Distributed Real-Time Physics for Scalable and Streamed Games and Simulations

Alexander James Ronald Brown

School of Computing
Newcastle University

This thesis is submitted for the degree of
Doctor of Philosophy

November 2020

I would like to dedicate this thesis to my loving parents and partner.

Acknowledgements

I would like to thank my supervisor Dr. Graham Morgan for his continuous support and guidance throughout my PhD study and research. I would like to acknowledge Dale Whinham for his improvements to the build and deploy system used in this project and for his conversion of some of the diagrams from the office whiteboard into InkScape. I would also like to acknowledge Carlos Guerrero Rodriguez for his contributions under my guidance to the visualiser, including the client-created replicas and static objects, client-defined server regions and reset features.

My sincere thanks to my parents and partner for their excellent proof-reading skills and support.

Abstract

In this study, a solution to delivering scalable real-time physics simulations is proposed. Although high performance computing simulations of physics related problems do exist, these are not real-time and do not model the real-time intricate interactions of rigid bodies for visual effect common in video games (favouring accuracy over real-time). As such, this study presents the first approach to real-time delivery of scalable, commercial grade, video game quality physics and is termed Aura Projection (AP). This approach takes the physics engine out of the player's machine and deploys it across standard cloud based infrastructures. The simulation world is divided into regions that are then allocated to multiple servers. A server maintains the physics for all simulated objects in its region. The contribution of this study is the ability to maintain a scalable simulation by allowing object interaction across region boundaries using predictive migration techniques. AP allows each object to project an aura that is used to determine object migration across servers to ensure seamless physics interactions between objects. AP allows player interaction at any point in real-time (influencing the simulation) in the same manner as any video game.

This study measures and evaluates both the scalability of AP and correctness of collisions within AP through experimentation and benchmarking. The experiments show that AP is a solution to scalable real-time physics by measuring computation workload with increasing computation resources. AP also demonstrates that collisions between rigid-bodies can be simulated correctly within a scalable real-time physics simulation, even when rigid-bodies are intersecting server-region boundaries; demonstrated through comparison of a distributed AP simulation to a single, centralised simulation. We believe that AP is the first successful demonstration of scalable real-time physics in an academic setting.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Real-Time Physics

Physics has been a part of video games for almost as long as video games have existed [57]. Real-time physics is necessary to provide realistic 2D and 3D environments to players as it allows players to interact with the simulated environment. For example, moving crates around, simulating the flight of a plane or the ballistics of a bullet. Physics engines are responsible for all of the calculations required to simulate physics. Physics engines for games aim to achieve ‘realism’, but sacrifice accuracy in order to achieve real-time speeds. Recently, the mathematical error from commercial physics engines has been reduced and physics engines are finding use in other simulations besides video games ([90, 48, 77]).

1.2 Scalable Games and Simulations

Scaling and more complex games are a selling point. This has been demonstrated by games such as EVE Aether Wars [37] and Worlds Adrift [11], built using HadeanOS and SpatialOS respectively. HadeanOS and SpatialOS both provide attempts at distributing physics by allowing physical interactions across servers. Despite the growing commercial interest in this area, there is no academic literature into the challenges and solutions of distributed real-time physics and HadeanOS and SpatialOS are both proprietary software.

Research into delivering scalability in such online worlds is focused on balancing real-time and consistency issues for enabling player-player interaction. The evolution of such research can be traced back to the earliest work on game area subdivision [49] through player-focused game area sub-division ([61, 35]) and eventually to the many commercial cloud based solutions as described by [89].

Traditionally real-time simulations are confined to a single machine, including real-time physics. Recent cloud technology has made available lower cost, low-latency and high-performance machines, including GPU-enabled instances. This means, real-time simulations can now run in the cloud as well as being economically viable. It is possible to rapidly provision resources on demand, allowing for fast scaling up and down of resources as needed.

Physics is the most challenging problem facing distributing games engines. Handling player-player interaction between servers, which is still not a solved problem, requires response times in the order of hundreds of milliseconds and still leads to inconsistencies in the game world. Real-time physics solvers work in iterations normally measured in the order of less than 10 milliseconds. Although this poses a significant challenge, achieving distributed real-time physics would allow the physics engine component of the game engine to be elastically scaled, meaning the physics engine is no longer limited in scale and complexity to what would be achievable on a single machine.

The potential uses of scaling real-time physics simulations are vast and include large world, physics-based games and games relying heavily on physics queries (often used by AI). Scaling real-time physics is useful not only for games, but large-scale simulations where querying or simulating the physical state of the world is heavily used, e.g. large-scale multi-agent AI, such as city simulations and military simulations.

Server authoritative physics would also allow over the network multiplayer games to rely heavily on physical elements. In games, typically only a limited part of the game is simulated on the server side and not physics, with focus on player-player interaction and not player-environment (physics) interaction, as this is a limit of existing technology. Synchronising physics states between players and servers is not possible due to latency, two players will have different views of the physical state of the world, based on their latency, interactions from those players with the physical state will cause divergence, which may not be possible to converge. This is normally worked around by not having physics affect gameplay and using kinematic sequences, ensuring determinism. A single physics simulation does not encounter these problems of multiple physics states, which have to be synchronised and can lead to divergent physics states.

Simulating the physical state of the world entirely in the cloud frees up resources on the user's machine allowing for more processing time to be spent on other systems, such as graphics and AI. Alternatively, simulating physics in the cloud could be combined with streamed gaming, by treating real-time physics as a microservice.

1.3 Streamed Gaming

Streamed gaming is an active area of research (e.g., [32]), where the game is executed on the server-side, the graphical output is streamed to a remote player and input is sent from the player to the server-side. Streamed gaming reduces the computing power requirements on player hardware, even allowing gaming on mobile devices with graphical quality beyond what would be possible if the game was running entirely on the device [79]. In addition, this removes the need for players to continuously upgrade their hardware [16].

