Horizontal scaling for spatial environments: Applying Distributed Computing Technique to Build Large Virtual Worlds

A Third Year Project

Fabian Desnoes

Supervised by Ligang He

Table of Contents

Abstract *3*

Introduction *3*

Related Work *7*

SpatialOS **7**

Background *8*

Latency **8**

Network Latency 10

Microservices **11**

Communication 12

Load Balancing **13**

Objectives *14*

Proposed Solution *14*

Client **14**

Transport API 15

View 16

API Gateway **18**

Frontend 20

Reverse Proxy 21

Datastore **22**

Distributed Caching 22

Query Routing 24

Ownership 25

Workers **25**

Work Distribution 27

Minimum Viable Product (MVP) *28*

Technologies Used **28**

Golang 28

Datastore 29

Project Management *31*

Scope **32**

User Stories 33

Features & Epics 34

Requirements 35

Timeline **36**

Risks **37**

Evaluation *37*

Objectives **38**

Project Management **39**

Limitations **39**

Conclusion *40*

Future Work **40**

Spatial DB.....40
References.....40

Note to Examiner: This project is a mixture of both research and practical project. The main goal to was to propose a design for the system and the secondary target was to develop a working prototype to benchmark and explore different approaches.

Abstract

This paper presents a platform designed to provide horizontal scaling for virtual worlds. Virtual worlds are becoming increasingly important due to the progression of AI, AR and VR. However, to capture this increase in value, these worlds must grow in size and complexity. By exploring techniques from distributed systems and modifying them to suit a low-latency environments, we present a system architecture that removes the limitations of traditional approaches through horizontal scaling. In addition, we outline the development of a demo that showcases the design's capabilities and demonstrates the potential of our approach.

Keywords: Distributed Computing, Fault Tolerant Systems, Networking, Horizontal Scaling, Load Balancing, Distributed Caching

Introduction

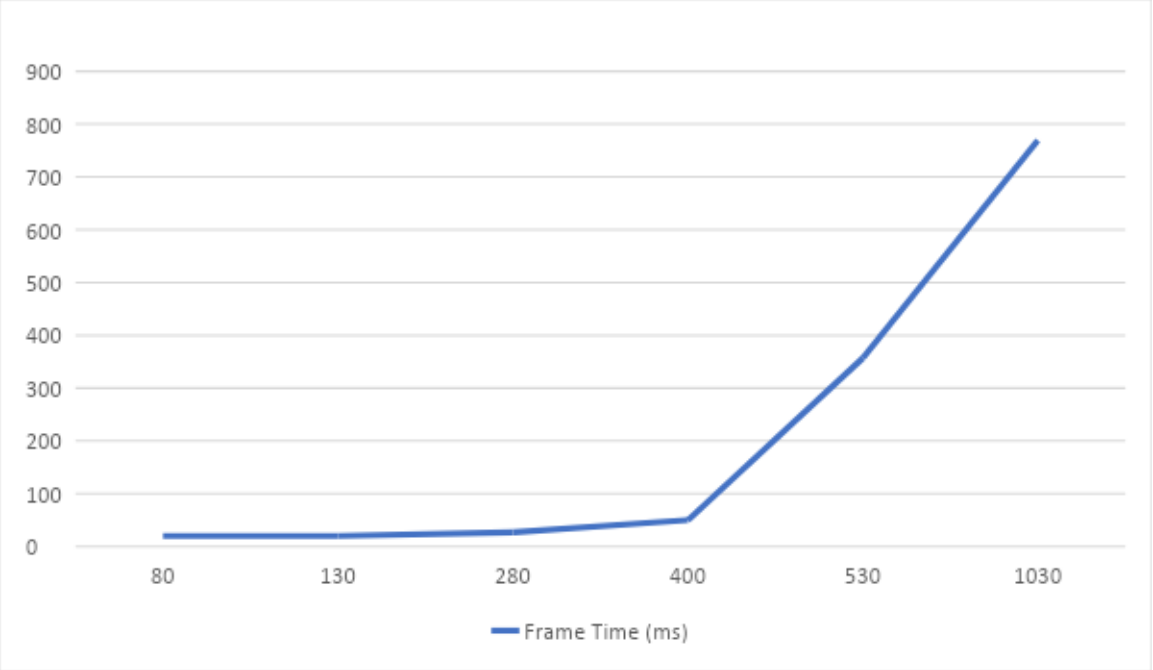
A virtual world is a real-time computer-simulated environment where agents can virtually interact with each other and the environment. As agents interact with the world, the changes must persist and synchronize with each agent. For example, in the gaming industry, Massively Multiplayer Online (MMO) games allow users to explore immersive environments through digital representations of themselves known as avatars. These virtual worlds also support interaction between avatars in real-time chat, virtual goods trading, and much more.

Over the last few decades, virtual worlds have primarily been used within the gaming industry to provide immersive experiences for players. However, over time new industries have begun adopting virtual worlds. For example, within the AI industry, virtual worlds are used to train large RL-based models that perform complex tasks such as Autonomous driving[2,3], advanced robotic locomotion4 and physics simulations. Training in virtual environments allows companies to be more agile and cost-effective. For example, building an autonomous vehicle costs hundreds of thousands of pounds. A company could design and train various models virtually by utilizing virtual environments before investing in a real-world prototype. Not only would this save the company money from not having to

manufacture the vehicle, but it would allow rapid prototyping as they would not have to wait for the prototypes to be manufactured, therefore improving the rate of development [5,6].
In addition, virtual worlds provide a safe, realistic environment to train RL models[6]. For example, allowing an autonomous vehicle to drive around a busy city would be incredibly dangerous. With virtual worlds, modern games engines, such as Unreal Engine 5, can be used to render high-fidelity environments in real time, providing a realistic, transferable environment to train these kinds of models.

Also, Consumer demand for high fidelity virtual worlds has also increased over the years due to technological advances in Virtual & Augmented Reality (VR/AR) headsets. For example, the number of AR & VR users is expected to grow from 171 million user in 2023[7] to 2.593 billion users by 2027[8].

As previously mentioned, a virtual world is a simulated environment that allows users to interact with each other and the environment in real-time. Defining what is classed as real-time can be quite difficult as it often varies based on the application. In game development, which closely aligns with virtual worlds, updating the world 25 times per second is classed as sufficient to provide a real-time experience to the user[9]. This means that in order to provide a seamless experience to the user, each update within the world must be made within 40 milliseconds. While this may not be a challenge in small virtual environments, it becomes increasingly difficult as the number of agents and users in the world grows.



Comparison of the number of objects and frames time for a single machine[11].

As most entity-based simulations have a time complexity of $O(n^2)$[10], increasing the size of a simulation can significantly decrease performance. To combat this, additional computing resources can be added to the system, however as the relationship between compute resources and execution time is not linear, the requirements of the system quickly surpass the capabilities of available consumer hardware[11].

There are two different methods for scaling for such a system: horizontal scaling and vertical scaling. Vertical scaling is when more compute resources, such as CPU cores or RAM, is added to a server to help meet an increase in demand. For example, if a database server reaches 80% memory usage, additional memory might be added to the server to help accommodate the increased demand. On the other hand, Horizontal scaling is when we increase the number of server instances in order to accommodate system load. For example, if a database server reaches 80% CPU load, then another server can be setup so that all traffic can be load balanced so that both servers now have 40% CPU load each.

Traditionally, most applications including virtual worlds are scaled using vertical scaling. Vertical scaling is a very cost-effective way to scale a system in comparison to horizontal scaling [12]. For example, in vertical scaling, increase CPU or RAM only requires purchasing some memory modules or a new CPU whereas with horizontal scaling, an entire new server, including a case, motherboard and power supply, is needed.

In addition, a system utilising vertical scaling is much easier to manager than one using horizontal scaling [12]. For example, a system administrator managing a vertically scaled application only needs to monitor and maintain a single machine whereas in horizonal scaling the number of machines could significantly increase.

Also, vertically scaled systems are significantly simpler to build than horizontally scaled systems [12]. For example, in a horizontally scaled system, data is scattered across machines and locating the correct information can be difficult. In a vertically scaled system, all data in located on a single machine, meaning no extra logic is required to access and maintain information.

Despite vertical scaling being a much cheaper and simpler method of scaling systems, it has a few limitations which prevent it being used to scale large systems. Firstly, there is a limit to the amount of computing resources you can add to a single system [12]. For example, on Amazon Web Services (AWS), a cloud computing provider, the maximum computing resource a single general purpose Virtual Machine (VM) can have is 128 CPU cores and 512GB of RAM_{13}. As a result, for systems which require higher levels of scalability, vertical scaling is not a feasible method and horizontal scaling is often preferred [12]. This is because with horizontal scaling, you can always increase the number of machines available to the system therefore providing a much more scalable alternative to vertical scaling.

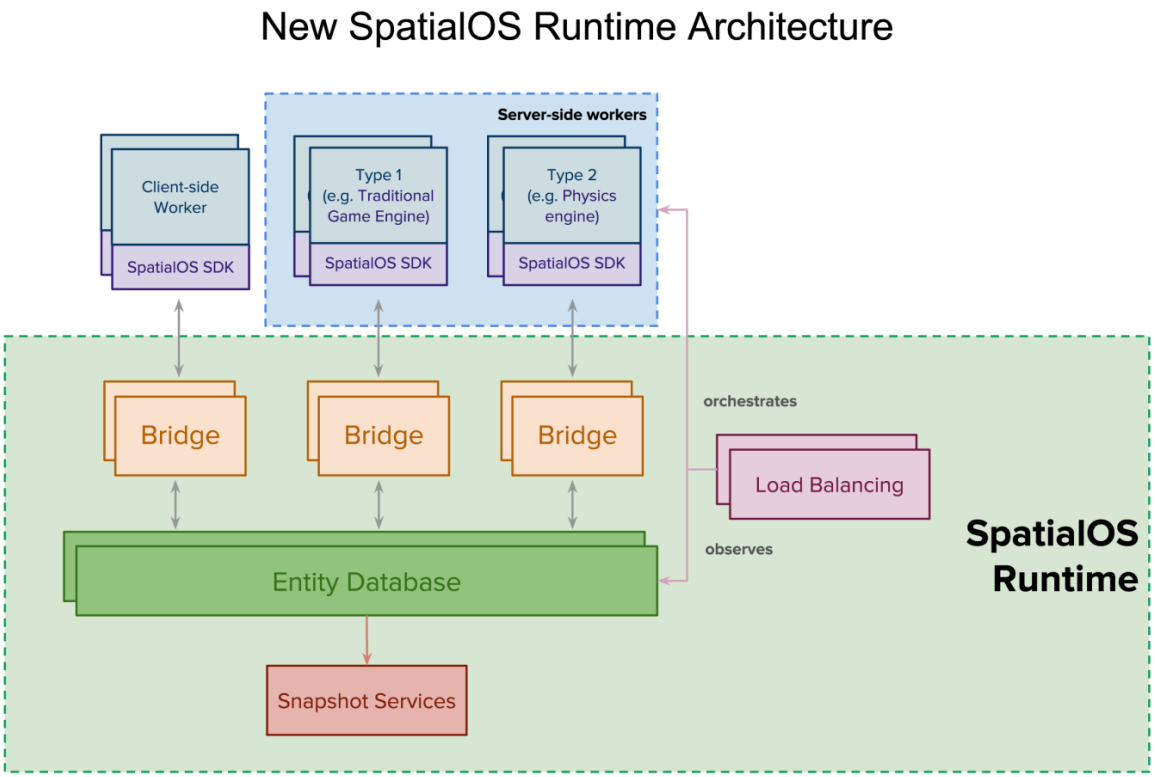
In addition, since vertically scaled systems only run on a single machine, they are less fault tolerant than systems running on multiple machines [12]. For example, if a server fails in a vertically scaled system, then the entire application would become unreachable. This can lead to significantly higher downtime for a service as hardware will eventually fail or system maintenance may be required. Horizontal scaling on the other hand does not suffer from this as the systems load is shared across multiple machines. If a single sever instance fails, then the load will dynamically be sent to the other machine that the system is running on. As a result, system which require fault tolerance often use horizontally scaling as it offers higher availability than vertical scaling.

