example will be carried through this section to illustrate concepts within the model.

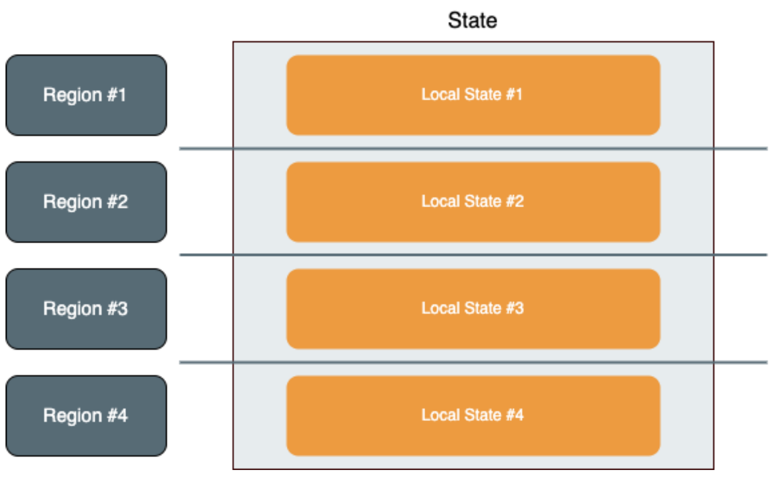

Figure 3.2: The state of the game divided into several "local" states.

3.2 Objects

Consider a Local state; this state contains the data for each “Object” within that state. An object could be a player, an enemy, the game’s score and so on. An object is essentially a grouping of data items within a state.

Consider a player object in the previous virtual environment example; as the player provides inputs to the game in real-life (i.e. presses buttons on a keyboard), the game will process these inputs, update the state, and the user will be shown a view after the effect of these changes.

3.2.1 Local Objects

Local objects are objects contained within one Local state of the game. Consider the virtual environment example again, shown in figure 3.2; if the player object is positioned in 'Region #1', then the data for this object will be stored in 'Local State #1'. A visualisation of objects stored in a Local state is shown below.

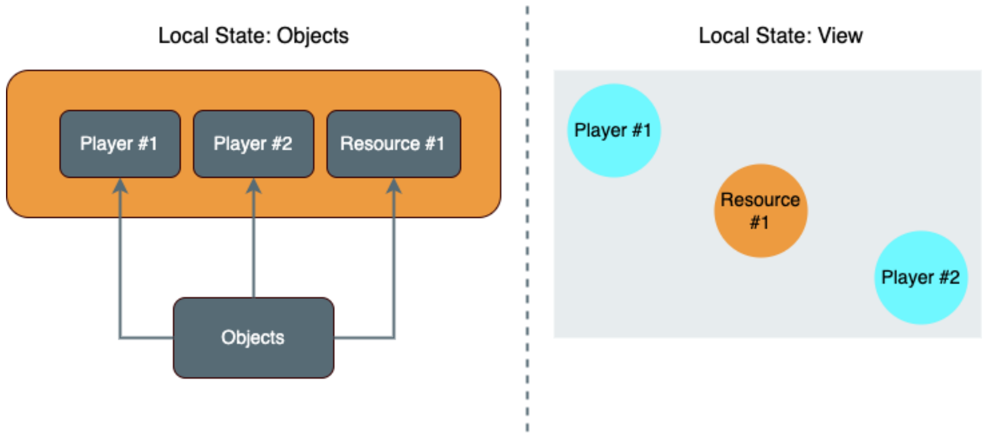

Figure 3.3: Illustration of a Local object.

3.2.2 Global Objects

A Global object is an object that is contained in all Local states. Local states are isolated from one another; therefore, they would need to mutually agree on the value of an object in specific scenarios.

To demonstrate this idea consider the virtual environment example again from figure 3.2. This time there exists a timer that, when depleted, concludes the game for every player. This timer is represented as an object within the games' state, a Global object. In the case where a player is only concerned with one Local state as opposed to all Local states, the game will not end if the timer isn't contained within that Local state. For this reason, the timer is declared a Global object contained in all Local states.

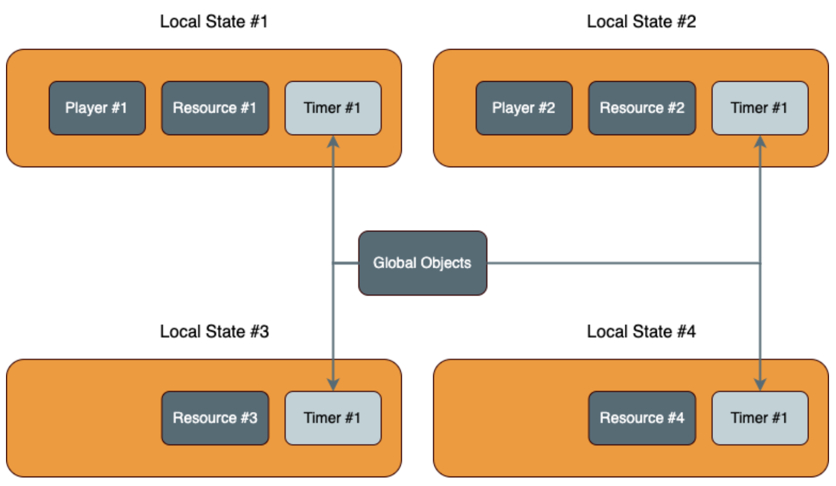

Figure 3.4: Illustration of a Global object.

3.2.3 Diffuse Objects

A Diffuse object is an object that exists in many Local states but not all. In certain games, it is important to store an object in several Local states.

Consider the virtual environment example again. In this example, the board has been divided into four distinct regions. Two regions share one weather system, whereas the other two share another weather system. In this case, each weather system object will need to be stored across the two relevant Local states. The weather system objects are Diffuse, as they exist across more than one Local state.

3.2.4 Abstraction

The words local, global and diffuse, were chosen to maintain a distinction between state and geographical virtual environment. For example, a card game does not usually have a virtual environment where some games do. This distinction is important as it allows a more abstract approach to defining states and objects.

3.2.5 Critical vs. Non-critical

While an object can belong to a state, each object is also defined as critical or non-critical. A critical object can be owned by a non-critical object. This is necessary as it allows for modelling a “critical resource”, an item of value for which there is a limited number, such as an item within a game that only one player can own. This defines a relationship between objects in the context of the game.

For example, consider a player as a non-critical object and a coin as a critical object. The player can own a coin, but the coin cannot own a player, illustrated in figure 3.5.

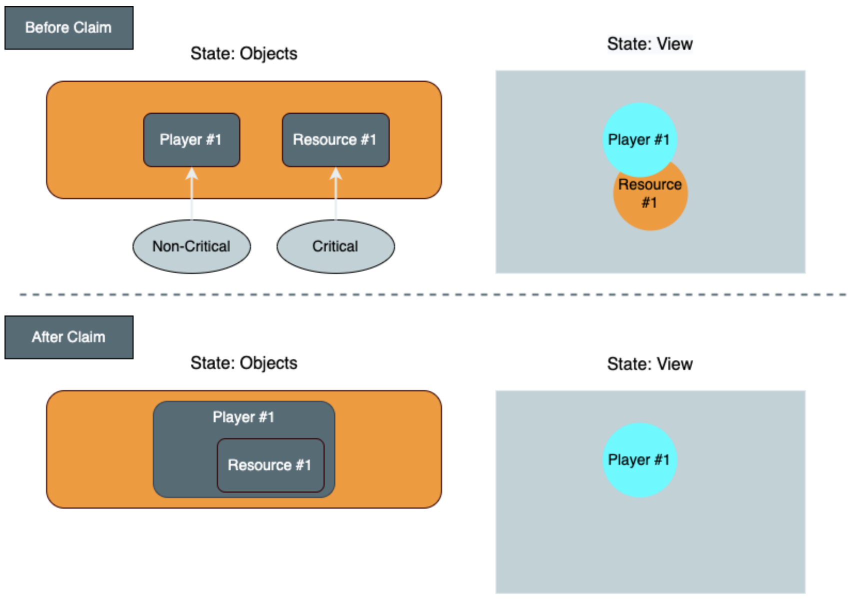

Figure 3.5: Illustration of Critical and Non-critical objects.

3.3 State Transfer

Online gaming requires some method to transfer state from one game client to another game client. This is achieved through event-based state transfer, which, in this model, is composed of a message protocol, an event-driven backbone and event streams.

3.3.1 Message Protocol

The model uses messages to update each object within the state of the game. This model refers to these messages as "events". Events obey a standard protocol that defines the type of event and objects to which the event references. Two event types are defined in the model, and these will be described here. As the model is designed to be reusable, more event types can be added as long as they provide purpose and benefit. Events can be divided into two general types.

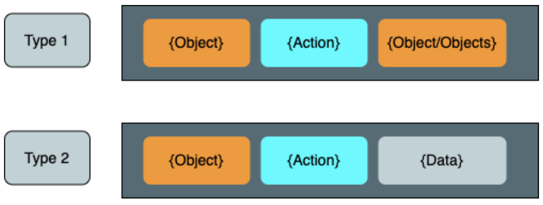

Figure 3.6: Illustration of message protocol.

Type 1: {Object}_{Action}_{Object/Objects}

In this event type, an object performs an action on another object or group of objects. An example of this will be if a player avatar (object) claims (action) a coin (object).

Type 2: {Object}_{Action}_{Data}

In this event type, an object performs an action with some data. An example of this would be if a player (object) were to move (action) to another position (data). These events should be strictly idempotent when sent to a low-priority stream. Priority streams are defined in the following sections.

3.3.2 Event-driven Backbone

In this model, once an event is constructed, we require a method to transfer these events to another game client running in another system. We also require the ability to transfer these events to many other game clients as we intend to scale up the number of concurrent players. The solution chosen to solve this problem is to use an event-driven backbone. Events should be sent to the backbone and be retrieved by other systems. The reason for this guarantees that events will be sequenced. The detail of this sequencing will be covered in the next section.

Consider the case of one game client producing events to the backbone, with many game client retrieving these events. As events are retrieved from the backbone, each game client should be able to build a state from these events that are consistent with the other states. This is because events are sequenced in the backbone.

Consider the more common case of many game clients producing events, with all game worlds consuming these events. Each game client should build a state from these events consistent with other game client. Sequencing is crucial for consistency in this case, as concurrent writes will need to be resolved within the backbone.

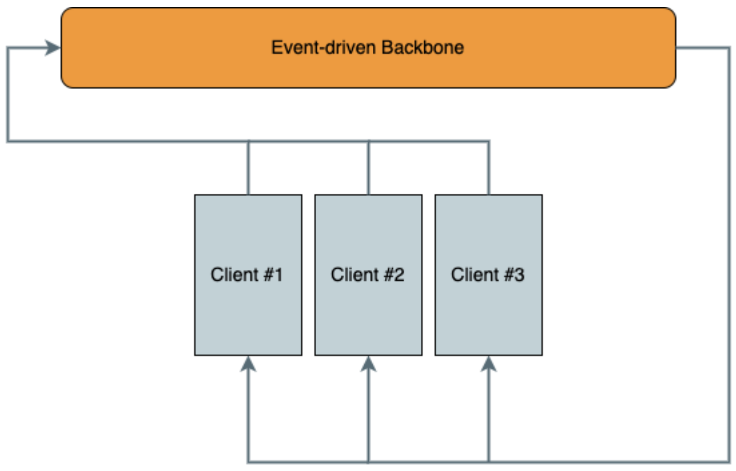

Figure 3.7: Illustration of multiple game clients producing events to the backbone.