The most difficult challenge in streamed gaming is the quality of experience of games, as the experience is significantly impacted by delay[18]. The effect on quality depends upon the time of game, for example low-latency games like first person shooters require response times as little as 100ms, whereas, real-time strategy games can have acceptable response times of up to 1000ms [79]. Recent advances in cloud gaming, including the use of edge cloud game servers, have enabled the cloud to be utilised to deliver low-latency gaming [92].

Examples of streamed gaming services include Sony's PSNOW [80], NVidia's GameStream [67] and Google's Stadia [34]. The existence of such services suggests the possibility of real-time interaction for all gaming genres across cloud infrastructures.

1.4 Summary

Real-time physics is an integral part of games technology. There has been recent commercial interest in using cloud computing to scale up games, including the most challenging aspect, the physics simulation component. In addition, scaling up physics simulations provides an increase in possible size and complexity of virtual worlds used in online games (a greater number of simulated entities and an increase in systems that are expensive to simulate). If achieved, real-time physics engines can be scaled beyond what is currently possible on a player's machine or single server and provides elastic scaling of resources for use in streamed gaming. Despite the recent interest and advantages, no academic literature (with the exception of [12] based on the work of this study) addressing scalable real-time physics has been produced, to date.

1.5 Purpose of Study

The purpose of this study is to test the following hypotheses: "Can scalable real-time physics be achieved?" and "Can real-time physics simulations remain correct when scaled?". This study is intended, for the first time in an academic setting, to demonstrate scalable real-time physics. The requirements needed for a solution to scalable real-time physics are discussed and several different approaches are considered before justifying the approach taken in this study's proposed solution. A prototype solution to scalable real-time physics, Aura Projection (AP), has been developed for this study and has provided a means of testing the above hypotheses through experiments that have been put forward by this study. AP was first presented in [12]. AP uses PhysX [66] (from Nvidia) as the physics engine, providing a simulation with equal accuracy and behaviour to commercial video games. The main challenge addressed by AP is to deploy multiple instances of PhysX in the cloud, across multiple servers and through the use of messaging services, while allowing objects within the physics simulation to be seamlessly passed and interact across server-region boundaries. Achieving interactions of objects across server-region boundaries is a complex problem to solve and small errors can easily be discerned by the player in the form of 'jitter'. In some cases errors can lead to great anomalies in behaviour and results in servers having significantly different states for the same set of objects.

1.6 Outline

The following section provides an outline for this study:

1. Background and Related Work:

    - Real-time physics engines overview. The reader is introduced to the topic of physics simulations and real-time and non-real-time physics simulations are compared. The following are defined: terms; workings; important and relevant aspects, e.g. body types; uses in games and simulations; typical performance of real-time simulations; and performance budget in real-time applications.
    - Scalable Non-Real Time Physics. Previous work and applications of scalable non-real time physics and how those techniques are not suitable for use in real-time physics simulations are discussed.
    - Online gaming. The wider topic of online gaming is discussed, including the main challenges faced and typical architectures used (client-sever, peer-to-peer, distributed servers). Important aspects to this study's work are also discussed, specifically, typical server frame-times and network tick updates, delta-encoding and protocols.
    - RakNet. A brief overview is given of the important features of RakNet, the network library used in this study's implementation (AP).
    - Consistency in Real-Time Physics. The challenges of consistency in physics are addressed. The client-side compensation technique of dead-reckoning is discussed and how it creates consistency problems.
    - Distributed virtual environments. Existing ways of distributing game worlds are reviewed and previous work carried out on these approaches. These fall into two categories, migratory (approach used by AP) and non-migratory. The studies found non-migratory to be the better solution when focusing on player-player interaction and why migratory is the better solution for physics body-physics body interaction will be discussed. Important concepts such as interest management are introduced to the reader.
    - Distributed Real-time Physics. Previous work looking specifically at distributing real-time physics will be reviewed and how AP differs from these approaches.
    - Cloud Computing. The wider context of cloud computing will be discussed and how AP is deployed on the cloud and also specifically conditions that affect AP in the cloud, i.e. latency and packet-loss
    - Real-time Physics for Scalable Simulations. Real-time simulations are being used more in scientific research, however these are limited to what can be achieved on a single machine. AP allows real-time physics simulations to be scaled, enabling large or complex simulations that otherwise wouldn't have been able, to run in real-time.

- Microservices for Game Engines. Game engines, generally, currently run as one single application (monolith), including graphics, physics, AI etc. (with the exception of databases for user data). AP separates out the physics element into its own microservice, much like many modern systems are splitting up different aspects of systems into their own microservices, for example web applications. This separation of physics into a microservice means physics can be elastically scaled independently from other systems such as graphics, along with other advantages of modularity.
- Streamed Gaming. The main foreseeable application of AP is Streamed Gaming, with the use of large multiplayer worlds, that are persistent and scalable with dedicated and elastic physics nodes. Through the use of AP physics is no longer limited to what can be simulated on a single machine.

2. Problem Definition:

   - Solution Requirements. What is required of a solution to scalable real-time physics is discussed.
   - Naive approach. How would a naive approach behave and what problems occur.
   - Considered Solutions. A number of solutions to scalable real-time physics are considered.
   - Proposed Solution. How AP works is looked at in more detail and justifications are made for its choice.
   - Aura Calculation. A detailed look at the important aspect of AP - the aura calculation.
   - Sub-optimal object hosting. One the predicted challenge with AP is looked at: sub-optimal object hosting and a solution which can be addressed in future work is discussed.
   - Thrashing. A further predicted challenge with AP, this consists of two types of thrashing, two and three object thrashing, three object thrashing has been solved in this implementation of AP and solutions to two object thrashing are discussed.
   - Islands. A solved challenge with AP. AP's solution to islands is discussed.
   - Corner Case. AP has been discussed in terms of a single boundary between two servers up to this point, but in order for a general solution to be achieved, AP must be able to support boundaries between three or more servers, such as at corners.