From this discussion is it clear that to provide a responsive and stable virtual world experience, vertical scaling is not sufficient. This is because large virtual worlds require large

amounts of computation and therefore benefit from the limitless scale offered by horizontal scaling. Despite this, most virtual worlds still employ vertically scaling due to the complexity and cost associated with designing, building and maintaining horizontally scaled systems. As a result, the primary aim of this project is to build an underlying system which supports the development & deployment of distributed virtual worlds. The aim will be to abstract the complex procedures required to create large-scale virtual worlds and make it more accessible for smaller development studios. In addition, this should allow developers to spend more time building the application and less time on the underlying infrastructure.

Related Work

SpatialOS



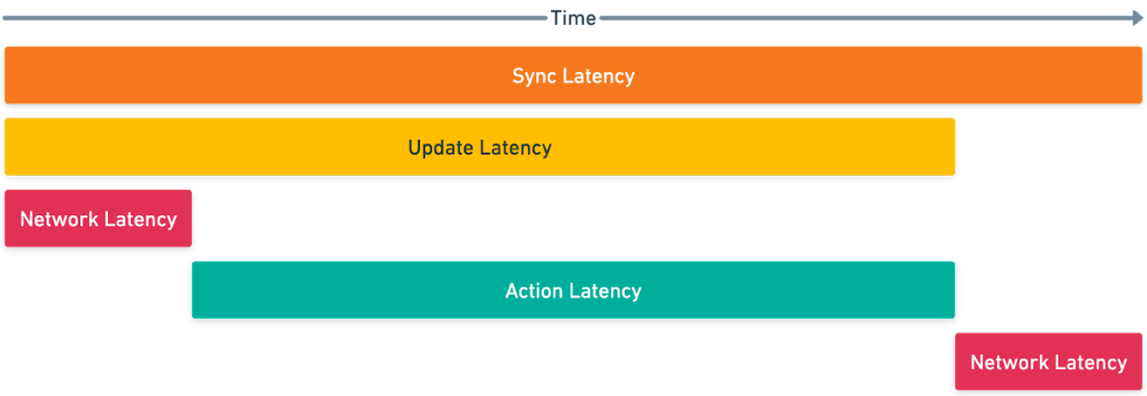
SpatialOS was a runtime environment created by Improbable to facilitate scalable, virtual worlds. It is now discontinued and now longer advertised on Improbable's website. Originally, SpatialOS used a worker system where each worker node would managed a region within the virtual world. It featured a shared memory architecture where workers would access a centralised cache via bridge nodes. These bridge nodes perform authentication to ensure workers were only accessing data they were managing. SpatialOS

failed to gain traction as many users stated it was too expensive to use for most use-cases. In addition, SpatialOS focused largely on the gaming sector, neglecting other use cases such as those previously mentioned.

Background

Latency

The latency of a system is the time delay between the cause and effect. Low latency is trivial to providing users with a smooth and responsive experience. This is because high latency can create noticeable delays and lag between user input and system response. For example, in online gaming, high latency can cause a delay between a player's action and the corresponding action in the game, which can negatively impact the player's ability to compete or enjoy the game as players with lower latency have a completive edge.



Within our system, we have defined four types of latency that contribute to the overall user experience: Sync latency, Update latency, Action latency, and Network latency. By identifying and defining these four different types of latency, we explore the factors that contribute to the overall performance of our system and take steps to minimize them for a smoother and more engaging user experience.

Sync latency is the time delay between when a user A performs an action and Player B receives an updated world state. Sync latency is particularly important in virtual worlds, as if it is high, an undesirable effect known as ‘Rubber banding’ can offer. Rubber Banding describes when objects appear to continuously teleport between different locations due to differences in the predicted state of an object and its actual position. For example, if Client A

doesn’t receive any updates from an entity, then it might assume that entity will continue to move in the same direction. If the entity in fact changed direction, but the update was received much later, then the entity would be in a completely different location therefore when the update is received, the entity would teleport to a its correct location.

Update latency is the time it takes for the world state to be updated after a user A performs an action. Update latency is similar to sync latency however it does not include the time taken to update each client. As a result, it focuses more on how long it takes for the client’s action to be acknowledged by the system.

Action latency is the time it takes for a user's action to be processed and performed by the system. Throughout the project, this was used as the primary benchmark to evaluate the performance of the system as it clearly showed the relationship between entity count and system performance.

Network latency is the time it takes for a user's action to reach the system. This type of latency is influenced by factors such as the physical distance between the user and the system, the quality of the network connection, and the amount of network congestion.[14] Network latency can impact all aspects of the user experience, including sync latency, update latency, and action latency.

When it comes to defining what is considered "low latency", it is important to take into account the specific use case and requirements of the system being developed. In other words, the level of latency that is acceptable for one application or use case may not be suitable for another.

In the case of our project, we have established that a simulation speed of 24 frames per second (FPS) is the minimum acceptable level for providing a responsive environment[9]. This means that the system must be able to update the game state at least 24 times per second to ensure a smooth and seamless experience for users. To achieve this level of responsiveness, we have set a goal of ensuring an average sync latency of less than 40 milliseconds. Sync latency, as previously defined, is the time delay between when a user performs an action and when all other players receive an updated game state.

Network Latency

Network latency, also known as network delay, refers to the time taken for data to travel between two points on the network. There are several different factors that can contribute to network latency.

One of the most significant is propagation delay. Propagation delay refers to the time taken for a signal to travel through a medium. This delay is directly proportional to the distance between the client and server and can be calculated using the equation distance / propagation speed. To minimize propagation delay, it is essential to minimize the distance between the client and server or choosing a medium which is optimised for the exchanged of data. For example, fibre optics cables are often used over copper cabling as fibre optic cables have a higher propagation speed[14]. Alternatively, Edge computing is a popular approach that aims to reduce the distance between the client and server by deploying small data centres closer to the end-users[15].

Another factor that can impact network latency is network congestion. The bandwidth of a network indicates the maximum amount of data can be transferred per second. As a network becomes more congested the maximum theorical bandwidth might be met, resulting in a loss of packets. In some cases, protocols such as TCP offer congestion control which control the rate at which packets are sent across a network to prevent this from occurring. By slowing down the rate at which packets are sent, the total round-time trip of each packet increase, resulting in higher latency. To combat this, high-bandwidth networks, such as 10 Gigabit networks, can be used to provide faster and more efficient data transmission[14].

Within the TCP/IP stack, transport protocols are responsible for sending data across a network. Each transport protocols provides different functionality and therefore can influence network latency. For example, TCP provides reliable communication by providing features such as flow control, error detection and reordering. This requires additional packet processing and results in an increase of latency when compared to unreliable protocols such as UDP. In summary, choosing an appropriate transport protocol is essential for reducing latency.

While there are several other factors that can impact network latency, including routing delays and processing overheads, minimizing propagation delay, reducing network congestion, and optimizing transport protocols are some of the most effective ways to ensure low latency and optimal network performance. By understanding and addressing these factors, we can ensure our system is responsive and offers a smooth user experience.

Packet Size

To send data across a network, it must first be divided in packets via a process called fragmentation. The size of each packet is determined by the networks Maximum Transmission Unit (MTU). MTU represents the largest size a packet can be before fragmentation is required. As most of the internet is built with Ethernet, most computer networks support an MTU of 1500 bytes[16]. As each packet sent across a network will suffer from network latency, it is important to optimize how we send data across the network. For example, smaller messages should be packaged together so that they can be sent in the same packet payload. For larger messages, data can be compressed as the computation required to perform the compression can often be compensated for by the reduction in latency.

Payload Format

Before data is sent across a network, it must be serialized and converted into binary. There are several different data exchange formats which offer a specification for performing such operations. Most data exchange formats make a trade-off between ease-of-use and speed. For example, XML uses strings to represent data and includes open and closing tags to specify data. As a result, it is very verbose in comparison to ither exchange formats such as JSON[18] and therefore performs worse[17]. In addition, some data exchange formats, such as Protocol Buffers, choose to represent data as raw binary and therefore are much more efficient when serialising data[17].

Microservices

There are two main software architectures: monolithic and microservices.

A monolithic application is where all the functionality is built as a single deployable unit.
The Monolithic architecture is a traditional approach where an entire application is built as a single unit or module, which contains all the functionality required by the application. The

monolithic application is designed to handle everything from the user interface to data storage and everything in between. On the other hand, microservice architecture is a modern approach where an application is broken down into small independent services, each performing a specific function or task.

Most software projects choose to adopt a monolithic architecture over a microservice architecture as they are easier to build and maintain[12, 19, 20]. For example, a microservice application will have several individual components all interacting with each other. In the case that the system becomes unavailable, it can be hard to identify where the failure has occurred and what is causing it. In contract, since monolithic applications consist of a single unit, it is easier to locate and debug system failures.

In addition, monolithic application often performs better than microservice applications [12, 19]. This is because microservice are fine-grained by design meaning high-level operation often require interactions between several services. As a result, each operation can require multiple different network calls to other services, therefore significantly increase the duration of an operation. As monolithic applications are deployed as a single unit, no additional network calls are required and therefore can be expected to perform better than microservice applications.

On the other hand, microservice application offer better scalability when compared to monolithic applications [12, 19, 20]. This is because each service within a microservice application can be scaled independently. Monolithic applications are a single large block meaning under heavy system load, the entire application must be scaled together. By not having fine-grained control over which parts of the system are scaled, it makes it harder for monolithic application to respond to system load and therefore often cost more.