3.3.3 Event Streams

In order for the game world in a system to transfer events to the backbone, we require some stream of communication. This model provides two streams of communication, defined by priority. According to the model, when a game world builds a state from sequenced events, it should be consistent with other games' states

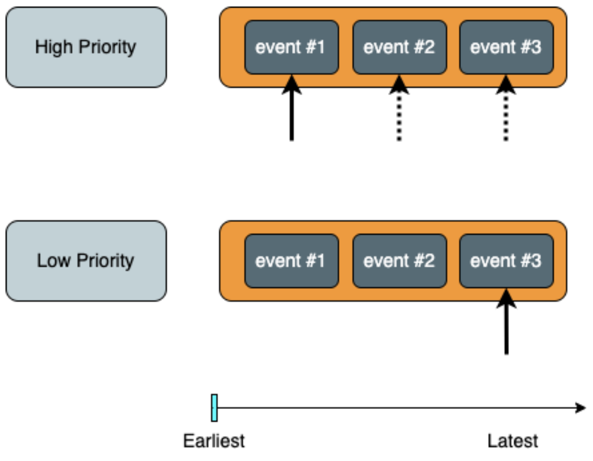

Figure 3.8: Illustration of priority streams.

High Priority

This stream should be used when it is vital that all events in the stream be read from beginning to end to build the correct state. This stream provides sequential consistency.

To illustrate the example use case for the high priority stream, consider a game world with two players and one coin, which acts as the critical object. If both players attempt to claim the coin, they will construct an event message and send this to the high priority stream. Then, as both players read from the sequenced stream from beginning to end, both states can agree that one of them claimed the resource first. If both players instead used the low priority stream, then both players would read the sequenced events but from different starting positions in the sequence.

Low Priority

This stream should be used when the stream can be read from any point to build the correct state. This stream provides consistency similar to eventual consistency. Events sent to this stream must be idempotent as systems may not have prior object states to mutate.

To illustrate the use case of the low priority stream, consider a game world with two players, each updating their respective positions. These players are non-critical objects.

Therefore as long as systems receive the latest positions of these objects, the state machine that is the game itself can build a consistent state, as it does not require historical events for these objects. This is not the case for critical objects.

3.4 Partitioning

The concept of partitioning was previously explored by dividing the state of the game into multiple Local states. However, this method of partitioning is directly related to streams and as streams have just been introduced it is relevant to mention its relationship with partitions.

The goal of partitioning the state into Local states is to increase the availability of the backbone as there are more backbone nodes to serve requests for events. Partitioning correctly will help protect performance at scale by increasing the throughput. The price of partitioning in this way means that there exists no total order between all streams. However because each Local state contains all information required to build a consistent view of that state, we have a total order where it is necessary.

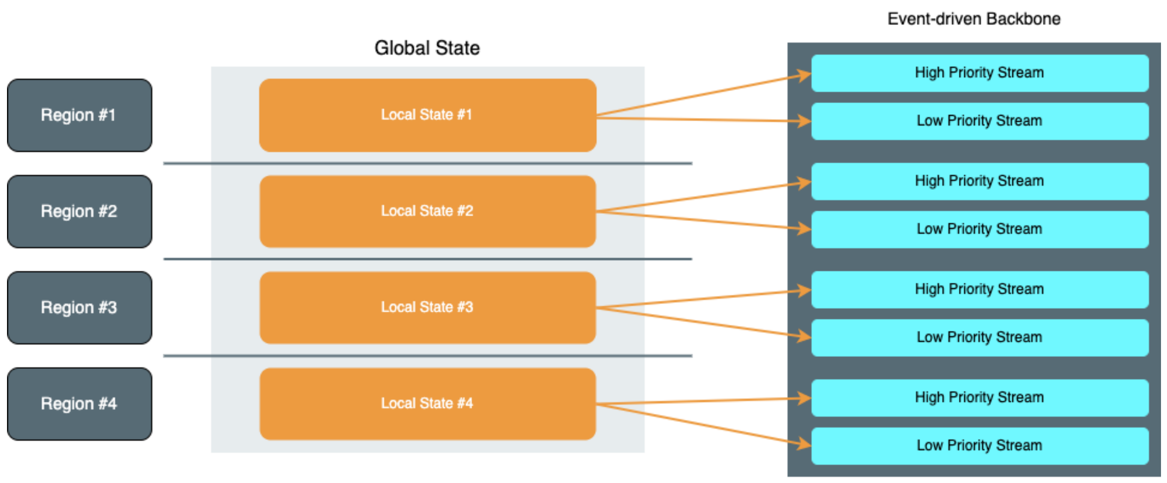

Figure 3.9: Illustration of example partitioning strategy for figure 3.2.

Figure 3.9 demonstrates how the partitioning strategy of figure 3.2 is related to the event-driven backbone. In this example, the virtual environment is partitioned by region with each partition assigned a low and high priority stream. Two priority streams are necessary to allow for different read positioning. While this serves as an example, partitioning can be implemented in a number of arbitrary ways and does not equate to a region or even a local state.

3.5 Consistency

This work hypothesises that total order is not necessary across all game events. This hypothesis is exploited to achieve a total order only where it is necessary to do so. As such, this model allows for event ordering per object or for any number of objects through using streams. As mentioned previously, the backbone should be assumed replicated. It is kept strongly consistent (detailed in the implementation chapter). Therefore, the consistency for each game client is sequentially consistent as each client will asynchronously poll the strongly consistent backbone for new events.

3.6 Area of Interest

In certain scenarios of games, game clients do not require the global state at all times. Consider an expansive virtual environment where each partition is so large in area that a player can't view outside of the partition boundary. In this large-scale world, the player is only concerned with the local state. To this end, in this example, the system will only need to retrieve the streams for the partition they are "interested" in. In addition to this, systems can also retrieve streams for multiple partitions if they require a consistent view for more than one partition.

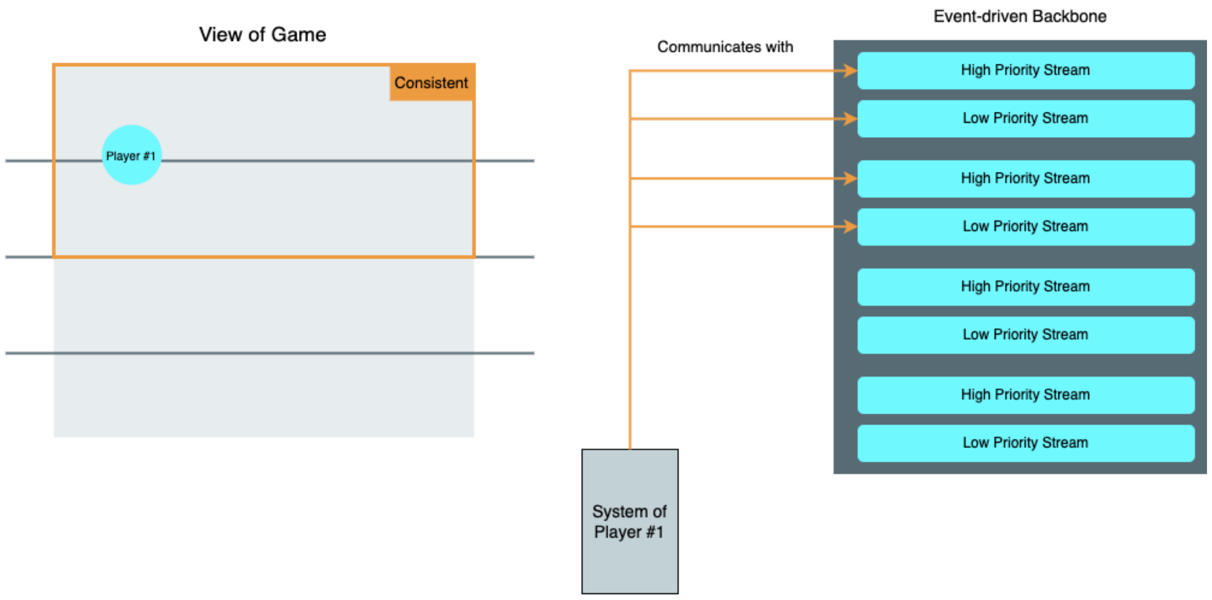

Figure 3.10: Illustration of area of interest technique.

The benefit of this area of interest design is that game clients only retrieve required events. This works to reduce the network and computation load on each system, as each system could potentially retrieve fewer unnecessary events. This can reduce network and computation load on the backbone by sending fewer unnecessary events.

3.7 Archetypes

The archetype defines the configuration of the model to fit the game. Each game is created differently and with a different set of requirements (Brandt 2009); as such, it will require a tailored approach. The framework is designed so that it is highly adaptable, allowing for alternations through abstractions. These abstractions aim to provide a programming model which can be applied to a wide variety of games.

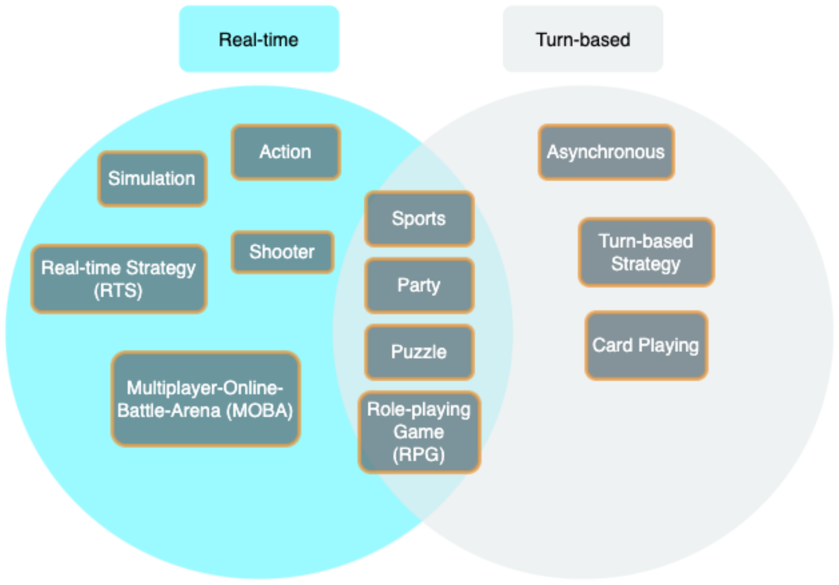

Figure 3.11: Illustration of game genres.