3. Implementation

   - System Architecture. How the deployment system is structured. How AP interacts with PhysX. How messages are exchanged between servers using RakNet and how the client communicates with the servers, enabling interaction.
   - Aura Implementation. A high-level overview of how the auras are implemented in AP.

- Algorithms. The algorithms used by AP are defined with pseudo-code and explained.
- FSM. A finite-state-machine representation of the state of an object in AP is presented.
- The Visualiser. The features and implementation of the visualiser are discussed, including server and client created replicas, static objects and interactivity. The challenges of multiple servers are explained and the solutions used are justified. Screenshots from demo scenes have been included.

4. Experiments and Results
    - Scalability Experiments. The design of experiments designed to address the first question of this thesis: "Can scalable real-time physics simulations be achieved?" are described and justified. AP's performance is measured with different numbers of servers in different topologies. The results of which, are analysed and discussed.
    - Collisions Correctness Experiments. Collisions correctness is defined and erroneous collisions are classified. The design of experiments designed to address the second question of this thesis: "Can real-time physics simulations remain correct when scaled?" are described and justified. The correctness of collisions are measured using different conditions and tolerances. The results of which, are analysed and discussed.
5. Conclusion and Future Work
    - Conclusion. The results and findings of this study's experiments are used to answer the questions asked in this thesis: "Can scalable real-time physics simulations be achieved?" and "Can real-time physics simulations remain correct when scaled?"
    - Future Work. Future possible research into scalable real-time physics is discussed and future implementation and experimentation plans for AP are described.

Chapter 2

Background and Related Work

In this chapter, the reader is introduced to physics simulations and the differences between real-time and non-real-time simulations along with important terms, workings, important and relevant aspects of physics simulations. Along with an introduction to physics simulations, the following points will also be discussed:

- Scalable non-real-time physics
- Online gaming
- RakNet
- Consistency in real-time physics
- Distributed virtual environments
- Distributed real-time physics
- Cloud computing
- Real-time physics for scalable simulations
- Microservices for game engines
- Streamed gaming

2.1 Real-time Physics Engines Overview

Real-time physics simulations (physics engines) are responsible for simulating the physical behaviour of objects and are based around Newtonian mechanics. The motion of the system is calculated given the forces acting on the system and this is known as the forward dynamics problem [10].

The main three aspects of the functionality of a physics engine are the following:

- Integration: Moving objects according to a set of physical rules and values i.e. velocity, drag, friction

- Collision Detection: Checking for collisions between two or more objects
- Collision Response: The reaction to collisions between objects

The term ‘object’ is used here to describe any physical entity. Objects can be any entity that physically interacts, for example, a barrel, a bullet, a fire particle, a jointed robot arm, the wheel of a vehicle, a mountain etc. The terms ‘world’ or ‘scene’ are often used to describe the physical system.

The numerical integrator used to update the positions of objects in the system, uses a discrete time value. Each discrete time value is known as a time step. Each step simulates a set period of time. The shorter the time period simulated, the more physically accurate the simulation will be. However, each simulation step must take less time to execute than the time it is intending to simulate, otherwise the simulation would be unable to run in real-time. It is common for the time step to be a fixed value, though physics engines can support variable time steps.

2.1.1 Linear Motion

Linear motion will first be discussed and the following variables are used:

- F - Force
- m - Mass
- a - Acceleration
- v - Velocity
- s - Displacement
- n - Current time step
- $n+1$ - Next time step

Acceleration is calculated from the sum of the forces acting on the body and with the use of the mass of the objects and Newton’s Second Law:

$$F = ma \tag{2.1}$$

The simplest numerical integrator is explicit Euler (or sometimes referred to simply as Euler) integration:

$$\begin{aligned} v_{n+1} &= v_n + a_{n+1}\Delta t \\ s_{n+1} &= s_n + v_n\Delta t \end{aligned} \tag{2.2}$$

However, this is not often used by physics engines as it can lead to instability unless short time steps are used.

A symplectic Euler (semi-implicit Euler) integrator is often used in physics engines. Symplectic Euler is similar to explicit Euler, except that the updated velocity is used before calculating the new position:

$$\begin{aligned} v_{n+1} &= v_n + a_n \Delta t \\ s_{n+1} &= s_n + v_{n+1} \Delta t \end{aligned} \tag{2.3}$$

Symplectic Euler is faster to compute and remains accurate over many iterations and is stable when longer time steps are used.

2.1.2 Angular Motion

In addition to solving linear motion, physics engines also solve angular motion. Angular motion will now be discussed and the following variables are used:

- T - Torque
- I - Inertia Matrix or Inertia Tensor
- α - Angular acceleration
- ω - Angular velocity
- θ - Orientation
- n - Current time step
- $n+1$ - Next time step

Orientation of objects are represented using Quaternions. Torque is calculated using the following equation:

$$T = I\alpha \tag{2.4}$$

This equation is the rotational equivalent of $F = ma$. Angular velocity ω can be integrated relative to angular acceleration α using the following equation:

$$\omega_{n+1} = \omega_n + \alpha_n \Delta t \tag{2.5}$$

Orientation θ can then be calculated using ω and the following equation:

$$\theta_{n+1} = \theta_n + \theta_n \omega_n \frac{\Delta t}{2} \tag{2.6}$$

This equation is a fast approximation of orientation.

2.1.3 Integration comparison with non-real-time simulations

The equations used in real-time physics engines use iterative processes and approximate a continuous process with discrete steps. As a result, errors due to inaccuracies in the discreteness accumulate over time due to the iterative nature of the integrators. Different real-time physics engines will often give different results for the same simulation, despite identical starting parameters [10].

2.1.4 Collision Detection

Objects are represented by geometry colliders. Collision detection works by detecting overlaps between geometry representations (contact determination). Typically objects have simple colliders, such as boxes, capsules, cylinders and spheres. More complex colliders are often supported, such as convex and concave hulls. Colliders can either be static or dynamic.