Communication

Microservices are fine-grained by design and therefore often require the coordination of several services to perform an operation[20]. To do this, services need a way to communicate with one another. There are two different communication patterns to choose from when implementing inter-service communication: synchronous and asynchronous communication[21].

Synchronous communication is when a client sends a request and blocks until a response has been received[21]. For example, when loading a webpage, synchronous communication is used to request the website resources in order to render the page. Synchronous communication is commonly used in situations where a service needs a response to continue its operation[21]. For example, a service which process an employee's payroll might need to request information from the employee service in order to pay an employee their salary. Without receiving information about the employee, the payroll service wouldn't be able to carry out its task and therefore synchronous communication is appropriate. One of the major drawbacks of synchronous communication is that it tightly couples services together[21]. For example, for two services to interact, the address of the receiver must be known in advance. This adds complexity to the system as service discovery must be performed.

Asynchronous communication is when a client sends a message and does not wait for a response. An example of asynchronous communication would be communication use a message-oriented middleware, such as a messaging queue. Messaging queues provide a way for service to append message to a queue while other service can pop message off the queue. Messaging queues allow services to add messages to a queue, while other services can remove messages from the same queue, eliminating direct communication therefore loosely coupling services. For example, when a user creates an account, they might upload a profile picture. The user service may then add a message to a message queue for the compression service to inform it that the picture should be compressed to save space. As this action doesn't have to be performed instantly, and it might take a lot of time, it makes sense to use asynchronous communication over synchronous communication. Also, in this case, the user service doesn't need a response from the compression service, it just needs a way to tell it to perform some tasks on its behalf.

Load Balancing

In order to benefit from horizontally scaling, load balancing must be performed to ensure each worker receives a similar amount of work. Ensuring each worker is equally balanced is paramount to providing a responsive user experience. If a worker was to become overloaded, then all entities within the region would suffer from lag as the game loop would not be able to execute with 40ms.

In spatial environments, such as a virtual world, entities usually interact with nearby entities. If round robin was used, then many cache misses would occur. This would cause a significantly decrease in the perform. As a result, we use an approach called spatial load balancing. This involves breaking the virtual up into regions and assigning areas with a similar number of entities to each worker. As a result, nearby entities are stored and managed by the same worker, therefore reducing the chance of a cache miss and improve system performance.

Objectives

- Systems components must show a linear relationship between performance and entity count.
- The systems must automatically scale to meet demand in real-time using horizontal scaling to ensure an average game loop time of less than 40ms.
- Clients must be able to establish bidirectional communication to send and receive updates to and from the system.
    - The system should allow clients to subscribe to entity updates to synchronize their local representation of the world.
- The system must persistently store the state of the world such that if the system shutdown, the world's state can be recovered.
- The system must provide a high-level abstraction of the systems via a client.
- The system should be highly available and have 99.99% uptime when running for 24 hours.
- The system could support a wide-range of different transport protocols to support different use-cases.

Proposed Solution

Client

A client is an interface which allows an application or service to access resources provided by a server. Within our system, clients are responsible for interacting with the system to provide a real-time representation of the virtual world to end-users. The client provides bidirectional communication between the user and the system, providing a way for the user to send input, while also receiving updates from other entities in the world. The primary aim of the client is

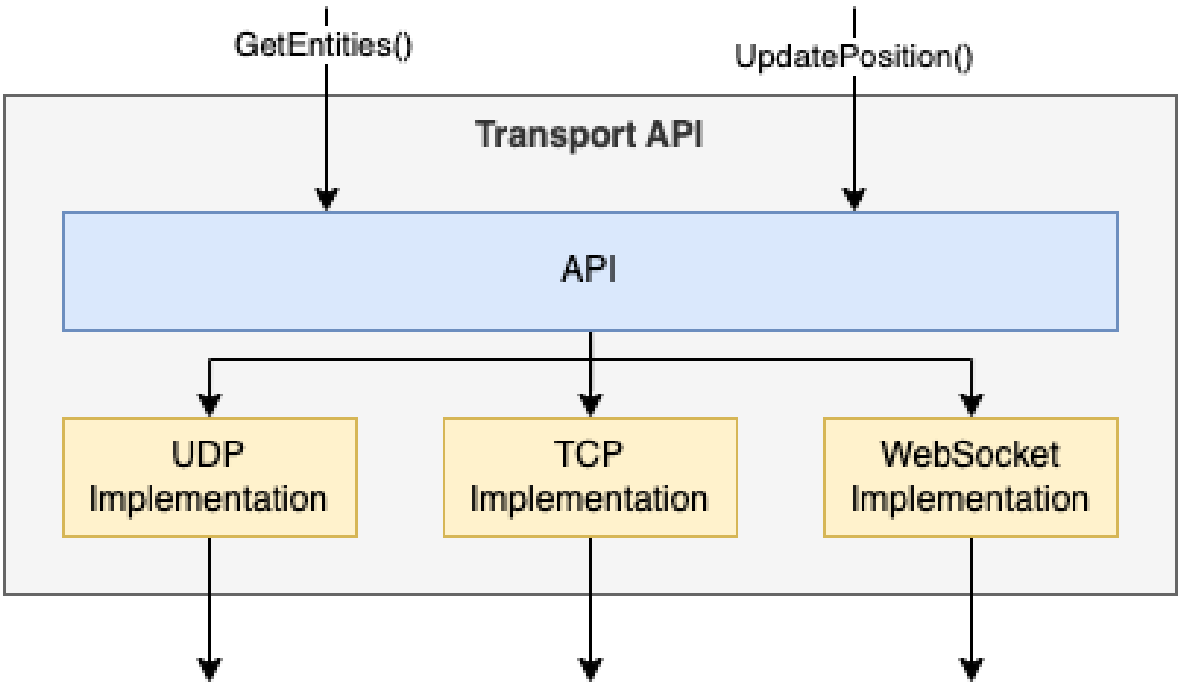
to provide a high-level API of the system so that the system can be used for a wide range of applications.

Transport API

To send and receive information, the client and server must establish a mechanism for sending and receiving messages. In computer networking, the TCP/IP stack illustrates the different protocols used to allow communication between two devices and how they interact with each other. Within the TCP/IP stack is the Transport Layer (L4). The transport layer is the 4th later within the TCP/IP stack and is responsible for transporting packets of data from a source host to a destination host.

Deciding which transport protocol should be used in a system can be challenging as each protocol makes different trade-offs between speed and reliability. For example, two of the most common transport protocols, User Datagram Protocol (UDP) and Transmission Control Protocol (TCP), both offer a completely different set of features. TCP is a reliable transport protocol which ensures packets are always received by the destination hosts. This is accomplished through several different mechanisms such as congestion control, packet ordering and an acknowledgement system. By offering high levels of reliability, TCP must perform significantly more packet processing in comparison to more lightweight protocols such as UDP, therefore increasing latency[22]. UDP prioritises speed over reliability and has a much smaller feature set. For example, UDP provide the minimum functionality to send messages across the network, therefore making it much faster.

As shown above, choosing which transport protocol a system should adopt is a difficult process which is heavily dependent on the systems use case. As a result, we wanted to ensure that our system provided flexibility for developers to decide which protocol they wanted to use for their application while abstracting the technical details for each protocol. This led to the development of the Transport API.



The Transport API is a component within the client which facilitates sending and receiving information to and from the system. It provides a simple higher-level interface which is consistent no matter what transport protocol is being used underneath. This provides developers with flexibility as they could change the protocol without having to update any mode. In addition, this allows developers to quickly create different clients for different platforms without having to write any code.

The Transport API exposes two types of operations: standard request-response and streaming. Request-response is a paradigm where a client spends a request to the server and then a single response it send back to the client. This is primarily used when the client wants the server to perform an action on its behalf. Streaming on the other hands is where a client establishes a connection with the server and sever messages are sent either to or from the server over a period of time. Streaming is commonly used in real-time systems where the client needs to aware of updates as soon as they happen.

View

The view provides a visual representation of the virtual world that users can interact with. Typically, a modern game engine, such and Unreal Engine or Unity, would be used to render the game world in real-time. Also, the view is responsible for handling all user inputs and sending them to the system via the Transport API.

Latency Compensation

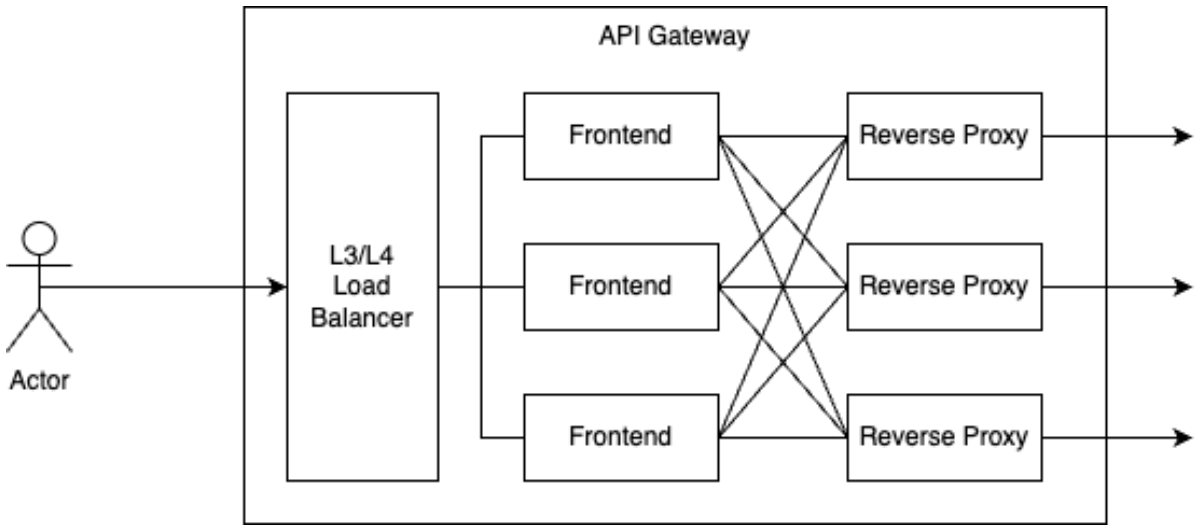
To provide a smooth experience to the user, the view is also responsible for performing client-side predications for all entities. Client-side prediction is a common technique used to compensate latency in networked applications such as multiplayer games[23]. It works by assuming the current local state of the world is up to date and that the outcome of performing the action locally will have a similar result to the updated state recovered from the server. For example, if the user moves their avatar forward in the world, the view will move the user's avatar forward in its local state, assuming the server will perform the same action. Sometimes however, the predicted state and actual state are very different which can result in a highly noticeable change in the client's world. To minimize this, we need to ensure that each client's local state is frequently updated and is never too out of date.