When tailoring the model to fit various game types, the first consideration must be the requirements of that game. To understand those requirements, games can be separated into two categories; real-time or turn-based, fig 3.11 provides an example of classification. Real-time games require a higher performance than turn-based games, as described by (BSA 2022). It is also mentioned that a real-time first-person shooter (FPS) requires a latency of between 5 and 60ms to be considered competitive, whereas a turn-based card game such as 'Hearthstone' can still be playable at latencies over 100ms. These requirements must be reflected in the model's configuration. Each game will have a different requirement, as was seen in (Brandt 2009). The main configurations of the model will be the tradeoff between performance and availability as seen in the CAP theorem (Brewer 2000), with proof (Gilbert & Lynch 2002). The configuration is specific to the backbone, as ncreasing the number of in-sync-replicas (ISRs) within the backbone also increases the latency in producing events which will have an impact on all players. Higher ISRs should always be preferred where performance can be traded for availability.

4 Implementation

This chapter details each facet of the implementation of this work. Firstly, a high-level overview of the Framework will be discussed. Then, the process through which the example game was represented using the model will be detailed. Finally, the development of both the framework API and game will be explained.

4.1 Framework

In the previous chapter, the Model was defined as a set of abstractions to build a distributed game. This section will explore how that model was applied to a reusable framework.

4.1.1 Event-driven Backbone

The core of the Model is the event-driven backbone. This component handles the event ingestion, sequencing and replication for the game. Two software products were explored to perform this task; RabbitMQ and Apache Kafka., though both are not viable options. Apache Kafka was chosen for several reasons. It operates on the dumb broker/smart consumer model meaning that messages are polled by the consumer (game) when appropriate. RabbitMQ operates inversely to this, pushing messages to the consumer process. A failure scenario would lead to unprocessed or out of sequence messages. Kafka is built with the idea of reading any event at any point in the event queue, whereas RabbitMQ is not. This means Kafka is much more appropriate to implement the idea of "Priority Streams" mentioned in the Model, as reading any point in the event queue is critical.

4.1.2 Framework API

In order to separate the game code from the framework, an API assists with accessing the backbone. This API handles the Kafka consumer, producer and helper functions as well as containing configuration details and other relevant Kafka states. The details of this API will be specified later in this chapter.

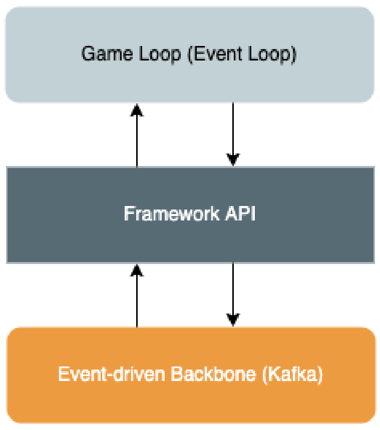

Figure 4.1: Framework API described to connect the game with the event-driven backbone.

4.1.3 Event Loop

The event loop also referred to as the "game loop" demonstrates the logical separation of the game itself from the framework and backbone. The game's code interfaces with the framework, and the framework orchestrates the backbone.

The concept of Objects was explored in the Model; these are implemented within the game as standard programming objects. To update an object in the shared state of the game, several steps need to occur in a specific sequence. Firstly, an input within the game creates an event. However, before this update can happen, it needs to be sequenced within the backbone for consistency. Therefore, if an object is requested to be updated, it must first access the framework API. This API maps the attempted update into an event that the Kafka producer will produce to the backbone. This ensures to a high degree that the event will be sequenced and stored in the queue. The game code will then interface with the framework to access the Kafka consumer. The consumer will consume events in a sequence that can be applied to the game. This whole process is analogous to the idea of a '2-Phase Commit'.

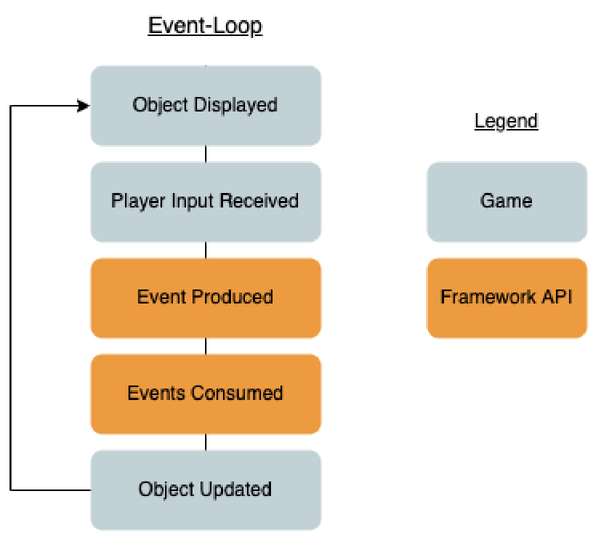

Figure 4.2: Illustration of event loop.

4.1.4 State Transfer

The Model specifies that state transfer occurs through an event-driven message protocol. For the above event loop to occur, attempted updates to objects are translated by the framework API to standardised events in a binary string. Events are then passed to the Kafka producer to transfer events to the backbone, the Kafka cluster.

Events are constructed by the specification in the Model chapter. Type 1 events are created to signify one object interacting with another. Type 2 events represent one object interacting with supplied data. It is hypothesised by this work that this abstraction is sufficient to model all updates within all games.

4.1.5 Priority Streams

The Model presents the abstract concept of streams which are then implemented in Kafka. Streams are implemented as Topics of one partition in Kafka. High priority streams are read by the consumer from the earliest offset, where Low priority streams are read from the latest offset. These offsets are committed to the Kafka cluster by the framework to continue from the last read offset for each stream.

4.1.6 Partitioning

Partitioning, as specified in the Model, pertains to the idea of separating the global state into several local states, and we will now explain how this is achieved within the framework. In the framework, the world of the game is divided into sections called ‘Regions’; each ‘Region’ is granted a high priority stream and low priority stream, implemented as separate topics. To create consistency for global objects, these events are sequenced to a high priority stream that is consumed by all game clients, regardless of Region. For diffuse objects these events are produced to a high priority stream that is consumed by game clients whose players are concerned with that region.

4.1.7 Replication

Priority Streams are replicated based on a parameter named replication factor. This is specified during topic configuration by the framework and different values can be supplied. In normal operation, two to three replicas are necessary. In addition, Kafka implements the idea of In-sync-replicas (ISR). The number of which should also be considered when analysing the performance requirements of the game. Increased replicas and ISRs will result in increased latency for each commit.

4.1.8 Architecture

Depending on the chosen deployment, the framework can be deployed as either a cluster within a data centre or distributed cluster, with each user running a Kafka broker node in that cluster. This choice of deployment is independent of the game's code or framework API.

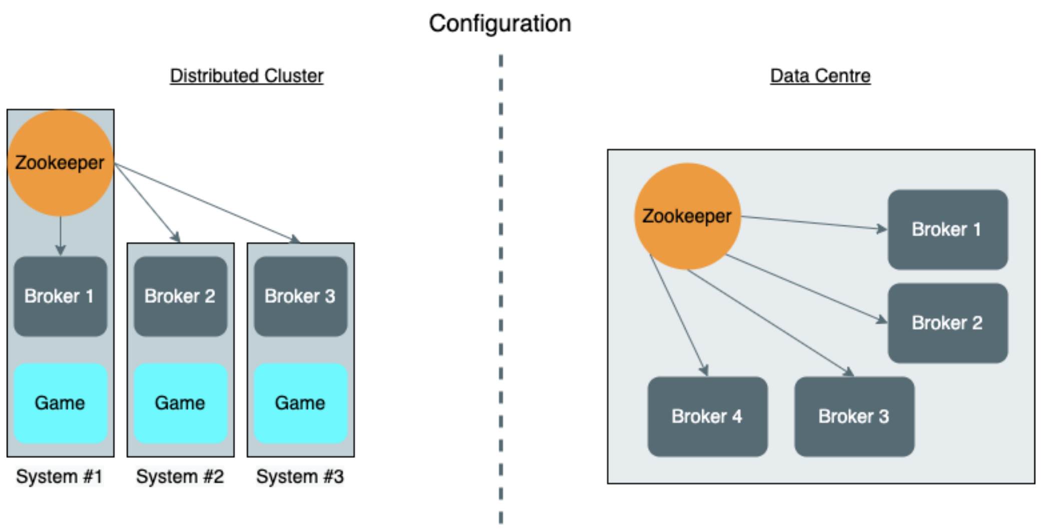

Figure 4.3: Illustration of two backbone deployment options, using players to host nodes (left), and using cloud services provider to host nodes (right).

As the framework relies on Kafka, one or more zookeeper nodes will be required to orchestrate the distributed system. This will either be run by a Kafka broker node in the data centre configuration or at least one player in the distributed cluster configuration.

4.2 Application of Model

This section reviews the example game, its requirements, as well as how it can be represented using the Model.

4.2.1 The Game

To demonstrate the process of building a distributed game, a toy example was created. This game served for development and evaluation purposes and was not intended to be elaborate. The game concept was designed to support a large number of players in real-time. In this game, users would input commands to move their avatars around a fully observable open world. In this virtual space, there is a limited quantity of coin objects; these served as critical resources. Players collect coins, and once they are all collected, the game is over, with the winner or winners having the most coins. With respect solely to design, this game supports an unlimited number of players. The performance of this scale will be evaluated in the following chapter.

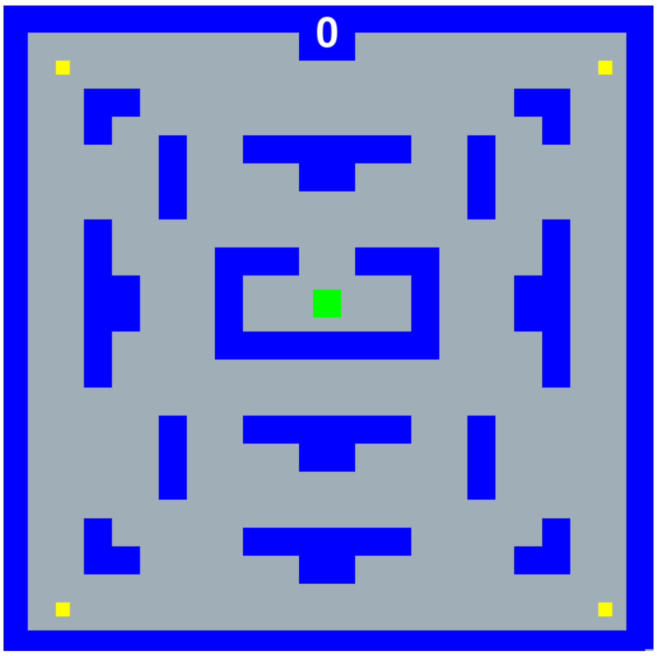

Figure 4.4: Illustration of the chosen game for implementation.

4.2.2 The Requirements

Before development, the game was categorised and its requirements defined. The game is meant to be played in real-time; it is competitive and scales to a large number of players. With this in mind, we can derive the functional requirements. Real-time suggests that the user expects a high degree of performance. From the user's perspective, they should not experience a high degree of latency concerning their actions taking effect in the game. More specifically, the time taken to process the player input event, commit it within the Kafka cluster and be read from the sequenced events should not exceed the threshold within the game considered to be noticeable (see noticeable latency as described in the section section 2.2.2 and figure 2.1). This threshold is chosen to be 60ms as described in (BSA 2022) as the threshold for competitiveness in games. In addition to this, the player's input should reliably reach the cluster to be sequenced. The game is competitive and so is required to be consistent with the true state of the game to allow for fairness. The game should be able to support a large number of players within the same game session. The implementation will be evaluated in the following chapter with respect to these functional requirements.