Optimisations for collision detection are used to avoid comparing every possible pair of objects. Phases of collision detection are used, known and the broadphase and the narrowphase.

In addition to informing the collision response phase, callbacks can also be generated, allowing any part of the software to be informed of collision events that have taken place. For example, the game logic may receive events of when a bullet hits an avatar in order to determine when a player has been shot in a shooting game.

2.1.5 Collision Response

Collision response is the phase of the real-time physics loop which deals with integrating into the model the effect produced by contacts between entities within the system. Collision response will take into account mass, coefficient of restitution (loss of kinetic energy during a collisions) and the location and angle of the collision. Collision response can be very simple to calculate for simple geometric shapes, but becomes expensive when more complex mesh representations are used.

2.1.6 Performance of Real-Time Physics Engines

In this section the performance of real-time physics engines will be discussed. Examples of different scenarios under different performance conditions will be used and discussed using time-sequence diagrams.

Physics engines are kept as separate as possible from the rendering loop and other logic in the main loop. Fig. 2.1 shows a time sequence diagram of a game loop with good performance. The target physics time step is 16ms. Every 16ms that is passed, a physics update is triggered, simulating 16ms of the simulation. Multiple update loops can be completed between physics steps being performed.

Fig. 2.2 shows a time sequence diagram of sub-stepping. If an update loop takes longer than the target time-step (in this case 16ms), multiple physics steps will need to be performed to update the simulation to the present time. This is known as sub-stepping. Sub-stepping is supported by PhysX. Longer update times may occur in many situations, such as a read from

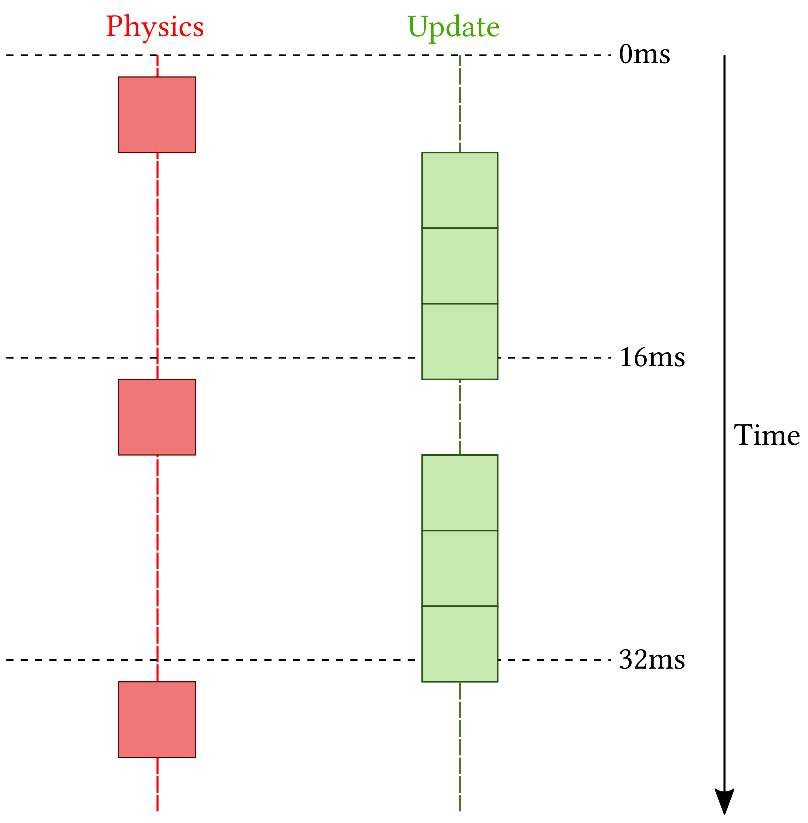


Fig. 2.1 Time sequence diagram of good performance.

the hard-drive, a background process performed by the OS or particularly expensive game logic or AI task.

Fig. 2.3 shows a time sequence diagram of a scenario in which the physics simulation is expensive to step. This could be the result of a large number of objects being simulated or complex collision detection. As a result of the expensive physics update the render frame rate is reduced to 60Hz, despite the main update only taking $3ms$.

2.2 Scalable Non-Real Time Physics

Scalable non-real time physics simulations include fluid simulations, meteorological simulations and accurate physical simulations e.g. robotics, wheeled and tracked vehicle dynamics and mechatronics. An example of a scalable non-real time physic is Project Chrono [82]. Non-real time physics simulations work very differently from real-time, accuracy is highly favoured as opposed to the fast, plausible simulations in real-time physics. In addition high latency (10s of ms) is a small issue relative to large time-steps used in such simulations.

2.3 Online Gaming

In this section an overview of online gaming challenges and architectures will be discussed. The main challenges facing online gaming are consistency, latency, fault-tolerance, fairness, cheating, and scalability. These will all be briefly explained, although they are not all the