Performance Optimizations

In virtual worlds, the number of entities or agents present can vary widely, and each of these entities can send multiple updates per second. For example, a world with 10, 000 entities each updating at 24 FPS, would result in 240, 000 updates per second for each client. In large worlds with thousands of entities, clients would become overwhelmed and struggle to store and process everything in real-time. As a result, a more efficient approach is used to process only portions of the world that are relevant to the client's position. For example, only entities within a predetermined radius of the user will be sent to the client.

Although this significantly reduces the number of updates received by a client, similar problems can arise in highly dense areas. To mitigate this, the Level of Detail (LOD) system is used to reduce the number of entity updates further. Essentially, the LOD system applies a weight to each entity based on its distance from the client. Entities with a higher weight are then updated more frequently than those with a low weight. For instance, if an entity is located right next to the user, then its position would get updated 24 times per second. On the other hand, if the entity is located 20 meters away from the user, then it would only receive 12 updates per second.

API Gateway



Our system is comprised of several different microservices which work in tandem to perform operations. For example, to update the position of an entity several service calls are required to perform collision detection, physic simulation as well as saving the new position. As a result, the client needs a way of interacting with the system services to perform operations.

One approach that we could have taken was to allow clients to directly communicate with each service. For example, each service could implement and API which the client would calls to receive a result. The client could then chain a series of service calls together to perform operations such as updating an entities position. On the other hand, another common approach would be to implement an API gateway. An API gateway acts as a single point of entry to a system and all requests go through it. For example, instead of making several requests to different services, a single request would be sent to the API gateway, and it would make requests to services on the clients behalf.

Allowing the client to directly access each service in the system can lead to tight-coupling[24]. This means that any changes made to a service would require updates to be made to the client as well. For example, if a service was changed or removed, the client would need to be updated accordingly. This can create a lot of technical debt and discourage changes being made to the system[24]. This can be particularly problematic when following an Agile development methodology, where changes to services are expected to occur frequently. Therefore, tight coupling can lead to a slower development process and hinder the system's ability to be easily updated and maintained over time.

In addition, directly accessing services would require the client to know the network address of each service[24]. As our system utilizes horizontally scaling, the number of services, and their network address, change on frequent basis. This process is known as service discovery and is later explored within this report. For now, it is important to note that this process requires querying an additional service whenever a request is made. As a result, this would significantly increase the overall response time.

To prevent tight coupling, an API gateway can provide a high-level abstraction of the system[24]. For example, instead of the client having to make several requests to different services, a single request can be sent to the API gateway, and it can make the requests on the clients behalf. This has two primary benefits. Firstly, it means that the client only needs to be aware of the network address of the API gateway. As this doesn't change frequently, it means no service discovery is require, therefore reducing the overall latency of the system. In addition, by loosely coupling the client, it means that internal changes to the system have no effect on the client. As a result, less technical debt is introduced when making changes to survive, therefore allowing us to efficiently iterate the design of the system.

Microservice are designed to be fine-grained and typically perform a single task/function[20]. This means that when performing high-level operations, several request will be made[20]. If our client was to directly communicate with services, lots of network requests would have to be made whenever an operation was performed, therefore introducing large amounts of latency. For example, if the round-time trip of a single request was 12ms, performing a high-level operation involving 3 different services would increase the overall round-time trip to 36ms. As latency is a primary concern of ours, this approach would not be feasible.

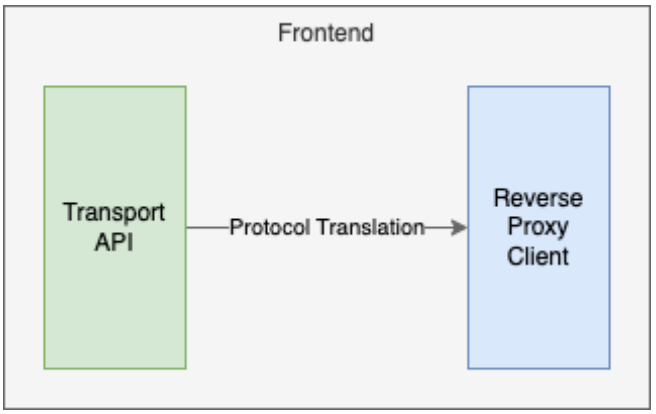
API gateways tackle this problem by aggregating services calls on the clients behalf[24]. As our system is deployed in the cloud, it means that all servers are connected via a high-speed 10 Gigabit network. Also, all services are deployed within a single geographical region meaning latency is significantly reduced. As a result, the latency penalty associated with make several requests is significantly reduced by performing them at the API gateway instead of the client.

Although API gateways provide many solutions to the problems discussed, they also introduce their own disadvantages. For example, since API gateways act as a single-entry

point to the system, they introduce a single point of failure. If the API gateway was to become unavailable, then the client would not be able to access any services. A potential solution to this problem would be to horizontally scale the API gateway so that multiple instances are always running. As a result, in the case of one of the instances failing, the other instance can continue to serve incoming requests, In addition, an API gateway adds an additional network trip onto all requests. For example, an operation requiring requests to 3 services would grow to 4 as the request would first have to reach the API gateway before the requesting resources from the services. As mentioned before API gateways do reduce the latency of these services calls and therefore this extra network call is compensated for.

Overall, we believe that implementing an API gateway was the better option for system. This is because when looking at our requirements, ensuring the system had low latency was a primary objective. Also, one of the main risks of the project was running out of time. By using an API gateway, it not only reduced the overall latency of the system, but it would allow us to build the system quicker and make improvements to service without worrying about the effects on the client.

Frontend

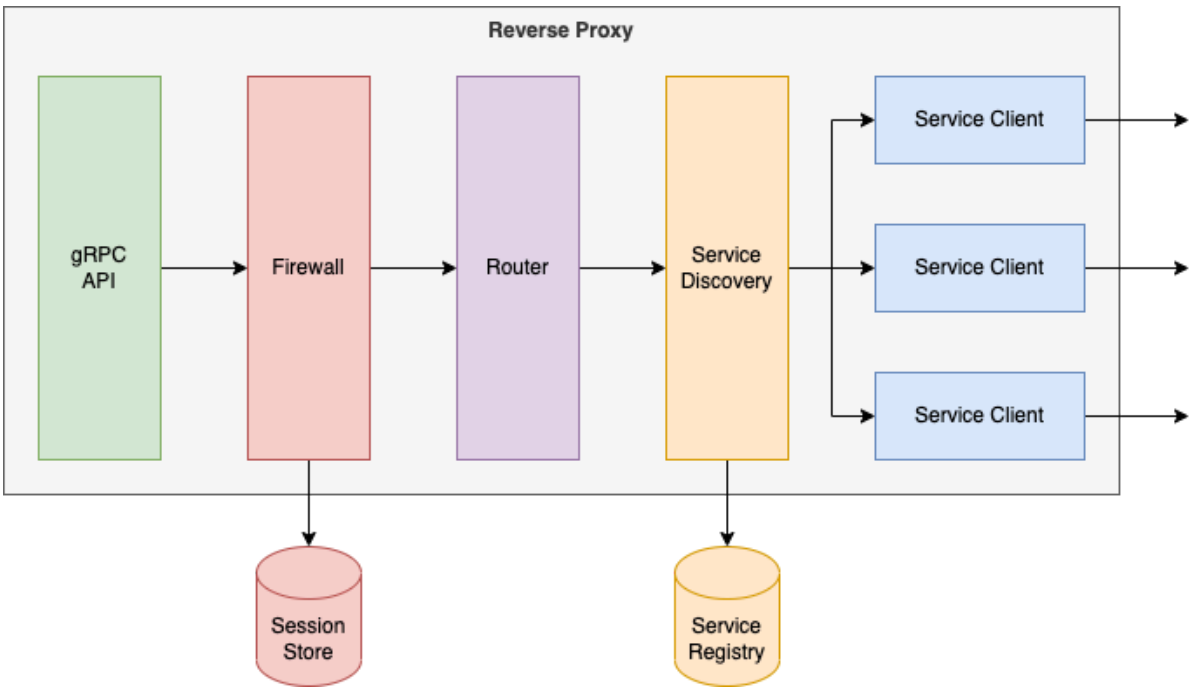


When designing the client, the Transport API provides support for several different Transport Protocol. The frontend is responsible for establishing and maintaining connections with clients, listening for incoming messages and converting the incoming protocol into gRPC, which is used by the reverse proxy client. Preforming the protocol translation allowed the reverse proxy to have a single API instead of having to support several different APIs for each transport protocol. This means that the reverse proxy would be loosely coupled with the client and therefore less technical debt was created when modifying the Transport API.

In addition, separating the frontend from the reverse proxy, allowed us to improve the overall availability of the system. If the reverse proxy was to become unavailable, then all connections would be lost. Connection-based protocols such as TCP perform expensive handshakes in order to re-establish connection. As a result, separating the frontend meant that we could deploy more frontends than reverse proxies to ensure that less connection are made to each frontend. Therefore, In the occasion of frontend failure, less users are affected, and their connections can be established quicker. Also, doing this would allow us to use a more reliable language such as Elixir which was used by WhatsApp for similar reasons[25].

As the Frontend was a mission critical component of the system, we needed to ensure that it was highly available. To achieve this, we utilized horizontally scaling to ensure several instances of the frontend were always running. This prevent the frontend being a single point of failure, while also ensuring it would be able to scale with the number of concurrent players. By horizontally scaling the Fronted, we had to deploy an L3/4 load balancer so that all connections were evenly distributed amongst frontends.

Reverse Proxy



A key responsibility of the API gateway is to provide a single endpoint that all client messages can be sent to. As previously mentioned, this ensures the client is loosely coupled to systems services, providing us with the flexibility to modify and change the inner workings of each service without having to update the client[24]. Once a message reaches the API

gateway, it needs to be redirected to the appropriate service so that the requested operation can be performed. This is the role of the reverse proxy.

The reverse proxy comprises of 5 components which work together to route incoming messages based on the message's content. The first component of the reverse proxy is a gRPC API which acts as the main entry point for all client requests and is responsible for receiving and deserializing incoming messages.

The second component of the reverse proxy is the firewall. The firewall in responsible for authentication incoming requests and ensuring that the client has permission to perform the requested actions. When a client connects to a frontend, the user provides login credentials to generate a unique id. This identifier is then sent with every request from the frontend so that the reverse proxy knows which client sent the message. A session store then stores a list of these unique identifier to ensure that only authorized clients can access the system.

Once the message has be authenticated, the router reads the messages and directs it to the appropriate microservice. Service discovery is also performed to get the network address of the microservice. The service registry stores this mapping and is queries to get the address.

Finally, the fifth component is the service clients, which communicate with the services API to forward the message.

Datastore

The Datastore is the a subsystem responsible for storing all entities in the virtual world.