In short, the game must display low latency (under 60ms), reliable state transfer, consistent view between systems, and scale to a large number of players.

4.2.3 The State

Following the model's standard process, the state was partitioned into regions with local states. The game space was horizontally sectioned with initially one partition for every player. This number represented a base standard for availability within this framework and was selected for the initial implementation. With the state partitioned, the objects were defined. One player object was required for each player to act as their avatar; several coins were needed to represent the critical resources. The players are defined as non-critical objects; the coins are considered critical objects.

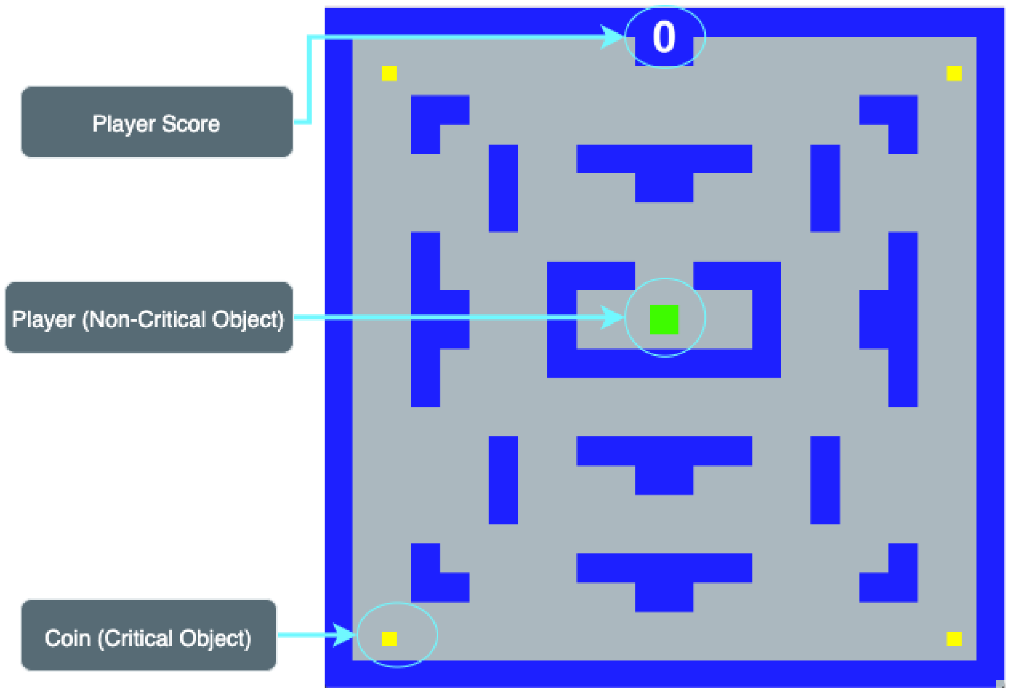

Figure 4.5: Illustration of the game state.

4.2.4 State Transfer

In this implementation, type 1 and type 2 event messaging protocols were used to transfer states. Type 1 represented the events when a non-critical player object would lay claim to a critical coin object. These event types used a “Claim” action to identify claiming the resource. These type 1 events were sent to the region’s high priority stream. Type 2 events represented the idempotent updates to non-critical objects, such as players updating their positions around the virtual environment. The action for these events was “Move”, communicating the objective. The “Data” was an x,y world-space coordinate. These events were transferred through the low-priority stream.

4.2.5 Partitioning and Replication

As was previously mentioned, initially, one partition per player was decided. This represented the baseline assumption that one partition per player was sufficient. This is evaluated in the following chapter. Concerning replication, a replication factor of two was chosen, with a minimum in-sync replica (ISR) value of two. This means that the specified replicas are kept identically in line with the leader, allowing for failover. Ideally, a replication factor of three would be selected. However, this was chosen as a compromise between performance and availability and reliability.

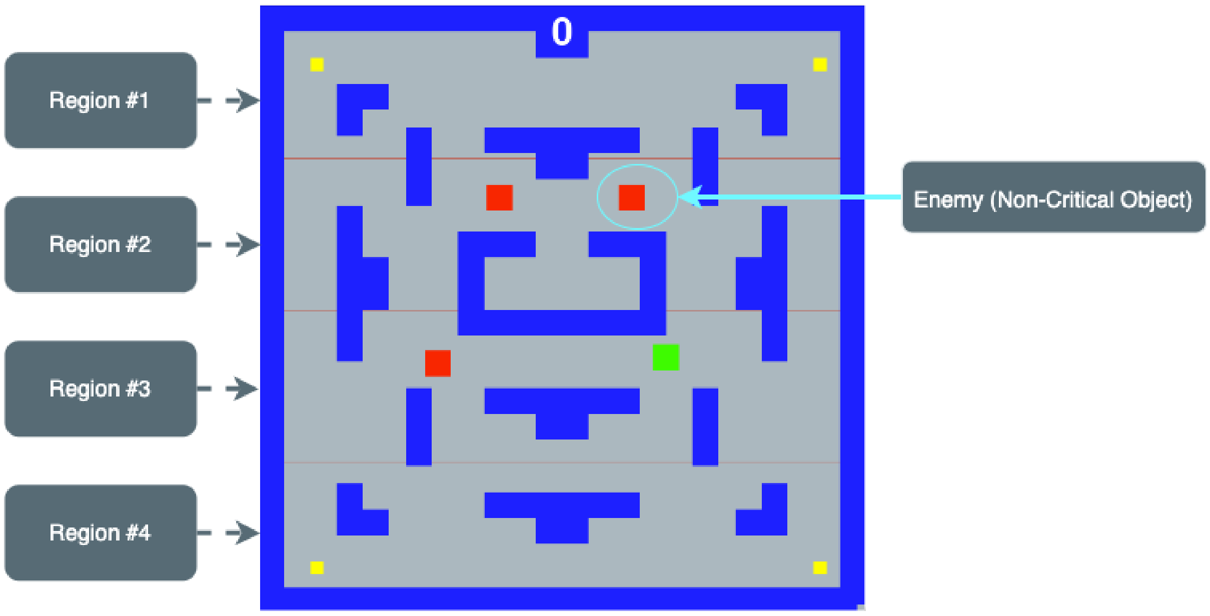

Figure 4.6: Illustration of how partitioning was implemented via horizontal sectioning.

4.2.6 Consistency

To explain how consistency was achieved in this game, consider the model, consistency is separated between the event-driven backbone (Kafka) and the states in each user’s system. As the minimum number of ISRs is two, an event must be replicated in two replicas before a consumer can read it. This is the “Commit” process within Kafka. This implies strong consistency within the backbone under this configuration.

However, the consistency process for the state contained in each user’s system is different. These state updates are polled asynchronously in relation to other systems between frames. While this is not strong consistency, it is sufficient as long as the game is updated between frames of the game-play itself. Sequential consistency is implemented for this purpose. As the backbone is kept strongly consistent, this guarantees event sequencing; a total order of events is achieved for each stream. High priority streams are read identically for all systems. Low priority streams are read from different points. However, because the updates to these objects are idempotent and read in the same total order, as long as the user’s systems are kept up to date with the backbone, their state is considered consistent. In addition to this, if a system cannot be kept up to date with the backbone, that user cannot produce events to the backbone Kafka cluster, as this would act as a detriment to the consistency system.

4.2.7 Area of Interest

This game is operated in two modes for demonstration. The first mode considers the game fully observable, and the area of interest (AOI) is large enough to show the entire world-space. For example, this means the entire board. The second mode assumes the board to be so large that it is not necessary to maintain the state for the entire world-space, the board. In this second mode, an area of interest is used to limit the number of events consumed by each user. This aims to reduce the time spent processing events that are not useful and provide not benefit to the client. This is a strategy to allow for scalable online games.

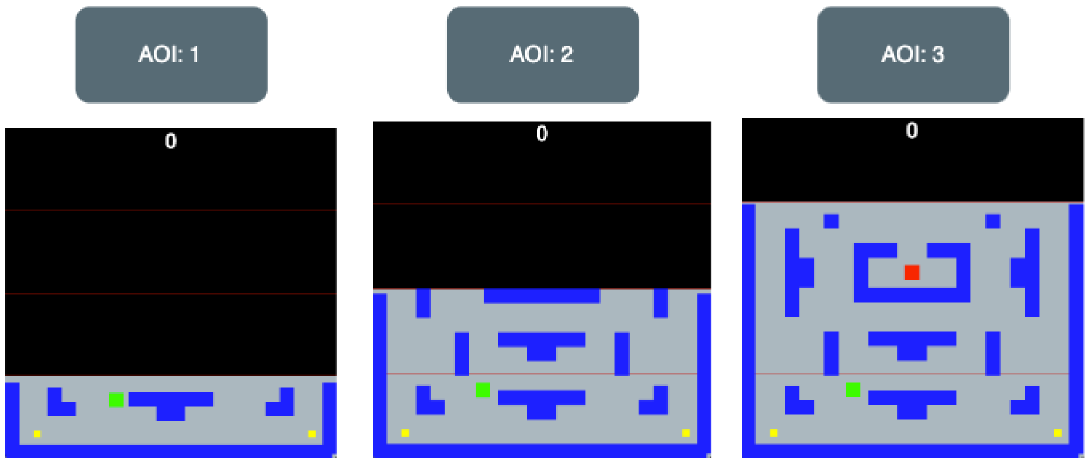

Figure 4.7: Illustration of the area of interest (AOI) implementation.

4.3 Development Process

The framework was used to build a distributed, online game to demonstrate the development process and accessibility, which is defined with respect to cost and simplicity. The development process will be detailed in this section. First, the environment will be detailed, then the framework implementation and finally, the game implementation.

4.3.1 The Environment

The code for this project was written in Python which permits rapid development. This was essential for the design and iteration process. An example game was created using the open-source python package "Pygame" (Authors 2021). This high-level graphics library builds on top of "SDL" and "OpenGL" to provide a simple graphical interface. The python module "Kafka-python" (Powers & Arthur 2016) was used in the framework code to create producers and consumers to access the Kafka cluster.

4.3.2 The Framework API

The high-level framework was specified at the start of this chapter. This section will detail how that was realised.

The API was implemented through a class named "Network Manager". This class contains all data and methods regarding interfacing with the backbone Kafka cluster. When an object is instantiated from this class, it initialises a producer and consumer. Initially, the producer will produce to the current region that the user is within. Similarly, the consumer will consume from the topics for that specified region. Currently, only one algorithm of partitioning is supported, this is the horizontal sectioning approach described in the model chapter. The Kafka configurations are fetched from a supplied configuration file.