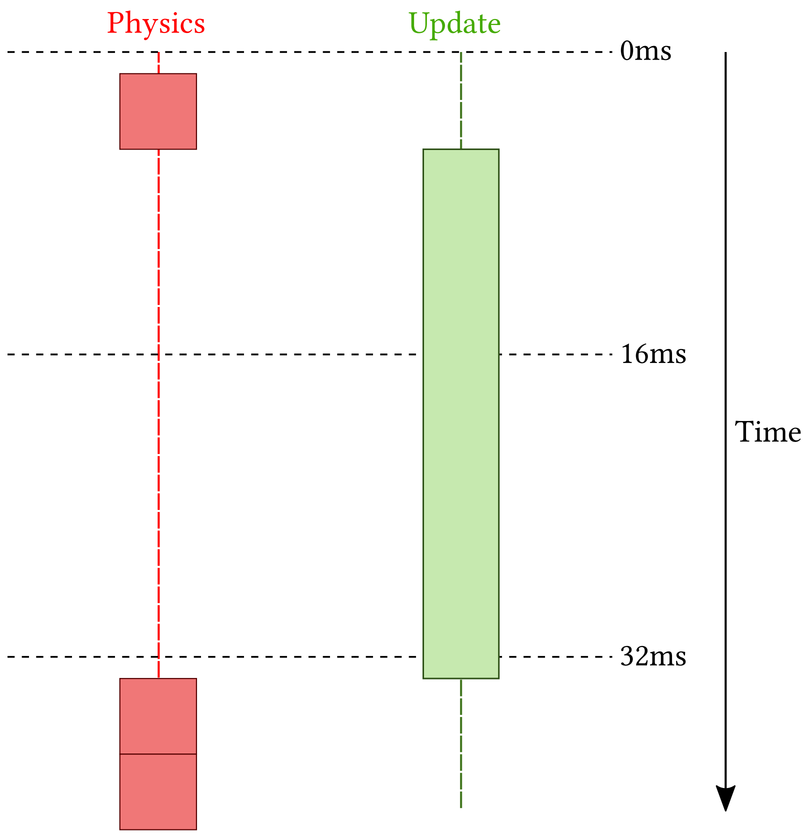


Fig. 2.2 Time sequence diagram of sub-stepping.

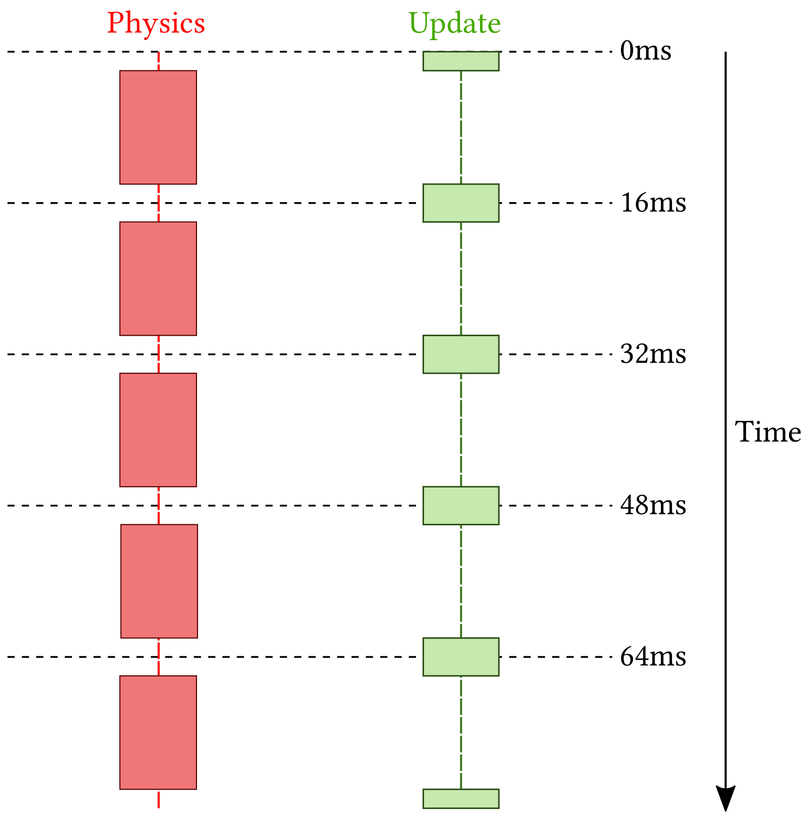


Fig. 2.3 Time sequence diagram of expensive physics.

main focus of this study, this is intended to make the reader aware of the challenges different architectures address. Finally, important aspects of online gaming that are relevant to this study are also discussed. The purpose of this study is to demonstrate scalability and consistency of real-time physics through the use of a distributed server architecture. In later sections, consistency will be defined in greater detail with respect to real-time physics and online gaming and existing work addressing scalability and consistency will be discussed.

2.3.1 Consistency

As online games are distributed (either between clients and a server, peers or multiple servers), inconsistent states can occur. Multiple peers/servers execute updates in parallel, however if these are executed out of order, such as due to messages being received out of order or messages lost entirely, inconsistent states can arise. Different aspects of game state require different levels of consistency, e.g. player life-death decisions require a high degree of consistency, whereas movements of other players have little effect on gameplay (in games like World of Warcraft) and so only require a low degree of consistency [76]. In addition, games often employ consistency resolution instead of inconsistency prevention/consistency maintenance [91].

Four categories of algorithms for consistency maintenance techniques for networked games are identified in [76]. The categories are as follows:

- Delay the processing of local events. Examples include Bucket Synchronisation [33] and Local Lag [51], in which local actions are grouped into buckets of time or delayed to allow for simultaneous execution on all clients.
- Alter the time frame for remote entities e.g. Remote Lag [5], in which local actions are applied immediately, but remote actions are delayed, allowing for interpolation (instead of relying on prediction), and Local Perception Filters [78], a refinement on Remote Lag, in which, instead of applying a fixed delay to remotely controlled entities (as in Remote Lag), a variable delay is applied determined by the position of the remote entity relative to the player.
- Absolute consistency e.g. Locking and Serialisation [21], similar to techniques used in transactional databases, and TimeWarp [51], in which each client keeps a list of commands and states ordered by time and when a command arrives late or out of order, the command is inserted into the list with the correct ordering and the commands are then applied in order, starting from the state immediately preceding the new command, until the state reaches the current time.
- Predictive algorithms, such as dead-reckoning (discussed further in section 2.5), in which future positions of remote entities are predicted based on prior data, such as position and velocity.