Distributed Caching

Caching involves storing data in high-speed temporary storages mediums, such as RAM, to improve the speed of I/O operations. For example, accessing information from disk induces a latency penalty of around 5-20ms[26]. As each iteration of the game loop requires all entities to be read from memory, reading straight from disk would consume 12.5– 50% of each iteration. Reading data directly from RAM is much faster as the average latency is only a few nanoseconds[26]. However, as RAM is volatile, if the system shuts down then all data is lost. Our system had to be persistent, and therefore we would have to write to disk if. To combat this we decided that deploying a cache in front of the database, and using a write back

architecture to ensure all changes to the cache where saved to disk, would allow performant read/writes while still offering persistent storage.

A write-back cache stores all data within a fast in-memory cache and continuously write data to disk in the background. This offers persistent while still giving low data access times[27]. Within our system this was called shared memory and was going to be where all workers read and write from at the beginning and end of each game loop iteration. This would allow our workers to be stateless and make scaling the world significantly easier.

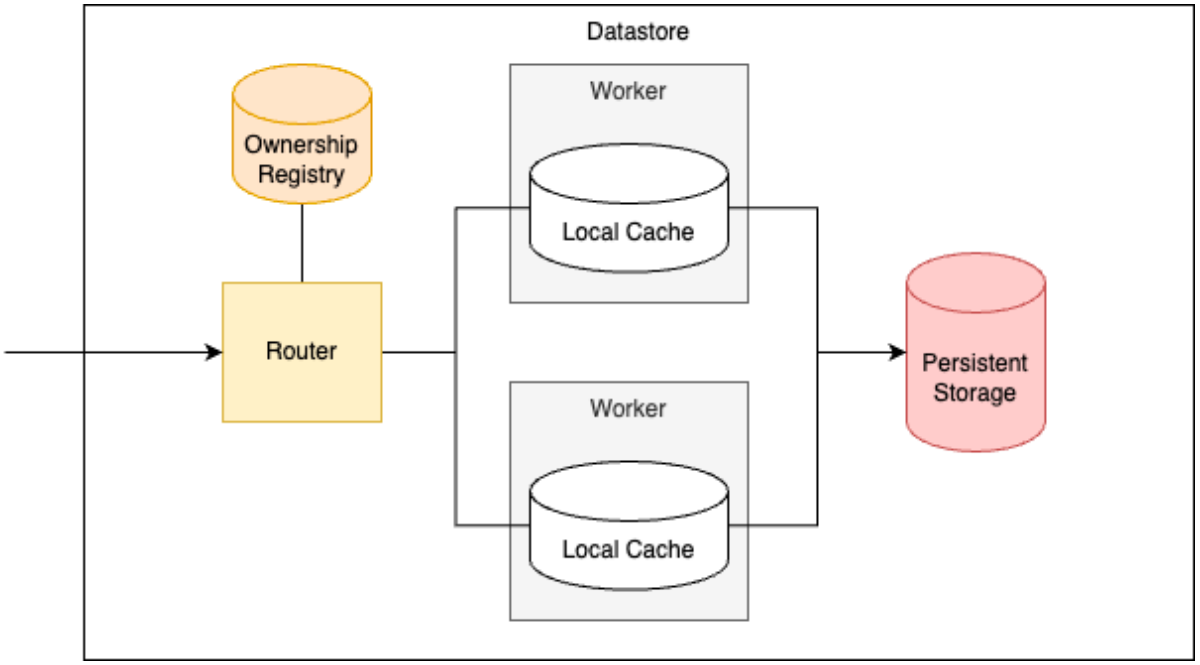
Unfortunately, when we built the shared memory model, the systems performance significantly decreased as the size of the virtual world grew. For example, when working with 100 entities, each game loop took only 200 microseconds to run. When we increased this number to 1000 entities, we saw an expected linear relation as the loop time rose to 2 milliseconds. As we increased the number of entities to 10,000, we expected the loop time to increase to 20 milliseconds however they rose all the way to 200 milliseconds.

During these tests Redis was used to store data in memory. As Redis does not support parallel write operations, updating the world would require may sequential writes each iteration. For example, a simulation of 10,000 entities would result in 480,000 sequential writes per second. A single Redis node was not able to handle this as Redis is single threaded and cannot be scaled vertically. Instead, clustering could be used to horizontally scale the number of Redis deployments.

Clustering is when we horizontally scale Redis instances and load balance incoming operations based on the key of the item[28]. After doing this we saw the linear performance decrease as expected.

Although we had shown that shared memory could scale to meet the systems demands, we were worried about the cost this would take to run the system. This benchmark performed by Redis and showed that it would cost 1 dollar per hour to run a Redis deployment capable of 5,800,000 QPS (Queries Per Second)[29]. From this, we calculated that a virtual world with only 10,000 entities would cost $100.97 per month just to run the shared memory. When looking at SpatialOS, many users said that the services were too expensive and therefore not

a feasible solution to the problem[30]. As a result, we wanted to explore the idea of in-process caching instead as this make sure of the Worker RAM instead of external servers.



In-process caching would require each worker to locally store all entities they are managing and write the back asynchronously to persistent storage. As each worker already has RAM, this method would add no extra cost to the system as well as increase system performance. The consequence however is that having stateful workers would significantly increase the complexity of the system. For example, how would we know which entity is being stored where, or how would be move entities from one worker to another?

Query Routing

Storing entities within workers means there is not a single location to query for data. For example, a client may wan to get all the entities within a 30m radius but how does it know which worker has the entities it needs. To abstract this logic from the rest of the system and prevent tight coupling, the datastore contains a router which takes all queries and performs them on behalf of the user. It uses the ownership registry to find out which worker has the entities it needs and requests the entities from the local cache of the relevant workers.

The introduction of a query router is like implementing an API gateway. By abstracting the local caches of each worker behind a single-entry point, datastore clients do not need to perform service discovery or reach to changes in how the worker caches are implemented.

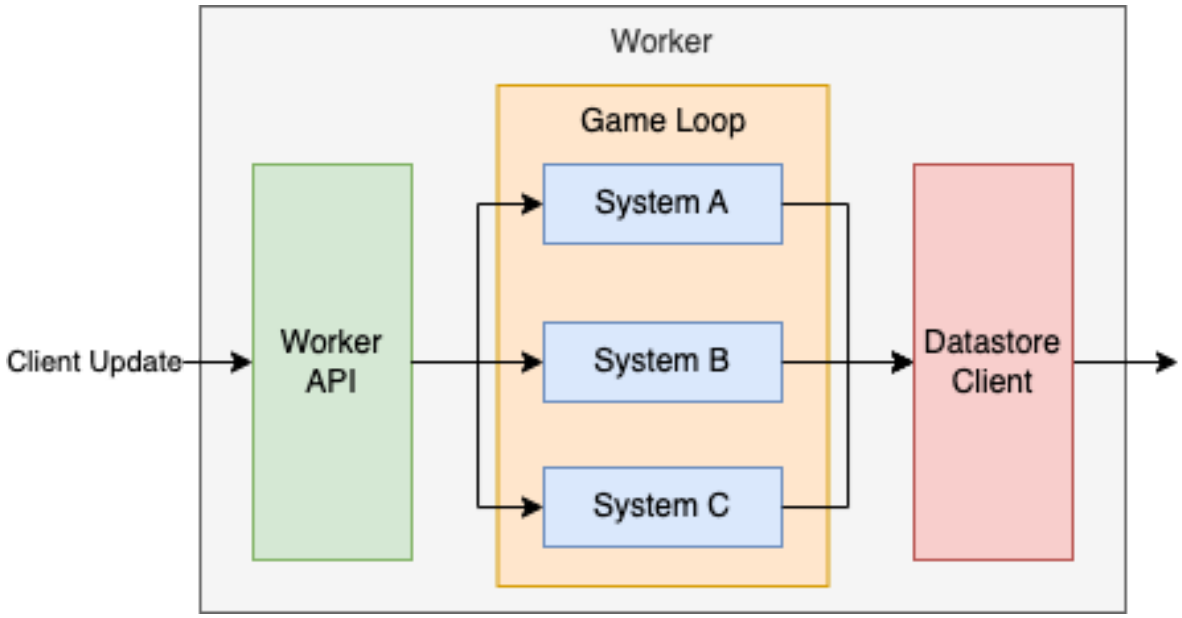
Ownership

Throughout the lifecycle of an entity, the worker in charge of managing It will change. For example, if Entity A originally is in a region managed by Worker 1, then it should be stored locally on Worker 1. However, if Entity A leaves this region and enters a new one managed by Worker 2, then Entity A should now be managed by Worker 2. In addition, if Entity B is on the edge of a region managed by Worker 1, then it might need to interact with Entity C in a neighbouring region managed by Worker 2. If so, Worker 1 will need a way to find out which Worker is managing Entity C.

The Ownership Registry stores this information and allows Workers and other services to know which Worker is managing an entity. The Ownership Status of an entity represents the relation between an entity and a worker.

Workers

The Entity Component System (ECS) is an example of Data-Oriented Design which is commonly used in games. For example, the Unity Game Engines DOTS is an implementation of an ECS which offers a significant performance increase compared to their native system[31]. The Entity Component System splits an application into three areas, Entities, Components and Systems. Entities represent objects within the game world, e.g., a Player. Components store the data related to each entity and use an ID to represent a relationship to an Entity. Systems then take in Components and perform data transformation on them.



Workers run a game loop at 24 iterations per second. During each iteration, a worker loads its

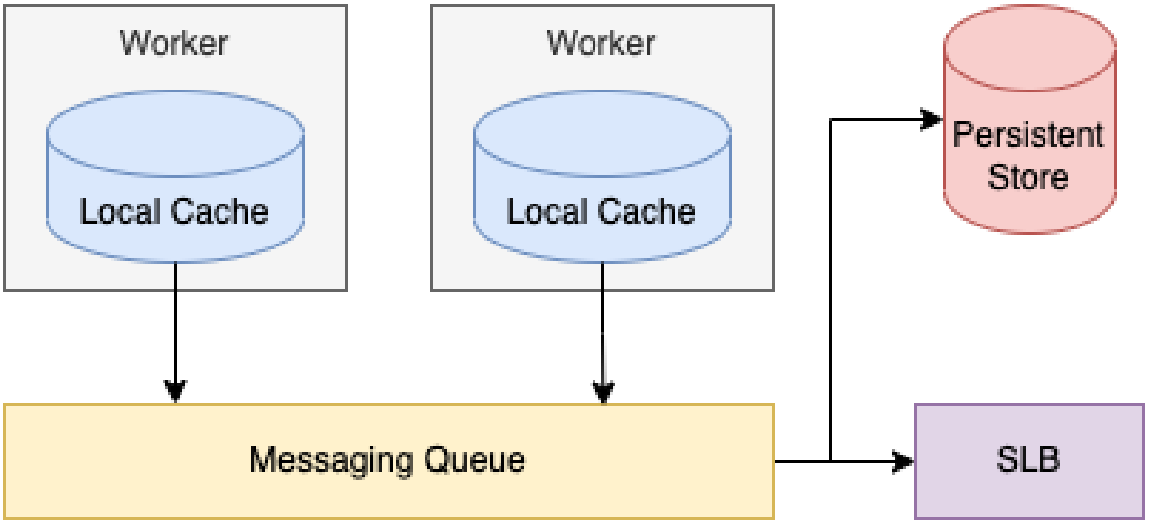
allocated entities from its local cache and runs systems to update each component attached to the entities. For example, every iteration, a worker might perform path finding information to find the next position that an AI agent should move to. Typically, each run of a system will modify a component in some way. The updated version of these components are then stored locally in the workers cache and written to the persistent store.