The following public methods will be described below:

Form Event Item

```python
def form_event_item(self, action, target, low_priority=True):
    # Creates a type 1 or type 2 event string
    self.produce_event_item('{}_{}_{}'.format(self.PID, action, target)
    , low_priority)
    logging.info('{}_{}_{}'.format(self.PID, action, target))
```

This method will form an event based on the event types specified in the model. The first object in both type 1 and type 2 events is the player avatar of the system. The action and target object are specified in the method parameters. This method formats the event item string passed to the "produce_event_item" method.

Produce Event Item

```python
def produce_event_item(self, event_item, low_priority=True):
    #Forwards the event item to the relative stream
    payload = str.encode(event_item)
    if low_priority:
        if self.cache == event_item:
            return
        logging.info('producing: '+ str(event_item) + ' ' + str(self.prodTopic))
        self.producer.send(self.prodTopic,payload)
        self.cache = event_item
    else:
        if self.cache_crit == event_item:
            return
        logging.info('producing: '+ str(event_item) + ' ' + str(self.prodTopicCritical))
        self.producer.send(self.prodTopicCritical,payload)
        self.cache_crit = event_item
```

This method will contact the producer contained within the class to transmit the given event item to the cluster. Depending on the value of "low_priority", the event will be sent to the respective priority stream.

Update Events

```python
def update_events(self):
    events = []
    for msg in self.consumer:
        e = msg.value.decode("utf-8")
        events.append(e)
    self.events = events
```

This method contacts the assigned topics (streams) and polls for new events. The events are stored within the Network Managers' events list property. The events list is cleared each time this is called. The events list is made accessible to the game code for custom processing.

Update Streams

```python
def update_topic(self, player, override=False):
    topicNumber = int((player.object.ypos + player.object.height/2) // self.regionSize)
    if KAFKA_TOPIC.format(topicNumber) != self.prodTopic or override ==
    True:
        #sends mov message to current topic, before switching
        self.produce_event_item('{}_{}_{}'.format(self.PID, 'mov',
topicNumber))
        logging.info(f'switching topic: {self.prodTopic} :{str(
topicNumber)}')
        self.prodTopic = KAFKA_TOPIC.format(topicNumber)
        self.prodTopicCritical = KAFKA_CRITICAL_TOPIC.format(
topicNumber)
        self.consTopic = topicNumber
        self.assign_topic()
        logging.info('switched to topic {}'.format(self.prodTopic))
        player.setRegion(topicNumber)
```

Given the player avatar's position in the world space, this method checks which region the avatar is in and assign the relevant topics (streams).

4.3.3 The Game

The example game was represented using the model and constructed using the framework API. It was composed of the initial setup and a game loop.

Initialisation

To initialise the game, several steps were required. Firstly, the relevant Pygame objects were instantiated, and the module itself was initialised. Next, An argument parser was created to supply command-line arguments to the game at run-time, such as running the game in "quiet" mode, meaning nothing would be displayed. This was useful for testing and evaluation. A "Game Screen" class was created to encapsulate data and methods relevant to the game's display.

Furthermore, the "Network Manager" mentioned above was initialised, the world space was defined, and the walls were created from a supplied map file. The final step in the initialisation process was the creation of objects as specified in the model. These were defined as classes, with the "Player" class representing the non-critical user from the perspective of the game client and the "Enemies" class representing all other non-critical users. The "Item" class represented the critical coin objects within the game.

Game Loop

The "game loop" represents a continuous main thread of the game. Each loop begins by drawing and displaying all objects and static entities (like walls) of the world of the game world. Then the game checks for user input and handles each input accordingly. If

Object Name	Avg. Description	Type	Critical Status
Player	Client's avatar	Local	Non-critical
Enemy	Other client avatars	Local	Non-critical
Coin	collectable reward	Global	Critical

Table 4.1: Description of 'Objects' created in this game.

the user inputs one of the pre-defined directional keys, such as up, down, left, and right the game captures these events. These events are then formatted and sent to the cluster for sequencing. The loop then calls the "update_events" method within the Network Manager object, fetching and storing new events from the cluster. These events are returned "in sequence" for the game to perform the relevant logic required, such as the player claiming an object or the player updating their position in the world.

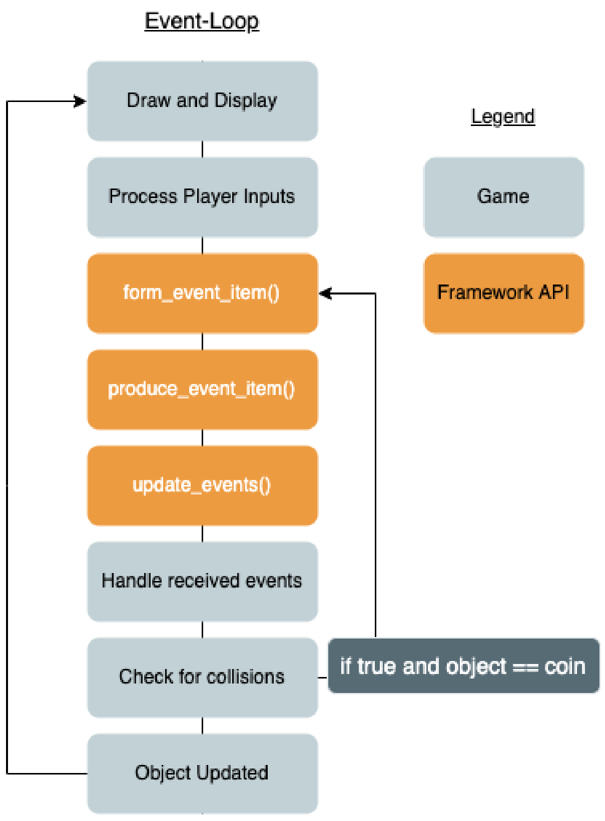

Figure 4.8: Illustration of the event loop in final game code.

Continuing the game loop, the player's position is checked, and if it is determined that the player has moved to another region (partition), the relevant topics (streams). Next, the collisions between game objects are calculated, and if a player collides with a critical object such as a coin, that player produces an event to the high priority stream to claim it. Finally, a check is performed to find if the game is over. This is done by checking for any remaining coin objects; all are collected then the game is over. Suppose the game is not yet complete; this loop repeats. A single iteration of the game loop is referred to as a “frame”.

5 Evaluation

This chapter describes the evaluation process and evaluation of the contributions of this work. As such, it is divided into three sections for each contribution: model, framework, and implementation of the distributed, online game. Within these sections, a combination of qualitative and quantitative analysis is performed.

5.1 Model

This section evaluates the design of the model. The model is evaluated by its accessibility and reusability which are more suited to qualitative investigation. Quantitative evaluation of the model was not completed in this work.

5.1.1 Qualitative

Accessibility

The objective of designing the model was to provide clear and simplified abstractions for creating a distributed online game. Distributed systems are difficult to build correctly because building these systems is an involved, multi-step process that involves complex theory and expert-level knowledge. For this reason, it is considered in this study that building distributed systems to power distributed games is inaccessible to those outside of this field, namely game developers. A comparatively simpler and standard process is presented by building abstractions on top of an event-driven backbone such as Apache Kafka.

To the user of the model, a standard process is used to design the game before implementation in the framework. The state is identified, partitioned accordingly and objects are defined within this state. From that point, the message protocol is established and the user is presented with priority streams without having to understand the inner workings of Apache Kafka. Events within a game represent updates for objects within the game state. This work hypothesises that total order is not necessary across all game events. This hypothesis is exploited to achieve a total order only where it is necessary to do so. As such, this model allows for event ordering per object or for any number of objects through using streams. This model standardises the process of designing for scale and presents the user with useful abstractions through an already considered and proven model.

Accessibility in this model also refers to the low cost of entry. This cost is specified relative to time and financial cost. By using a standard process, many common issues in designing distributed online games are already addressed, and by this rationale, time is potentially saved. By extension, this aims to reduce the cost required to hire for this design and development process. By using open-source software, there is no direct financial cost.

Reusability

The model was designed to abstract the details of distributed systems by considering multiple games and requirements in the design process. While the implemented game had influenced the model, other games with different archetypes were also considered. To demonstrate this idea, consider partitioning in the model; in the base example, it is implemented by dividing the board into equal sections and then using priority streams for those partitions; however, the board can be partitioned by any approach. In addition to this, consider the event messaging protocol; any arbitrary action can be specified to any object or data relevant. This allows for useful abstraction that allows the model to be applied to a large sample of games. However, while the model may be used for a large number of games, the implementation may limit its performance.

5.2 Framework

The framework refers to the software implementation of the ideas and abstractions present in the model. The architecture is formed using Apache Kafka, and the user is provided access to this architecture through an API. This framework is evaluated based on its accessibility and reusability. Limitations of this framework will also be mentioned.

5.2.1 Qualitative

Accessibility

The framework is tightly coupled with the concepts presented in the model, it is considered similarly accessible. In addition to those points, the framework API was implemented in Python, which according to a stack overflow survey Anon (2021), is the 3rd most popular programming language. There are limitations to using python, which will be discussed in the limitations section. The API successfully abstracted the communication with Kafka so that the developer does not directly interface with it. This streamlines the implementation post the model and design stage, simplifying the overall development process.

Reusability

The abstractions presented in the model were implemented in the framework, and for these same reasons, the framework is designed to be reusable. However, there are limitations, and these are addressed below.

Limitations

'Python' was chosen to implement the framework for various reasons, such as its popularity Anon (2021) and ease of use as it is a high-level programming language as well as availability of high-level graphics libraries like 'Pygame'. This helped greatly during the ideation stage of this work. However, Python is limited by its performance compared to other languages such as 'C++' and 'Java'Tamimi (2021) because it is a higher-level language. Due to the performance overhead of Python, it is not widely used to create games Carpenter (2022). Unreal Engine, a popular game engine used to make 'Fornite' Epic (2022*b*) uses exposes C++ for its game scripting. Therefore, the framework would be implemented in a more popular game-development language like C++ in future work.

5.2.2 Quantitative

The framework was evaluated based on Scalability and Performance. Specifically, the capacity of a single stream was evaluated to identify the cost of processing an event through the backbone. Empirical data was gathered through experiments.

Experimental Setup

The experimental setup consisted of simulating a 'player' by sending 100 sequential requests using the framework API and recording the latencies as they are processed by the backbone and received again from the API. The number of concurrent users was increased and the latencies for each user were recorded. This acts as purely a simulation to isolate and analyse the performance of the framework without network lag or client lag.

Results

The results of the aforementioned experiments are illustrated in figures 5.1 and 5.2, along with analysis of these figures below.