Consistency resolution can be divided into the following two categories:

- Decision-making. Determining the outcome of decisions in the presence of inconsistency. The decisions can be determined by choosing a perspective, i.e. the server, in which the

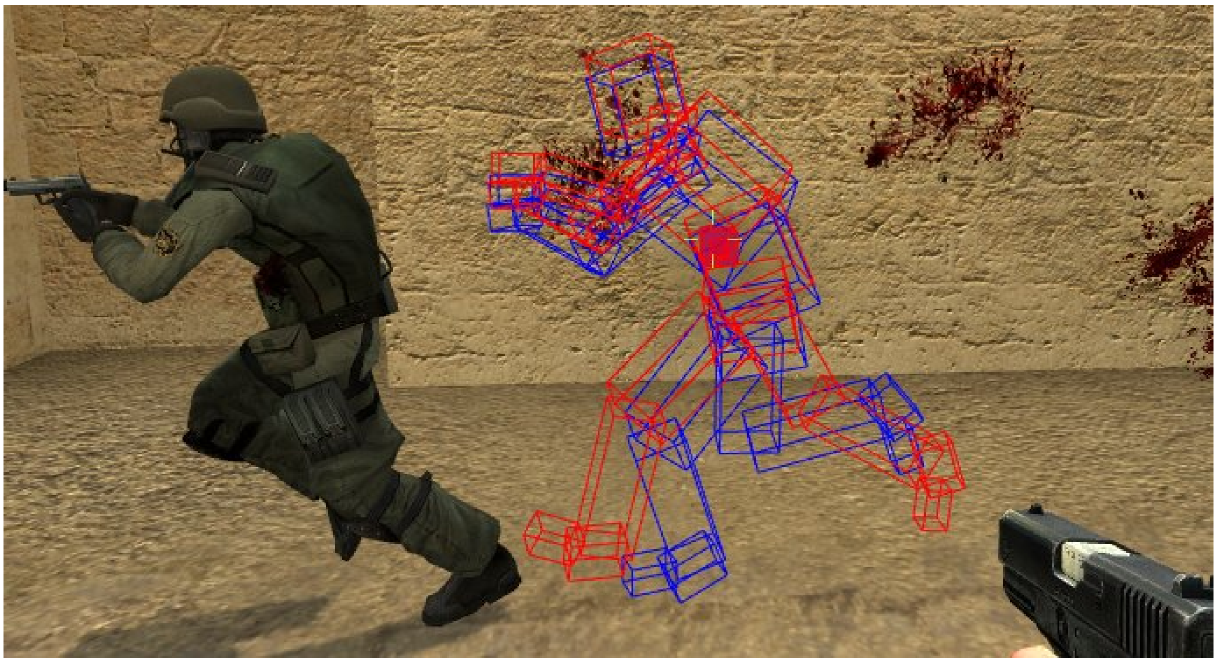


Fig. 2.4 A screenshot taken when 200ms of latency was present on a client. The red hitboxes show the target position on the client (100ms + interpolated period). While the user command was travelling to the server, the target continued to move to the left. When the user command arrived at the server, the server reverted the target position (blue hitboxes) using the estimated time of the command execution. The server calculates whether or not the hist was successful and sends a confirmation message to the client [88].

server maintains the whole state necessary to determine the decision; source client, the perspective of the client that initiates the action (e.g. shoots an enemy player); and target client, the perspective of the client that the action targets (e.g. target of an enemy player shooting) [76].

- Error Repair. Repairing inconsistencies as they are discovered. The three basic approaches to error repair are: Correct Immediately, in which remote entities are immediately updated to the latest received positions; Smoothly Correct, where remote entities' positions are interpolated over time to the latest position; and Tolerate, where if the inconsistency has no effect on gameplay, the inconsistency is not corrected [76].

2.3.2 Latency

Latency can significantly affect the quality of experience of a game and can impact both a player's performance in a game [4, 20] and subjective perception of the game [24]. However, this depends heavily on the game genre and actions taken by the player [19, 20]. Fig. 2.4 shows an example of the effects of latency on a networked game. Where the target appears to the player and hitboxes on the server used to detect a player's shot are significantly different.

2.3.3 Fault-tolerance

Fault-tolerance is a challenge for online gaming and can result from failures or unscheduled disconnections of server or peers which can lead to the loss of game state [91].

2.3.4 Fairness

Fairness is the degree of difference among all players' gaming environments, i.e. players should be treated equally and no game advantage/disadvantage should be given to players due to factors like latency [91].

2.3.5 Cheating

Cheating is when players maliciously gain an unfair advantage in a game. Cheating reduces the quality of experience for non-cheating players and is a main concern in the design of game architecture [91].

2.3.6 Scalability

Scalability in online games is typically measured in terms of players that can be supported [85]. An online gaming system's ability to scale depends upon the architecture employed, discussed below. However, scalability in terms of supported players should not be confused with the scalability of real-time physics, which is the focus of this study. Scalability of real-time physics can be measured in terms of objects that can be simulated while maintaining real-time speeds.

2.3.7 Architectures

Online game architectures fall into the following main categories: client-server; peer-to-peer; hybrid peer-to-peer; and distributed server (also referred to as multiserver). Each type has advantages and disadvantages and suitability depends on a number of factors, including number of players and whether or not the game is short-session-based or persistent.

Client-Server architecture is a type of architecture in which the game is executed and game state is managed entirely by the server. Clients connect to the server and receive the necessary information about the game world. All messages and interactions are sent through the game server and the server is responsible for disseminating messages to the appropriate clients. The main challenge with this approach is scalability, as a single server can only support a limited number of players [28]. This is normally solved with the addition of more servers, i.e. switching to a distributed server architecture. As all communication goes through the server, clients do not directly communicate with each other, meaning latency can be increased for interactions with other players. As there is only one central version of the game state, consistency resolution is simplified to solving inconsistencies between server and client and the server treated as the correct master copy. In addition, fault-tolerance is simplified to keeping a single backup of the sate of the central server, although this adds more complexity and cost, and

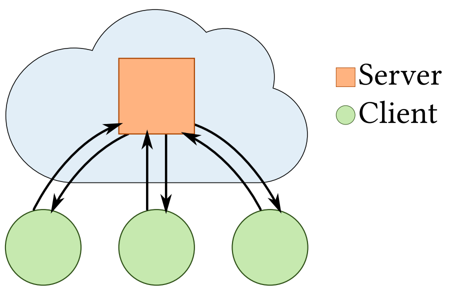


Fig. 2.5 Server hosted networked game.