In addition, the Worker exposes an API which allows clients to send updates to workers. For example, if a user interacts with an item in the world, the update it then forwarded to the worker managing that entity and the system performs the desired action. This routing is performed by the Reverse Proxy and utilizes the ownership registry to identify where the update should be routed.

Horizontal scaling of workers is managed by Kubernetes' Horizontal Pod Auto-scaler. Kubernetes is a container orchestration tool to manage microservice-based system. A Horizontal Pod Auto-scaler is a component of the Kubernetes ecosystem that automatically updates the number of instances to match demand. It achieves this by intermittently running a control loop that compares a metric against a desired $value_{32}$. In our case, the metric used to determine the number of instances that should be deployed is the average time taken to execute the game loop. As mentioned before, each game loop must run in under 40ms to ensure a smooth, response user experience. As the average time approaches 40ms, more workers are deployed. As this is a custom metric, the Kubernetes' Metric API must be used to provide the current value. The Horizontal Pod Auto-scaler will the use the following equation to determine how many worker instances should be deployed:

desiredReplicas = ceil[currentReplicas * (currentMetricValue / desiredMetricValue)]

Work Distribution



Spatial Load Balancers are component within our system that monitor the activity of each region and each worker has been assigned an equal number of entities. SLB subscribe to all entity updates and uses this as a balancing metric.

The following algorithm is then performed to load balance regions across workers.

1. Create a Quadtree with a single node representing the entire world.
2. Select the leaf node with the largest number of entity updates.
3. Expand the node by creating 4 children nodes.
4. Repeat steps 2-3 until the difference between the minimum and maximum leaf is less than 10% of the total world entity updates.
5. Distribute the leaf nodes across the set of available workers.

When a region is assigned to a worker, the entities within that region need to be handed off form the old worker to the new one. This process is known as a Region Handover. To perform a Region Handover, we update the ownership registry so that the datastore clients can perform entity handovers until all the new regions entities are being updated by the worker.

SLBs communicate with workers through a TCP connection which is establish upon the creation of a worker. Although the communication between workers and SLBs would be more well suited for asynchronous messaging, such as a message broker, a TCP connection is

used to also provide health monitoring. As SLBs need to be aware of how many workers are available at all times, having TCP connections allow SLBs to respond to worker failure immediately. This ensures that if a worker failure occurs, the regions managed by the unhealthy worker are immediately reassigned to other healthy workers.

Minimum Viable Product (MVP)

At the start of the project an MVP was created as a platform to test and benchmark new ideas. As there were few examples or established best practices, we were faced with numerous design decisions and trade-offs to make. By building an MVP, we avoided investing too much time and resources in a single idea, and instead remain agile and responsive to the findings from our benchmarks. For example, originally all entities within the world were stored using a shared memory architecture. However, after conducting benchmarking, we realised that this approach would be too expensive and may not be feasible for the scale they were aiming for. This prompted the us to explore alternative implementations and ultimately led to a more effective and efficient design through in-process caching.

In addition, the MVP allowed the us to showcase a working example of the system and demonstrate that the suggested design elements were not only feasible but also effective for horizontally scaling spatial environments. By presenting a tangible example of the system in action, we were able to experience what the system would be like to for our users. This meant that we could identify improvements and future work that could be done. Overall, the MVP proved to be an invaluable tool for refining our ideas and demonstrating the potential of their system to others.

Technologies Used

Throughout the project, we made use of as many open-source technologies as we could. We wanted to ensure we were not reinventing the wheel and wanted to only focus one exploring new ideas.

Golang

The Go programming language was used to build most of the system. Go is an open-source programming language that was developed by Google in 2007. It is well known for being memory efficient and having great support for concurrency using goroutines and channel. A

goroutine is a lightweight threat which can be executed concurrently whole channels offer support for communication, as well as synchronisation between goroutines.

Datastore

When choosing a database for a project the CAP theorem is often used to decide what type of database is most suitable. The CAP theorem states that a database can only possess a maximum of 2 of the following traits: Consistency, Availability and Partition Tolerance[33]. A consistent Database ensures that all nodes have the same data simultaneous. For example, MongoDB ensures this by force all write operations to go through a master nodes[34]. Changes to this node are then pushed to all secondary nodes. An available database ensures that the database always remains operational. For example, this is often achieved through replication. Replication involves deploying several instances of a database which all changes are saved to such that if the main instance fails, the replicas are still able to serve requests[34]. This usually comes at the cost of consistent as ensuring all operations are written to replica nodes significantly slows down write operations[34]. Partition tolerance refers to how well a database can handle failures between its partitions. For example, if a node removes unresponsive, are the other nodes still able to perform.

As we could only priority 2 of the 3 mentioned traits, we had to decide what was most important for our system. Primarily, we wanted to focus on scalability so ensuring the database we selected has good partition tolerance was paramount. In addition, we wanted to ensure that the datastore was always available as if it became unavailable, the entire system would collapse. However, we also believed that consistency was very important as we wanted to ensure that all clients received an update to date and synchronised view of the world. As a result, we decided that partition tolerance and consistency were the most important as we knew we could still improve availability as a secondary focus.

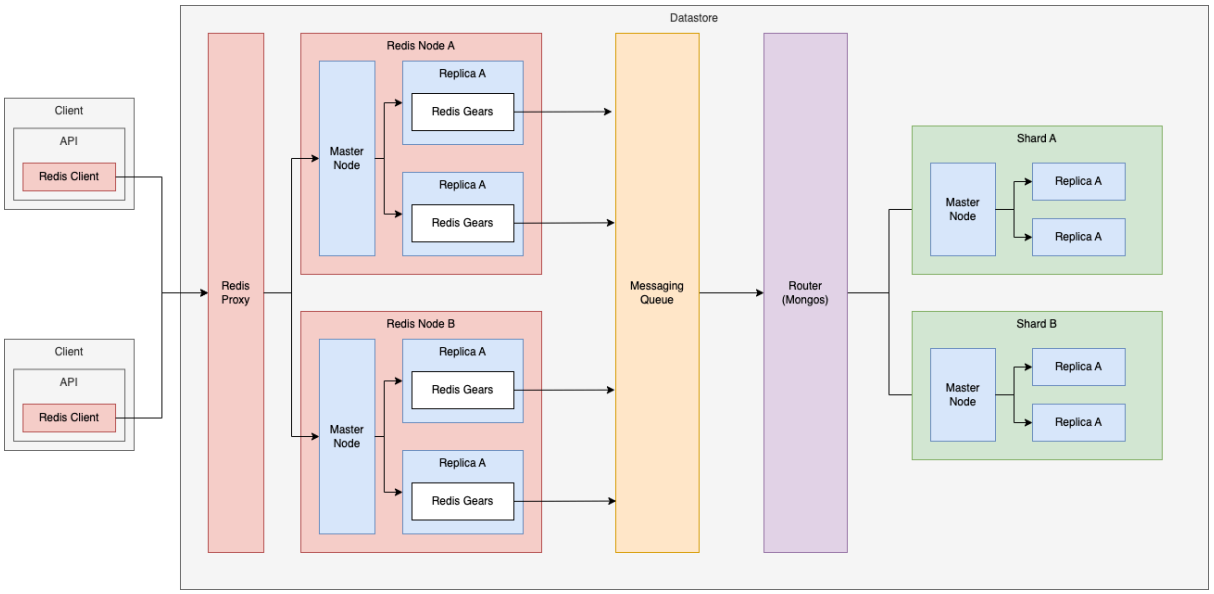
After narrowing down possible databases through the CAP theory, we began exploring uses two different databases MongoDB and Redis.

MongoDB is a document-oriented, no-SQL database which offers very fast read speeds due to utilization of indexes. Each item in a MongoDB is index based on a key, which in our case suits the ECS pattern as every entity is identified by a unique identifier. MongoDB stores each record in the form of BSON (Binary JSON) and there is schemeless. This offers more

flexibility than traditional SQL databases as each entity/object can have different attributes and fields. This also suits ECS well as different entities will have different components and by storing them as a single object, can be accessed without performing any join operations.

In addition, MongoDB can be configured to be highly available using replications. This allows use to deploy several secondary nodes which will serve as backups in case the master node fails.

Redis is an open-source in-memory key-value pair database. It is commonly used to implementing caching due to its speed and scalability. Redi also provides basic Pub/Sub messaging and is therefore sometimes used as a message broker.



To implement the datastore we wanted to use a combination of both Redis and MongoDB so that we could mitigate the disadvantages of each with the advantages of the other. For example, Redis was deployed as the primary read/write location and a wrote-back cache pattern was used to push all changes to MongoDB.

Redis Gears, a dynamic framework that enables developers to write and execute functions that implement data flows in Redis, was used to configure Redis as a write-back cache. The rgsync library offered ready-made Write-back recipes for Redis Gears via a python script. This made setting things up very easy. One thing that was done to improve the performance

of the system was to only perform the write-back on the replica nodes. This decreased the load on the primary node, improving system performance.

Using a write-back architecture also had the benefit of improving the availability of MongoDB. When the master node of a MongoDB deployment fails, an election process is performed to decide which of the replica nodes should become the new master. This process can take up to 12 seconds which during this time prevents reads and writes from the database. By making Redis the primary interface for reading and writing data, it meant that this election process was have no effect on the rest of the system, therefore providing added fault tolerant. Also, to prevent this effect the write-back mechanism, Redis Gears appended all changes to a message broker which meant that even if MongoDB was down, the messages were still there for when it became available again. Message were then routed via a Mongos server based on the entity ID.

Project Management

The scope of this project was large, involving the creation of a platform that provides horizontal scaling for virtual worlds. With pre-existing solutions such as SpatialOS, developed by Improbable, and Hadean having raised millions of dollars and employing hundreds of people, it was clear that this was a complex and challenging task. Given the limited timeframe of only 12 weeks, effective project management was crucial to ensure the project's success. It was necessary to carefully select what to implement and what to leave out to achieve the project's objectives within the given time frame. Therefore, this section will describe the project management approach used to ensure timely completion of the project while maintaining the quality of the deliverables.

Project management is an essential part of any project, and our horizontal scaling platform for virtual worlds was no exception. As we knew the scope of the project was significant, we recognized the importance of efficient and effective project management to ensure its successful completion. We chose Agile Development as our methodology as it aligned with our project's requirements.