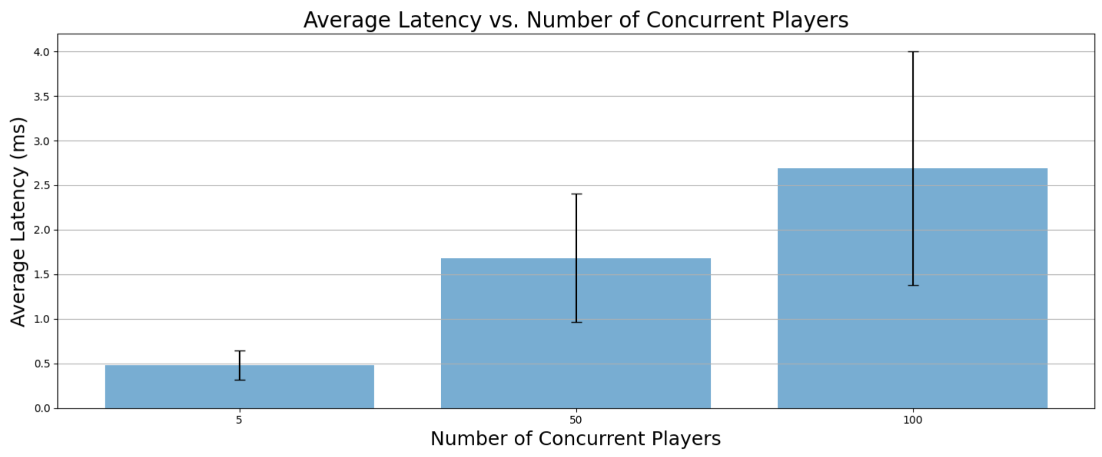

Figure 5.1: Average latencies for concurrent groups of players, with standard deviation as error bars, following the above experimental setup.

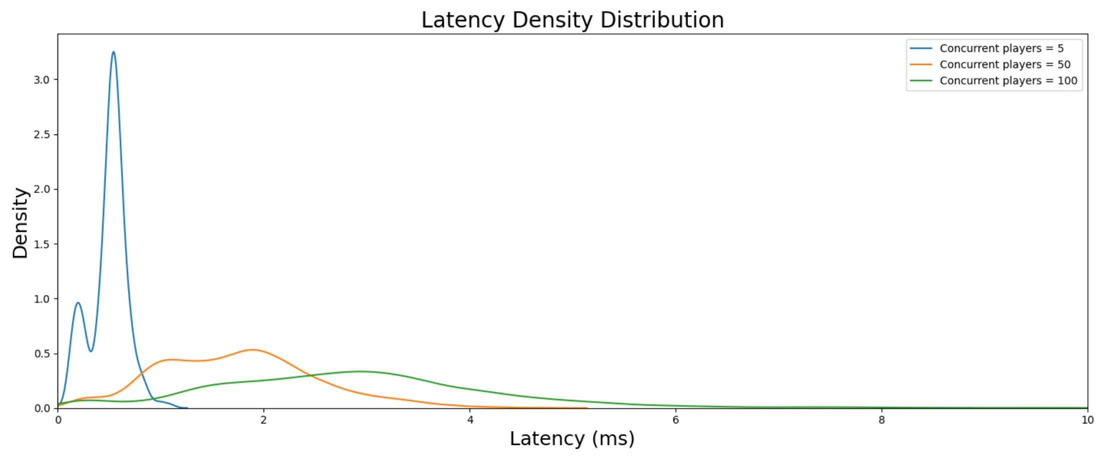

Figure 5.2: Density distribution of latencies for concurrent groups of players, following the above experimental setup.

Scalability and Performance

This experiment attempts to evaluate the baseline performance of the framework and system. Figure 5.1 illustrates the relationship between average latency and concurrent players for this setup. For 5 players, this average latency sits at around 0.5 ms and scales up to over 2.5ms for 100 players. As the number of concurrent players increases, so does the standard deviation, as shown in figure 5.2. Under the heavy load of 100 simulated players in a single stream, the system demonstrates an average latency of 2.5ms at baseline without network delay. The relationship between average latencies and concurrent players is demonstrated not to scale equally, as scaling 5 concurrent users by 20 does not scale 0.5ms to 10ms. This non-uniform scaling suggests scalability with a performance of less than 3ms.

The data suggests a significant drop in performance of the node between the 70 and 100 players, seen in figure 5.3. It is hypothesised that as a failure in the node. This may be remedied by increasing the number of available threads to serve requests or buffer size. If this system is to serve 100 players per partition, vertical scaling would be necessary to satisfy the number of incoming concurrent requests for this partition. Further testing with different configurations of virtual machines on Azure would be necessary to validate this.

5.3 Implementation

5.3.1 Qualitative

Data was collected by creating a test environment and running the game in a cloud deployment on Microsoft Azure. The number of concurrent players was scaled for each test. The following is a qualitative analysis based on several experiments.

Experimental Setup

The experimental setup was similar to the previous experiment. This time ‘Players’ were actual clients of the example game described in the implementation chapter. Each client produced 100 sequential events using the framework API, and the latencies were recorded. More specifically, the latencies described the time taken from the message production in the respective client to consuming that event sequenced by the cluster. In this case, the backbone was deployed on the cloud computing provider Microsoft Azure in a ‘B2s’ VM (2 CPUs, 4GB memory) to simulate a ‘Data Centre’ deployment. In addition to measuring latencies, the number of lost events was also recorded. Events were considered lost when the client produced an event to the framework but could not read that event from the stream.

Results

This section describes the results of the above experimental approach. Firstly, the results for performance is presented, secondly the results for reliability are presented. These are discussed in greater detail in the following section.

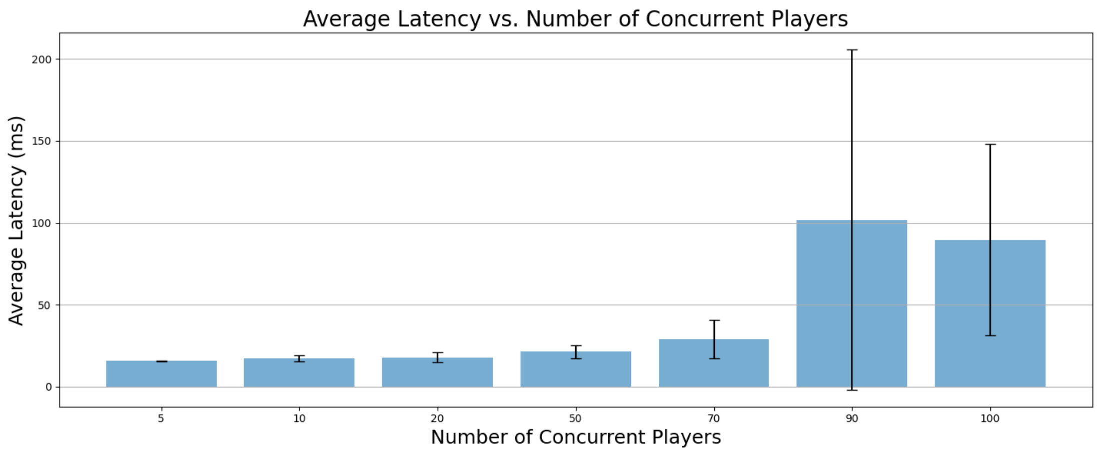

Figure 5.3: Average latencies demonstrated for a data center deployment, with standard deviation as error bars, following the above experimental setup.

Average latencies for each permutation of concurrent players are compared in figure 5.3. It is clear to see that there is a great rise in average latency between 70 and 90 players. This seemingly breaks the steady upward trend seen from 5 to 70 players before that peak.

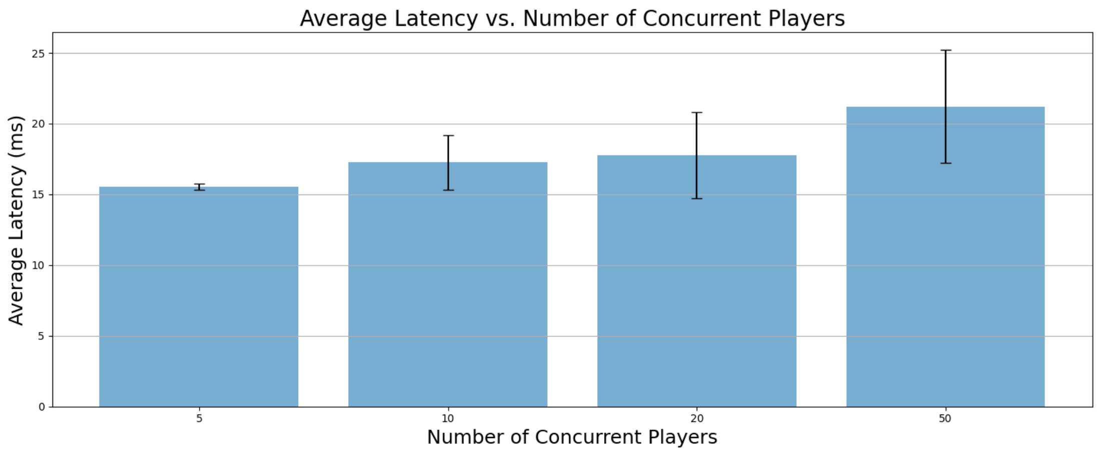

Figure 5.4: Closer look at average latencies.

The above figure 5.4 shows a close-up view of figure 5.3. Here the average latencies can more finely be identified. At 5 players, we see a latency of 15.5ms which increases to 21ms at 50 players. For only a 26% increase in latency we can achieve a 900% increase in the number of concurrent players per partition (between 5 and 50 players).

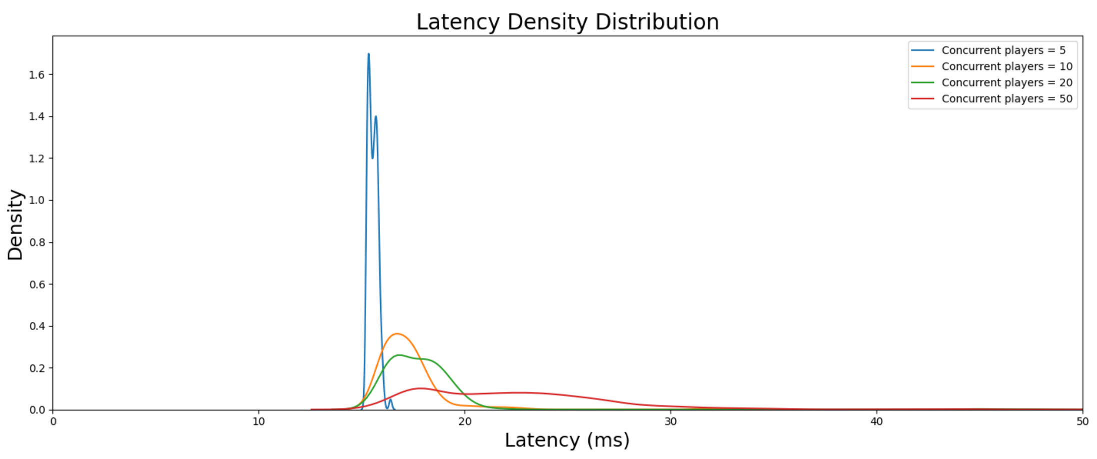

Figure 5.5: Density distribution of latencies for concurrent groups of players (5 to 50), following the above experimental setup.