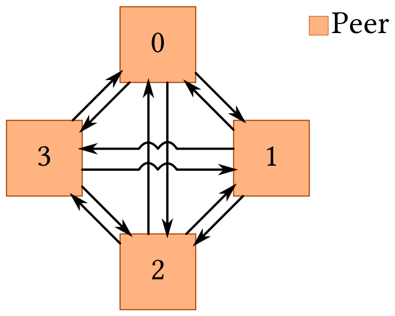


Fig. 2.6 Peer-to-peer hosted networked game.

can reduce scalability. Client-Server architectures are less prone to cheating than peer-to-peer based architectures, as the game provider has complete control over the game state [91].

Peer-to-Peer, often referred to as‘P2P’, is a type of architecture in which the game is executed and game state is managed entirely by the peers, although peers are often only responsible for a subset of the game state, such as the entities they are currently interacting with. A peer-to-peer architecture is shown in Fig. 2.6. Peers are all responsible for message forwarding to each other. In other words, peers act as both server and client. Peer-to-Peer reduces the cost for the game provider of maintaining expensive servers and can reduce latency between players, as communication is performed directly between peers, rather than communicating through a server. In addition, it provides scalability in terms of compute power, as the more peers that join, the more resources available to the system. However, as more peers join, the greater the communication overhead is between peers. In addition, peer-to-peer is prone to the problems of fault-tolerance, data consistency and cheating by players. A peer is more likely to fail or experience unscheduled disconnection than a dedicated server. As each peer is responsible for the game state, it is possible for communication errors to occur, resulting in two peers with different game states or a peer maliciously altering the game state or the messages being sent to other peers in order to cheat. The fault-tolerance requirements of short-session-based games are lower than persistent games, making peer-to-peer architectures better suited to short-session-based games [91]. Peer-to-Peer architectures have received much research attention in previous years, with many solutions that attempt to address the problems above, through the use of techniques discussed in [91].

Hybrid Peer-to-Peer architectures are a combination of client-server and peer-to-peer architectures. The peer-to-peer part of the system can be responsible for different aspects of the system, including Cooperative Message Dissemination, State Distribution, Basic Server Control and distributing software updates. [91]. Hybrid Peer-to-Peer enables trade-offs between consistency control and scalability, determined by how the system is divided between server and peer-to-peer. The more responsibility peers have, the greater the scalability but at a loss of consistency control.

Distributed server or Multi-Server architectures use multiple servers and the computational workload is divided between the servers. This can be achieved in one of two ways, either through ‘shards’ or through division of the game world.

In the ‘shards’ method, complete and separate instances of the game world exist, each maintained by one server. Each server is responsible for a different set of clients. The game can be scaled through the introduction of more servers, supporting more clients. However, as each instance of the game world is separate, there can be no interaction in the game world between players on different shards [91].

In the game world division method, there is a single instance of the game world and the workload is distributed either through each server being responsible for a region or a set of players. Division methods are discussed further in Section 2.6. Even with the workload distributed between servers, one server may be become overloaded if the workload is too high, for example, by a large number of players in a region being managed by a single server. Dynamic load-balancing schemes can be employed to improve distribution between servers and prevent servers being overloaded when resources are available on other servers. The disadvantage of distributed servers is the need for complex hand-offs when players migrate or interact between servers and can lead to consistency problems between servers as latency will always exist between servers [91].

The distributed server approach enables scalability as more servers can be utilised, allowing support for more players. However, depending on workload distribution, more problems are introduced. Fault-tolerance is potentially higher than a single server solution, even if no backup system is employed, as only players connected to the failing server will be affected [91].

2.3.8 Update Rate and Network Tick Rate

Two important aspects of online video game servers are the update rate and network tick rate, which will now both be defined. Update rate is the rate at which the server updates the game state, such as game logic updates and AI. Network tick rate is the rate at which the server communicates the game state to clients and clients communicate interactions with the server, such as movement input for a player character [73]. The two rates can be independent of each other and the network tick rate can be lower than the server update rate, reducing the network requirements through less frequent communication between server and client.

In many cases it is not possible for a server to communicate the entire game state to the each client due to bandwidth constraints. Instead, techniques are used to prioritise and filter the information that is most relevant to each client, in order to reduce the data that needs to be communicated with clients [73]. These techniques are known as Interest Management and discussed further in 2.6.5.

The rates chosen for network tick rates depend on a variety of factors, including the required level of accuracy and bandwidth restrictions. Higher tick rates enable higher accuracy, but at the cost of more bandwidth being required. Different genres of games have different accuracy demands. For example, in first-person-shooters (FPS), higher tick rates result in a higher mean shooting accuracy for players [45]. Whereas, genres like real-time-strategy (RTS) games have more relaxed accuracy requirements. In the RTS Age of Empires, latency only becomes noticeable if it exceeds $500ms$, meaning network tick rates can be very low without being noticeable to the player [73]. Tick rates of popular network games range from 128Hz for CS:GO to 20Hz for Minecraft [55].

2.3.9 Delta-Encoding

A common optimisation used by networked games is delta-encoding. Typically in games entities and their state change little from one update to the next, i.e. it is expected that only a small number of entities change between updates. Delta-encoding reduces network traffic by the server only encoding the difference (delta) between updates, reducing the size of messages that need to be sent [8]. Additionally, if an object's state has not changed, no message is required to be sent, dramatically reducing network overhead in cases of a large number of objects which are not changing. The main drawback of this technique is a missed update message sent to a client would lead to a different entity state from that point for that client and can lead to missing entities or entities that have been deleted on the server, remaining on the client. As result of this, delta-encoding either requires reliable message transmission between server and client or a periodically transmitted full "key" frame, containing the state of all entities [7].