Agile Development is a methodology that emphasizes the importance of quick iterations and consistent progress[35]. It is based on twelve principles, including satisfying the customer through early and continuous delivery of valuable software, welcoming changing

requirements, delivering working software frequently, and promoting sustainable development[35].

The main advantages of Agile Development include predictable scheduling and delivery, flexibility, and efficiency[35]. In our project, we wanted to minimize the risk of running out of time and not having anything to show during the demonstration. Agile's iterative nature helped us focus on delivering the most critical features first and prevented us from over-engineering a single feature. Additionally, since this was largely a design project, we knew that the requirements were likely to change frequently. Agile's flexibility suited us well as it allowed us to make changes and improvements in each iteration.
f
Furthermore, working efficiently was important as we had a limited timeframe to complete the project. Agile Development's focus on value helped us prioritize our work to maximize the end value while minimizing the amount of work we did. Having a predictable schedule was also crucial as it allowed us to track our progress accurately and monitor our risks better.

However, Agile Development also has some disadvantages, such as limited documentation and no finite end[35]. To mitigate these disadvantages, we adopted using design documents to provide us with documentation, which helped us during the final report writing process. We also recognized that not having a clear end goal was beneficial in our case as we knew it would be hard to predict what the final version of the project would look like and just wanted to work on the project for as long as possible.

In conclusion, Agile Development was the ideal methodology for our horizontal scaling platform project. Its advantages, such as predictable scheduling, flexibility, and efficiency, aligned with our project requirements, while we mitigated its disadvantages through various strategies.

Scope

When starting a project, it is important to define key objectives to outline what work will be included or excluded by the end of the project. This process is called scope management and runs throughout the lifecycle of a project. It starts by identifying the projects goals and objective and identify the key requirements to define success. Once large goals have been identified, they can then be broken down into deliverables, which are smaller more

manageable pieces of work. These can then be monitored throughout the project to track progress and ensure the project is completed within the allotted timeframe. As we knew the scope of the project was very large, it was important to understand which components of the project were most important so that we could focus our effort on the work which would produce the most value. Without this, we risked missing the projects deadline which would resulting in the projects failure.

User Stories

User stories are the primary method for defining scope with Agile[35, 37]. User stories are short goals written from the perspective of an end-user of the project. By focusing on what the user wants, it ensures the value is maximised throughout the project and provides a set of key motivations behind every feature[37]. This helps reduce the amount in a project as only the core requirements of user's are focused on.

A good user story has four key components. Firstly, a user stories should clearly define when the work is done[37]. For example, this is usually done by describe a specific action that the user should be able to carry out once the work is complete. By having a clearly outlined definition of what completion looks like, it prevent scope creep as it prevents overengineering features and adding excess functionality[37]. In addition, it ensures that feature are not underdeveloped and that we only move on once a user is satisfied.

Secondly, each user story should clearly outline the scope of the work that needs to be completed by identifying any tasks/subtasks. These should outline the specific steps that need to be taken in order to carry out the work. Not only does this provide a clear roadmap for how the user story should be complete, but it also provides a way to estimate the time and cost associated with each user story[37].

Thirdly, a user story should outline who the work will benefit by defining user personas. This describes what kind of user will benefit from the work as well as provide an indication for the motivation behind the feature, This helps better understand what the user's needs are as well what their expectations are37.

Finally, a user story should include clear deadline for when work on the feature should stop. Within Agile, work which extends past a deadline is halted and rescheduled at the end of a sprint. This prevents allocating too much time to a single feature and stops roadblocks effecting the duration of the entire project[37]. For example, it is better that a single feature is postponed rather than spending too much time on a single feature and delaying the entire project. Also, the 80/20 rule outlines that even if a feature is not fully complete, the initial work done is likely good enough to satisfy a customer and partially solve their problem. Therefore pausing further work on a feature can actually be beneficial by maximising overall user satisfaction.

To develop good user stories, we recognized the importance of gathering insights from industry professionals. We scheduled interviews with experts in the relevant field and prepared a series of questions designed to elicit the necessary information that could later be converted into user stories. These interviews allowed us to identify the key pain pints of potential users and identified short coming within existing solutions. As a result, it ensured we solely focused on developing new approaches to the problem rather than rebuilding functionality that would not meet user expectations. The user stories generated from these interviews were later used to define clear milestones throughout the project so that we could use to reflect on progress and ensure that the project was progressing steadily.

Features & Epics

In Agile project management, Epics and Features are important components in defining and managing the scope of a project[35, 36]. Epics are a high-level concept that represents a significant amount of work that needs to be completed in order to deliver a valuable outcome for the end user. Epics are typically too large to be completed in a single sprint and are often supported by a business case, which outlines the goals and objectives of the Epic[36]. An Epic can be thought of as a collection of related features or user stories that work together to deliver a specific outcome.

Features, on the other hand, are smaller chunks of work that are derived from Epics. They are individual deliverables that contribute to the completion of an Epic. A feature is typically something that adds value to the end user, and moves the project towards the completion of the Epic[36]. Features are often represented as user stories that describe the functionality or

behaviours of a software system from the perspective of the end user. Features can be completed within a single sprint and are typically the building blocks of an Epic[36]. Grouping user stories into features allowed for better organization and prioritization of work. By grouping related user stories together, we could identify which stories were dependent on each other and which ones could be completed independently. This provided a basis for deciding what the focus of each sprint would be and allowed us to plan and execute sprints more efficiently.

Requirements

A SMART objective is a goal-setting framework that is commonly used in project management to help ensure that objectives are well-defined and achievable. The acronym SMART stands for Specific, Measurable, Achievable, Relevant, and Time-bound.

The SMART framework was used within this project as during the design process, several difficult decision had to be made. For example, justifying the trade-off between complexity and latency. By defining clear and concise requirements, it meant that we could simply refer back to the main requirements of project and ensure whatever decision that was made maximised value and increased the likelihood of the project being successful

One weakness of SMART objectives is that they don't consider the relative importance of each objective. As a result, this makes is challenging to decide the order in which tasks should be completed within each sprint. To overcome this limitation, a secondary framework can be used in tandem with the SMART framework to provide a way to assign priorities to each task.

The MoSCoW method is a widely used prioritization technique for requirements in project management. It helps to assign priorities to requirements based on their importance and relevance to the project goals. The acronym MoSCoW stands for four different levels of prioritization: Must have, Should have, Could have, and Won't have. The Must have category comprises the highest priority requirements, while the Won't have category contains the least important requirements. This method is especially helpful for teams to organize and prioritize many requirements for a project, which can be overwhelming otherwise. The MoSCoW framework can easily be applied to SMART objectives by including the priority level with

the objective. For instance, a SMART objective could be "Develop and launch a mobile app with Must have features for iOS and Android platforms within six months." This makes it easy to intuitively identify the priority of each requirement and make informed decisions about which items should be tackled first, second, and so on. Using the MoSCoW method allowed us to fully adopt Agile principles and maximise value creation.

Timeline

Agile is an iterative framework where work is completed in increments called sprints. A sprint is a timeboxed period where a team focuses on a set of specific task, usually in the form of a feature. Each sprint is usually between one and four weeks long and comprises of four main stages: Planning, Implementation, Reviewing and Reflection. Sprint planning is a short phase in which the sprint's goals are refined and discussed to create clear and concise user stories. Also, this often includes assigning roles and responsibilities, or identifying what resources will be required to carry out the work. The implementation phase is when the proposed work is completed and is followed by a review phase where teams ensure that the work produce meets all requirements. Finally, a retrospective is performed to perform a self-assessment and identify what improvements can be made for the next sprint cycle.

As a primary objective of the project was to research and explore different technique for scaling virtual worlds, we need a platform which allow us to quickly iterate and benchmark different approaches. As a result, we initialise the project by build a Minimum Viable Product (MVP). After developing the MVP, we used our benchmarking results to plan the focus of each sprint, each addressing a different weakness within the system. In addition, by starting with an MVP, it meant that we would have a functional demonstration which we could use to present our finding and share our work with others. Furthermore, by using an iterative approach, it allowed us to remain flexible and adaptable in response to changing circumstances. For example, if our findings suggest that the priority of a feature should be increased, we could easily change the ordering in which sprints were carried to consistently ensure that the most important features would be completed first.

After identifying the focus of each sprint, we needed to plan ordering and duration for each sprint. By referring to our MoSCoW objectives and user stories, we were able to create a Gantt Chart to visualise the ordering and expected duration of each sprint. As a large part of each sprint involved exploring new concepts and testing different solution, it was difficult to

estimate the duration of each sprint. The Gantt Chart proved to be a valuable asset as it allowed us to monitor progress closely, ensuring that the chart was updated regularly after each sprint.

Risks

Identifying our projects risks was critical for the success of the project because it allowed us to proactively anticipate and mitigate potential issues before they occurred. For example, by identifying that the cost of cloud computing resources might prevent us from being able to build and test the system, we planned ahead and applied to various grants to ensure we could accumulate enough cloud computing credits to complete the project. As this process can take a lot of time, leaving it until we needed them would have set the project back and risked not completing the project within the 12-week deadline.

During the planning phase of our project, a technique called SWOT analysis was used to identify and mitigate risks. SWOT analysis is a planning tool used to identify strengths, weaknesses, threats and opportunities through a fact-based, data-drive process. By identifying these 4 key areas early on, we were able to utilises different opportunities to handle threats before they even became a problem. For example, we used open-source libraries and cloud computing resource to reduce the duration and cost of the project. This allowed us to reduce the severity of our risk, meaning less time had to be spend monitoring them therefore increasing our productivity through the project.

Also, during the review phase of each sprint, we reviewed the status of each risk and evaluated the likelihood of it occurring. By frequently monitoring our risks on a frequent basis, it meant that we could quickly react to any signals suggesting that our project was under threat and ensure that any issues were addressed in a timely manner

Evaluation

To evaluate the success of the project we referred to the original objectives which were set at the beginning of the project. This allowed to objectively evaluate the status of the project as well as identify key areas of improvements.