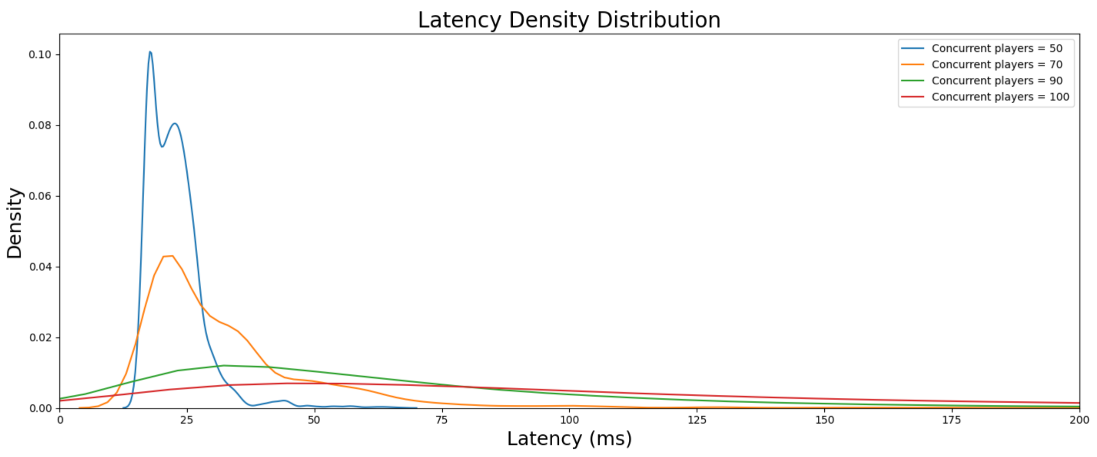

Figure 5.6: Density distribution of latencies for concurrent groups of players (50 to 100), following the above experimental setup.

In the above figures 5.5 and 5.6, the PDFs of latencies can be seen. It should be that 5.5 refers to the range of 5 to 50 concurrent players, and figure 5.6 the range of 50 to 100 players. In addition to this, note the relative scales of density, figure 5.5 describes densities of over 1, and this is due to the mathematical integral over a length of less than 1ms. An interesting feature of these two graphs is that they both exhibit similar trends between adjacent groups of concurrent players.

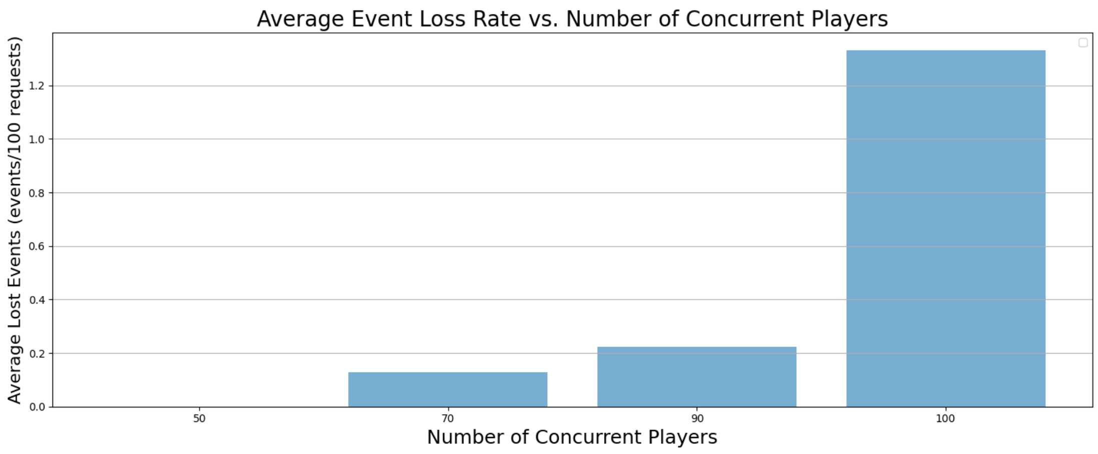

Figure 5.7: Event loss rate (events/100 requests) expressed as an average compared with an increasing number of concurrent players.

Concurrent Players	Avg. Loss Rate (events/100)	Std. Deviation (events/100)
50	0.000	0.000
70	0.129	0.411
90	0.222	0.757
100	1.33	4.186

Table 5.1: Average event loss rate in tabular form.

Figure 5.7 describes the average loss rate of events sent to the backbone by groups of concurrent clients. The values for figure 5.7 are also described in table 5.1. Once again there appears to be a steady trend (between 50 and 90 players) and then a rapid jump towards the end of the graph (90 to 100 players). It appears some bottleneck is causing issue here.

Performance and Scalability

The latency test demonstrates the worst-case scenario for this example game with 100 players concurrently writing 100 sequential events to a single partition. According to the 'British Esports Association', a latency of between 5 and 60ms is considered suitable for competitive play within first-person shooter games (FPS), while any higher than 100ms is generally undesirable for competitive play, but may be suitable for other game types BSA (2022). FPS games are considered the most demanding relative to latency and general performance BSA (2022), hence the results will be compared to this as an upper limit for performance.

From Fig 5.3, it can be seen that average latency lies within the threshold of 5 to 60ms for up to 70 concurrent players even considering the standard deviation. However, for 90 to 100 concurrent players, this threshold is reached. While the average latencies lie below 100ms, the standard deviations express the distribution lies over this number. Figures 5.5 and 5.6 further demonstrate this distribution of latencies for each group of players. While this performance is shown to be suitable for under 70 concurrent players per stream, more testing would be necessary to validate performance for a higher number of players.

Reliability

The number of lost events was recorded in the same experiment and is visualised in figure 5.7. Lost events are those that are not committed to the topic, due to buffer overflow or network failure. For consistent game-play, the player's actions should take effect in the context of the game for the user to view the effects of that action. For that to occur in this system, player events should not become lost after they are produced. The data suggests that an average event loss rate of just over 1.2% is experienced by players in the game with 100 concurrent players. Therefore, the user can expect 1 in every 100 actions they make will be lost by the server. While this is suitable for the implemented game, it is understood this may not be suitable for others. From figure 5.7 a significant relative increase in average loss rate can be seen between 90 and 100 players, this may be due to a failure within the deployment which would need to be explored in future work.

5.4 Limitations

This section covers the overall limitations of the methods provided.

Security

Security was not a primary goal for this work and was left unexplored. Future work would involve securing each stream and providing at least a basic level of security or encryption.

Dynamic Partitioning

Dynamic partitioning was not implemented in this work and would allow for an increased level of availability within the system. The example game implementation was statically partitioned at start-up based on an input parameter that describes the player limit for that session. This would be a goal for future work.

Message Compression

Event Compression was left unexplored in this work. Currently, events consist of plain-text strings converted to binary. A compression system is implemented in Kafka using several different algorithms, but this was not explored due to lack of time as the compression and decompression time would also need to be explored and evaluated.

Maximum Load on Partition

The example game implementation used Kafka as its event-driven backbone. The overall capacity of the example game is limited by its single partition capacity. This is because there is a chance that all players will be located in the same partition at once. While this chance is relatively low, it should be accounted for. By vertically scaling the nodes of each partition for a larger number of requests, the partition's ability to handle maximum load would be greatly improved. Another potential solution would be to partition during periods of high load dynamically.

Total Ordering Between Partitions

A fundamental limitation of this system comes in the form of total ordering between partitions. This is not achieved by this system; however, the model provides solutions to this problem. If total ordering is required for certain events in the game, these can be described by the model as either "Global" or "Diffuse" objects. While this may be helpful in certain games, it is not a total solution to total ordering between partitions.

5.5 Summary

This chapter performed an evaluation of the core contributions of this work: The model, the framework and the implemented distributed online game. The model was reviewed with respect to accessibility and reusability. The framework was evaluated with respect to accessibility, reusability, scalability and performance. The implemented game was evaluated with respect to scalability and performance. Limitations were also discussed with respect to the framework and the overall approach.

6 Conclusion

6.1 Summary

Chapter one provided initial background information regarding the online gaming industry as well as an outline of the problem, motivation, approach, challenges and contributions. Chapter two provided background on state of the art, which informed the design and implementation of this work. Chapter three described the reusable model that can be used to describe how a game can become distributed, as well as the terms and abstractions that permit reusability. Chapter four discussed the implementation of these ideas through a framework API as well as the creation of a distributed online game using these methods. Chapter five contained qualitative and quantitative analyses of the model, framework and game implementation as well a comment on the limitations of the methods presented in this work.

6.2 Key Findings

The implemented game was shown to support 100 players in a single partition, with an average event latency of 90ms and an average event loss rate of 1.33 events per 100 events. This did not meet the performance and reliability requirements necessary for a competitive first-person shooter (FPS) game. However, reducing the number of players to 70 satisfied these requirements for a competitive FPS game, as the recorded average latency was 28.9ms and an event loss rate of 0.12%, as the threshold for acceptable performance was 60ms.

The reason for this significant relative drop in performance between 70 and 100 players was hypothesised to be a failure of the Kafka node. The node was hosted on Microsoft Azure on the lowest tier of virtual machines.

6.3 Limitations of Work

This work evaluated the performance of a single type of distributed online game. However, there are a wide variety of games as expressed in the archetypes within the model. This work was concerned with building an abstract model and framework as well as an implementation of a distributed game using these methods. As the focus was not on building games and was more related to the process through which games are modelled to be distributed, other games were not explored. In addition to this, creating games is not a trivial task and would require more time.

The game that was explored was implemented through Pygame in the Python language as was the framework API. However, C++ is a more popular programming language for building games so a framework in this language would be more practical and as it is a lower-level language, more efficient in terms of execution time.

6.4 Future Work

In future work, the following areas will be explored:

Different deployment tiers would be investigated to identify the performance deficit between 70 and 100 concurrent players per partition found in this work. Currently, the chosen virtual machine is the 'B2' model from Azure. While the data centre deployment was explored in this study, future work will involve investigating the use of the distributed cluster approach picture in figure 4.3.

This work was concerned with building the model and framework, as well as an example distributed game for evaluation purposes. The implementation of more games would need to be explored in future work to validate the proposed methods further.

Future work would reconstruct the framework API in a more popular language for game development, such as C++ or Java.

Security was not explored in this study and will be explored in future work. In addition to this, encryption and event compression would also be investigated relative to any impact on performance these might incur.

6.5 Final Conclusion

This work presented an abstract model that was successfully used to describe a competitive, distributed online game. That model helped to standardise the process of creating a distributed online game, increasing accessibility. The model described a suitable level of consistency through an area of interest approach. The framework was composed of an event-driven backbone architecture and Python API. The framework was successfully applied to implement the modelled, distributed online game. The framework and implementation were measured and evaluated against performance and reliability concerning latency and event loss rate. The empirical evidence that was gathered suggests that the implemented game could support up to 70 concurrent players in a single partition and still provide an acceptable level of performance and reliability for competitive, first-person shooters.

These methods increase accessibility by providing an open-source, low cost, standard approach and modelling and implementing distributed, online games. While designed with reusability in mind, these methods will be explored through building more games in future work.

Based on the findings of this study, these methods stand as a valid basis for building distributed and scalable online games as well as future work and further investigation.