Objectives

- Systems components must show a linear relationship between performance and entity count.
    - **Status:** Completed
    - **Evidence:** To evaluate the status of this objectives, benchmarking had to be performed on various samples to show the relationship between entity count and system performance for each component. This was carried out by individually deploying each system component on a Virtual Machine deployed in the cloud. This ensure that each test was fair as each benchmark would have access to the same compute resources.
- The systems must automatically scale to meet demand in real-time using horizontal scaling to ensure an average game loop time of less than 40ms.
    - **Status:** Completed
    - **Evidence:** During a 24 hour benchmark, Horizontal pod auto scaling was used to ensure that the average loop time never exceed 32 milliseconds.
- Clients must be able to establish bidirectional communication to send and receive updates to and from the system.
    - The system should allow clients to subscribe to entity updates to synchronize their local representation of the world.
        - **Status:** Completed
        - **Evidence:** The MVP used this communication to visualise the state of the world.
- The system must persistently store the state of the world such that if the system shutdown, the world's state can be recovered.
    - **Status:** Completed
    - **Evidence:** The datastore wrote all data to MongoDB which ensured the data was stored persistent. Whenever Redis was started, it would load from MongoDB and recover the world state.
- The system must provide a high-level abstraction of the systems via a client.
    - **Status:** Partially Completed
    - Evidence:

- The system should be highly available and have 99.99% uptime when running for 24 hours.
    - **Status:** Completed
    - **Evidence:** The system was run for 24 hours in order to produce performance benchmarks. During this, the system achieved an uptime of 100% therefore meeting the requirement.
- The system could support a wide-range of different transport protocols to support different use-cases.
    - **Status:** Partially Completed
    - **Evidence:** The system's design does include this however the MVP never demonstrated using different protocols for communication.

Project Management

Going in to this project we knew that effectively managing the projects was paramount for its success. Overall, the project progressed smoothly and we achieved the primary objects of the project, however, during preparation for the project's presentation, not enough time was allocated to ensure the demonstration was working correct. As a result, it mean that the demonstration had to be omitted from the presentation. In reflection, we believe that this would have been more avoidable if more planning was done for the demonstration. As it was not fully part of the project, it was not accounted for during the planning stage of the project.

Overall, we believe that choosing an Agile development strategy was extremely effective and would definitely be used in future projects.

Limitations

At the start of the project, we had no experience building system of this nature. This lack of experience meant that we knew that we would learn new concepts and method for doing things throughout the project and would therefore lead to some limitations.

Primary, I think that the biggest limitation of the system is that the Go programming language was used to implement the datastore system. As Go uses a garbage collector, it is not well suited for projects that require lots of memory. As we chose to adopt a local caching system instead of a shred memory model, Go was no longer an effective language for the implementation of workers. As a result, I think that using a language such as Rust would be a

better choice as it would improve performance and is much more memory efficient. Rust also offers many similar benefits to Go, such as support for concurrency.

In addition, after seeing how expensive is would be to run the system with a shared memory model, we found Redis alternatives, such as DragonFly DB and Key DB, which promised much higher levels of performance. As a result, I think it would have been good to evaluate these different options and see if they were more cost effective than Redis. If so, this would have greatly reduced the complexity of the system.

Conclusion

To conclude, the work carried out within this project explored different approaches for building highly scalable virtual worlds through horizontal scaling. A design has been proposed which shows the desired characterises of a highly scalable spatial compute platform. When compared with existing solutions, our platform shows a considerable reduction in cost as well offering better performance. Since most of the existing solutions were built by companies with 100s of employees and millions in funding I am extremely proud of the accomplishments that we were able to make within the 12 week project. We plan to produce an open-source version of this product so that work can be continued and the ideas mentioned through this report can be explored further.

Future Work

Spatial DB

As a large portion of the project was working on an alterative for the shared memory model, we believe that more work can be done in developing an in-memory spatial database that is optimised for spatial queries. As virtual worlds grow in size and complexity, the need for such as database will increase. In addition, the database should support Pub/Sub communication so that clients can subscribe to regions in space.

References

1. Nevelsteen, K.J. (2017) “Virtual world, defined from a technological perspective and applied to video games, mixed reality, and the metaverse,” *Computer Animation and Virtual Worlds*, 29(1). Available at: https://doi.org/10.1002/cav.1752.

2. Zhang, H. *et al.* (2019) “CityFlow: A multi-agent reinforcement learning environment for large scale city traffic scenario,” *The World Wide Web Conference* [Preprint]. Available at: https://doi.org/10.1145/3308558.3314139.
3. Kurach, K. *et al.* (2020) “Google Research Football: A novel reinforcement learning environment,” *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04), pp. 4501–4510. Available at: https://doi.org/10.1609/aaai.v34i04.5878.
4. Richter, F., Orosco, R. and Yip, M. (2019) “Open-Sourced Reinforcement Learning Environments for Surgical Robotics.” Available at: https://doi.org/https://doi.org/10.48550/arXiv.1903.02090.
5. Osinski, B. *et al.* (2020) “Simulation-based reinforcement learning for real-world autonomous driving,” *2020 IEEE International Conference on Robotics and Automation (ICRA)* [Preprint]. Available at: https://doi.org/10.1109/icra40945.2020.9196730.
6. Dulac-Arnold, G., Mankowitz, D. and Hester, T. (2019) “Challenges of Real-World Reinforcement Learning.” Available at: https://doi.org/https://doi.org/10.48550/arXiv.1904.12901.
7. Alsop, T. (2022) *U.S. VR and AR users 2017-2023*, *Statista*. Available at: https://www.statista.com/statistics/1017008/united-states-vr-ar-users/ (Accessed: April 24, 2023).
8. *AR & VR - worldwide: Statista market forecast* (no date) *Statista*. Available at: https://www.statista.com/outlook/amo/ar-vr/worldwide (Accessed: April 28, 2023).
9. Witters, K.· J. (2009) *DeWiTTERS Game Loop*, *deWiTTERS*. Available at: https://dewitters.com/dewitters-gameloop/ (Accessed: April 30, 2023).
10. *Hadean Simulate Product Datasheet* (no date) *Hadean Simulate Product Datasheetturtl.co*. Available at: https://resources.hadean.com/story/hadean-simulate/page/4/1 (Accessed: April 22, 2023).
11. Koszela, J. and Szymczyk, M. (2018) “Distributed processing in virtual simulation using cloud computing environment,” *MATEC Web of Conferences*, 210, p. 04018. Available at: https://doi.org/10.1051/matecconf/201821004018.
12. Tailor, U. and Patel, P. (2016) “A Survey on Comparative Analysis of Horizontal Scaling and Vertical Scaling of Cloud Computing Resources ,” *International Journal for Science and Advance Research In Technology*, 2(6).

13. *AWS EC2 Instance Types* (no date) *Amazon Web Services*. Available at: https://aws.amazon.com/ec2/instance-types/.
14. Lunin, P. (2022) *Network latency: How latency, bandwidth & packet drops impact your speed*, *Scaleway*. Available at: https://www.scaleway.com/en/blog/understanding-network-latency/ (Accessed: April 15, 2023).
15. *What is Edge Computing & Why is it important?* (no date) *Accenture*. Available at: https://www.accenture.com/us-en/insights/cloud/edge-computing-index#:~:text=Edge%20computing%20is%20an%20emerging,led%20results%20in%20real%20time. (Accessed: April 12, 2023).
16. *What is MTU & MSS: Fragmentation explained* (2022) *Imperva*. Available at: https://www.imperva.com/learn/application-security/what-is-mtu-mss/#:~:text=Maximum%20Transmission%20Unit%20(MTU),-MTU%20is%20the&text=The%20internet's%20transmission%20control%20protocol,each%20packet%20in%20any%20transmission. (Accessed: April 12, 2023).
17. K. Maeda, "Performance evaluation of object serialization libraries in XML, JSON and binary formats," 2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP), Bangkok, Thailand, 2012, pp. 177-182, doi: 10.1109/DICTAP.2012.6215346.
18. *Introducing json* (no date) *JSON*. Available at: https://www.json.org/json-en.html (Accessed: April 16, 2023).
19. O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, 2018, pp. 000149-000154, doi: 10.1109/CINTI.2018.8928192.
20. Newman, S. (2015) *Building microservices*. Sebastopol: O'Reilly Media, Inc, USA.
21. Karabey Aksakalli, I. *et al.* (2021) “Deployment and communication patterns in microservice architectures: A systematic literature review,” *Journal of Systems and Software*, 180, p. 111014. Available at: https://doi.org/10.1016/j.jss.2021.111014.
22. G. Xylomenos and G. C. Polyzos, "TCP and UDP performance over a wireless LAN," IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320), New York, NY, USA, 1999, pp. 439-446 vol.2, doi: 10.1109/INFCOM.1999.751376.

23. Bernier, Y.W., 2001, March. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference* (Vol. 98033, No. 425).
24. *Microservices pattern: API gateway pattern* (no date) *microservices.io*. Available at: https://microservices.io/patterns/apigateway.html (Accessed: April 11, 2023).
25. *1 million is so 2011* (no date) *WhatsApp.com*. Available at: https://blog.whatsapp.com/1-million-is-so-2011 (Accessed: April 22, 2023).
26. Gilheany, S. (2003) *Ram is 100 thousand times faster than disk for database access*, *Directions Magazine - GIS News and Geospatial*. Available at: https://www.directionsmag.com/article/3794 (Accessed: April 15, 2023).
27. Contributor, T.T. (2012) *What is write back?: Definition from TechTarget*, *WhatIs.com*. TechTarget. Available at: https://www.techtarget.com/whatis/definition/write-back#:~:text=Write%20back%20optimizes%20the%20system,crash%20or%20other%20adverse%20event. (Accessed: April 3, 2023).
28. *Scaling with Redis Cluster* (no date) *Redis*. Available at: https://redis.io/docs/management/scaling/ (Accessed: January 8, 2023).
29. Lau, G. (2023) *Redis Enterprise and google cloud T2D benchmarks*, *Redis*. Available at: https://redis.com/blog/redis-enterprise-google-cloud-t2d-benchmarks/ (Accessed: April 11, 2023).
30. A bigger cost failure is spatial OS, from improbable. they spent about $400M of ...: Hacker news (no date) A bigger cost failure is Spatial OS, from Improbable. They spent about $400M of ... | Hacker News. Available at: https://news.ycombinator.com/item?id=27310655 (Accessed: April 11, 2023).
31. Bayliss, J.D. (2022) "The data-oriented design process for game development," *Computer*, 55(5), pp. 31–38. Available at: https://doi.org/10.1109/mc.2022.3155108.
32. *Horizontal pod autoscaling* (2023) *Kubernetes*. Available at: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (Accessed: April 14, 2023).
33. S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem," in Computer, vol. 45, no. 2, pp. 30-36, Feb. 2012, doi: 10.1109/MC.2011.389.
34. *MongoDB - Automatic Failover* (no date) *Replication - MongoDB Manual*. Available at: https://www.mongodb.com/docs/manual/replication/#automatic-failover (Accessed: April 23, 2023).

35. *Agile Manifesto* (no date) *Principles behind the Agile Manifesto*. Available at: https://agilemanifesto.org/principles.html (Accessed: April 11, 2023).
36. *Epics features and stories - University of Glasgow* (no date). Available at: https://www.gla.ac.uk/media/Media_730149_smxx.pdf (Accessed: April 29, 2023).
37. Atlassian (no date) *User stories: Examples and template*, *Atlassian*. Available at: https://www.atlassian.com/agile/project-management/user-stories (Accessed: March 29, 2023).
38.