Bibliography

Anon (2021), 'Stack overflow developer survey 2021'.
URL: *https://insights.stackoverflow.com/survey/2021#technology-programming-scripting-and-markup-languages-professional-developers*

Anon (2022*a*), 'Lag'.
URL: *https://roblox.fandom.com/wiki/Lag*

Anon (2022*b*), 'Mag (video game)'.
URL: *https://en.wikipedia.org/wiki/MAG_(video_game)*

Authors, A. (2022), 'Documentation'.
URL: *https://kafka.apache.org/documentation/*

Authors, P. (2021), 'Pygame/pygame: Pygame (the library) is a free and open source python programming language library for making multimedia applications like games built on top of the excellent sdl library. c, python, native, opengl.'.
URL: *https://github.com/pygame/pygame*

Beigbeder, T., Coughlan, R., Lusher, C., Plunkett, J., Agu, E. & Claypool, M. (2004), The effects of loss and latency on user performance in unreal tournament 2003®, *in* 'Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games', NetGames '04, Association for Computing Machinery, New York, NY, USA, p. 144–151.
URL: *https://doi-org.elib.tcd.ie/10.1145/1016540.1016556*

Bettner, P. & Terrano, M. (2001), 'Gdc 2001: 1500 archers on a 28.8: Network programming in age of empires and beyond', *Retrieved April* **26**, 2002.

Brandt, D. H. (2009), 'Accelerating online gaming'.

Brewer, E. A. (2000), Towards robust distributed systems (abstract), *in* 'Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing', PODC '00, Association for Computing Machinery, New York, NY, USA, p. 7.
URL: *https://doi-org.elib.tcd.ie/10.1145/343477.343502*

BSA (2022), ‘Ping, latency and lag: What you need to know’.
URL: *https://britishesports.org/general-esports-info/ping-latency-and-lag-what-you-need-to-know/*

Burbeck, D. W., Bolles, E. E., Frady, W. E. & Grabbe, E. M. (1954), The digitac airborne control system, *in* ‘Proceedings of the February 11-12, 1954, Western Computer Conference: Trends in Computers: Automatic Control and Data Processing’, AIEE-IRE ’54 (Western), Association for Computing Machinery, New York, NY, USA, p. 38–44.
URL: *https://doi-org.elib.tcd.ie/10.1145/1455200.1455206*

Carpenter, A. (2022), ‘6 most popular programming languages for game development’.
URL: *https://www.codecademy.com/resources/blog/programming-languages-for-game-development/*

Chambers, C., Feng, W.-C., Sahu, S., Saha, D. & Brandt, D. (2010), ‘Characterizing online games’, *IEEE/ACM Trans. Netw.* **18**(3), 899–910.
URL: *https://doi.org/10.1109/TNET.2009.2034371*

Cheriton, D. R. (1985), ‘Preliminary thoughts on problem-oriented shared memory: A decentralized approach to distributed systems’, *SIGOPS Oper. Syst. Rev.* **19**(4), 26–33.
URL: *https://doi-org.elib.tcd.ie/10.1145/858336.858338*

Clark, M. (2021), ‘Facebook is spending $50 million to ’responsibly’ build the metaverse’.
URL: *https://www.theverge.com/2021/9/27/22696578/facebook-metaverse-ar-vr-fund-research-definition*

Corporation, R. (2022).
URL: *https://www.roblox.com/*

Dean, B. (2022), ‘Roblox user and growth stats 2022’.
URL: *https://backlinko.com/roblox-users*

Dev Team, E. (2021).
URL: *https://www.ea.com/en-gb/games/battlefield/battlefield-2042*

DFC (2020), ‘Global video game consumer population passes 3 billion’.
URL: *https://www.dfcint.com/dossier/global-video-game-consumer-population/*

Diot, C. & Gautier, L. (1999), ‘A distributed architecture for multiplayer interactive applications on the internet’, *IEEE Network* **13**(4), 6–15.
URL: *https://ieeexplore-ieee-org.elib.tcd.ie/document/777437*

Dubois, M., Scheurich, C. & Briggs, F. (1986), Memory access buffering in multiprocessors, *in* 'Proceedings of the 13th Annual International Symposium on Computer Architecture', ISCA '86, IEEE Computer Society Press, Washington, DC, USA, p. 434–442.

EA (2022).
URL: *https://www.ea.com/en-gb/games/apex-legends/about*

Epic (2020), 'Apple x epic, quarterly business review'.
URL: *https://sx.l7y.ca/DX-3519.pdf*

Epic (2022*a*), 'Fortnite – a free-to-play battle royale game and more'.
URL: *https://www.epicgames.com/fortnite/en-US/home*

Epic (2022*b*), 'The most powerful real-time 3d creation tool'.
URL: *https://www.unrealengine.com/en-US/*

FortniteGame (2020), 'We defeated him!'.
URL: *https://twitter.com/FortniteGame/status/1333954074371383296?ref_src\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{t\global\mathchardef\accent@spacefactor\spacefactor}\let\begingroup\let\typeout\protect\begingroup\def\MessageBreak{ ˙(Font) }\let\protect\immediate\write\m@ne{LaTeXFontInfo: \def{}oninputline143.}\endgroup\endgroup\relax\let\ignorespaces\relax\accent9t\egroup\spacefactor\accent@spacefactorwsrc%5Etfw%7Ctwcamp%5Etweetembed%7Ctwterm%5E1333954074371383296%7Ctwgr%5E%7Ctwcon%5Es1_&amp;ref_url=https%3A%2F%2Fwww.esports.net%2Fnews%2Ffortnite-player-count%2F*

Fritsch, T., Ritter, H. & Schiller, J. (2005), The effect of latency and network limitations on mmorpgs: A field study of everquest2, *in* 'Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games', NetGames '05, Association for Computing Machinery, New York, NY, USA, p. 1–9.
URL: *https://doi-org.elib.tcd.ie/10.1145/1103599.1103623*

Garcia-Molina, H. & Wiederhold, G. (1982), 'Read-only transactions in a distributed database', *ACM Trans. Database Syst.* **7**(2), 209–234.
URL: *https://doi-org.elib.tcd.ie/10.1145/319702.319704*

Gilbert, S. & Lynch, N. (2002), 'Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services', *SIGACT News* **33**(2), 51–59.
URL: *https://doi-org.elib.tcd.ie/10.1145/564585.564601*

Glinka, F., Ploss, A., Gorlatch, S. & Müller-Iden, J. (2008), 'High-level development of multiserver online games', *International Journal of Computer Games Technology*

2008, 327387.
URL: *https://doi.org/10.1155/2008/327387*

Hutto, P. & Ahamad, M. (1990), Slow memory: weakening consistency to enhance concurrency in distributed shared memories, *in* 'Proceedings.,10th International Conference on Distributed Computing Systems', pp. 302–309.
URL: *https://ieeexplore-ieee-org.elib.tcd.ie/document/89297*

Lamport (1979), 'How to make a multiprocessor computer that correctly executes multiprocess programs', *IEEE Transactions on Computers* **C-28**(9), 690–691.
URL: *https://ieeexplore-ieee-org.elib.tcd.ie/document/1675439*

Lamport, L. (1978), 'Time, clocks, and the ordering of events in a distributed system', *Commun. ACM* **21**(7), 558–565.
URL: *https://doi-org.elib.tcd.ie/10.1145/359545.359563*

Nichols, J. & Claypool, M. (2004), The effects of latency on online madden nfl football, *in* 'Proceedings of the 14th International Workshop on Network and Operating Systems Support for Digital Audio and Video', NOSSDAV '04, Association for Computing Machinery, New York, NY, USA, p. 146–151.
URL: *https://doi-org.elib.tcd.ie/10.1145/1005847.1005879*

Pantel, L. & Wolf, L. C. (2002), On the impact of delay on real-time multiplayer games, *in* 'Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video', NOSSDAV '02, Association for Computing Machinery, New York, NY, USA, p. 23–29.
URL: *https://doi-org.elib.tcd.ie/10.1145/507670.507674*

Powers, D. & Arthur, D. (2016).
URL: *https://kafka-python.readthedocs.io/en/master/index.html*

Roehl, B. (1995), Distributed virtual reality: An overview, *in* 'Proceedings of the First Symposium on Virtual Reality Modeling Language', VRML '95, Association for Computing Machinery, New York, NY, USA, p. 39–43.
URL: *https://doi-org.elib.tcd.ie/10.1145/217306.217312*

Sheldon, N., Girard, E., Borg, S., Claypool, M. & Agu, E. (2003), The effect of latency on user performance in warcraft iii, *in* 'Proceedings of the 2nd Workshop on Network and System Support for Games', NetGames '03, Association for Computing Machinery, New York, NY, USA, p. 3–14.
URL: *https://doi-org.elib.tcd.ie/10.1145/963900.963901*

Smed, J., Kaukoranta, T. & Hakonen, H. (2001), Aspects of networking in multiplayer computer games.

URL: *https://www.researchgate.net/publication/269251176_Aspects_of_Networking_in_Multiplayer_Computer_Games*

Steinke, R. C. & Nutt, G. J. (2004), 'A unified theory of shared memory consistency', *J. ACM* **51**(5), 800–849.
URL: *https://doi-org.elib.tcd.ie/10.1145/1017460.1017464*

Tamimi, N. (2021), 'How fast is c++ compared to python?'.
URL: *https://towardsdatascience.com/how-fast-is-c-compared-to-python-978f18f474c7*

Tanenbaum, A. & Van Steen, M. (2007), *Distributed systems*, Pearson Prentice Hall.

Team, U. D. (2022), 'Ultima online: New legacy'.
URL: *https://uo.com/*

Valve (2022*a*), 'Counter-strike: Global offensive'.
URL: *https://store.steampowered.com/app/730/CounterStrike_Global_Offensive/*

Valve (2022*b*), 'Dota 2'.
URL: *https://store.steampowered.com/app/570/Dota_2/*

Valve (2022*c*), 'Steam: Game and player statistics'.
URL: *https://store.steampowered.com/stats/*

Valve (2022*d*), 'Steam store'.
URL: *https://store.steampowered.com/*

Vogels, W. (2009), 'Eventually consistent', *Commun. ACM* **52**(1), 40–44.
URL: *https://doi-org.elib.tcd.ie/10.1145/1435417.1435432*

Waldo, J. (2008), 'Scaling in games and virtual worlds', *Commun. ACM* **51**(8), 38–44.
URL: *https://doi-org.elib.tcd.ie/10.1145/1378704.1378716*

Ward, I. (2022), 'Warzone: Best free battle royale game'.
URL: *https://www.callofduty.com/warzone*

Webb, S. D., Lau, W. & Soh, S. (2006), Ngs: An application layer network game simulator, *in* 'Proceedings of the 3rd Australasian Conference on Interactive Entertainment', IE '06, Murdoch University, Murdoch, AUS, p. 15–22.
URL: *https://dl-acm-org.elib.tcd.ie/doi/10.5555/1231894.1231897*

Williams, L. (2022), 'A pandemic is a dream come true for gamers'.
URL: *https://www.bloomberg.com/opinion/articles/2022-01-16/pandemic-s-boost-for-video-game-industry-is-a-dream-come-true-kyh9nekz#:*

~:text=The%20video%2Dgame%20industry%20could,reach%20%24219%20billion%20by%202024.