2.3.10 Protocols

Games use both TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). Most games use the UDP network protocol for transmission of short and time-sensitive data and TCP for long and content-aware data, providing services such as matchmaking [60, 14]. UDP offers best effort communication. Unlike TCP, UDP is a connection-less protocol and does not guarantee data delivery, but suffers from less latency. In many situations in online games, lost packets are not critical, for example in an online game in which the server informs each client of the positions of each player, there are a large number of updates, meaning a missed update is not critical [73].

2.4 RakNet

RakNet is the gaming network library used in the development of the implementation of AP. Features of RakNet used by AP will now be described.

2.4.1 Reliability and Ordering layer

RakNet uses UDP but also provides a reliability and ordering layer. When messages have the reliable option enabled, RakNet guarantees that UDP packets will arrive at their destination

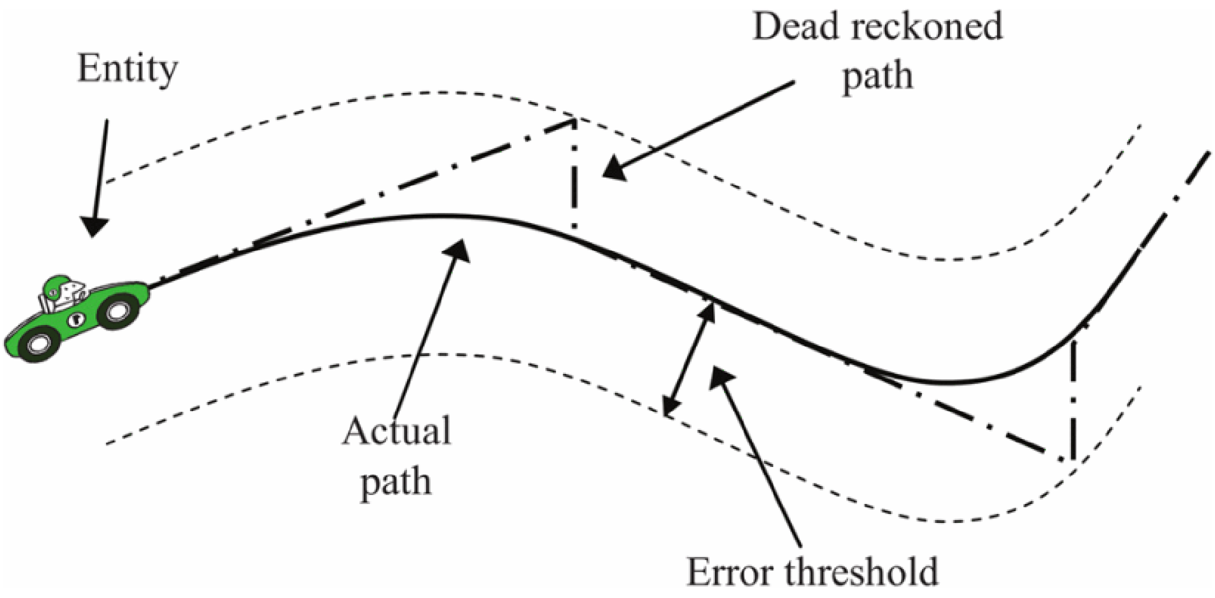


Fig. 2.7 An example of dead-reckoning [53].

eventually. When messages have the ordered option enabled, RakNet guarantees that the packets are ordered at the destination. Ordering has the advantage of removing the need to handle out of order packets [74].

2.4.2 Replica Manager

RakNet provides a replica manager plugin, which supports the use of replicas and delta-encoding for the updates of their states. Replicas are replicated entities, hosted by one node and replicated on connected nodes. For example, a server is responsible for the updates of a set of entities, the entities are then replicated on all connected clients. The replica manager handles the broadcast of: existing game entities to new connections, new game entities to existing connections; deleted game entities to existing entities and updates of states of replicated entities [75].

2.5 Consistency in Real-Time Physics

Maintaining consistency in the physics aspect of games when distributed is great challenge. This is due to the time-sensitive nature of physics (millisecond differences can result in completely divergent results, discussed further in 3.1.4) [53].

Consistency in physics can be divided into two categories:

- Consistency in simulation view, for example, differences between the physics simulation on the server and a client.
- Consistency within the simulation itself (when the simulation is distributed between devices).

The former is a very common issue in online gaming, particularly due to high latency and limited bandwidth between servers and client, necessitating the need for client-side correction techniques. The most common technique used is dead-reckoning, shown in Fig. 2.7. The use of dead-reckoning in games is a well researched topic, including the suitability for different genres [71], the accuracy of dead-reckoning [1], its effects on fairness [2] and enhancements of the technique, such as [53] which takes into account environmental (static) objects when dead-reckoning is performed.

Although these correction techniques may be suitable for handling inconsistencies in simulation view between servers and clients, they are not suitable for maintaining consistency within a distributed simulation, which is the focus of this research. This is because in order to prevent diverging results, strong consistency of collisions must be maintained (discussed further in 3.1.4).

There are two approaches to handling the authority of physics in networked games:

1. Server-authoritative physics - the server is responsible for simulating the physics aspect of the game.
2. Client-authoritative physics - the physics is simulated on the clients in a distributed manner.

Server-authoritative physics on a single server does not suffer from inconsistencies within the simulation, meaning there is no potential for the physics state to diverge as there is only one single simulation. Inconsistencies between the server and client will exist due to latency, but the state on the client can be corrected. However, server-authoritative physics uses up limited computational resources in single server systems.

Server-authoritative physics in distributed server systems creates complex problems at the boundaries between servers. These boundary problems include:

- The handing off (migration) of physical entities between servers, e.g. when an object traverses between server regions.
- Maintaining consistent states of physical entities between servers, i.e. servers will need to be aware of the state of objects being simulated on a different server and this state needs to be kept consistent between the two servers.
- The interaction of entities between servers, i.e. objects being simulated on different servers that should physically interact with each other.

Client-authoritative physics creates consistency problems within the simulation, as the physics states on different clients will quickly diverge (discussed further in 3.1.4). This is especially true with high latencies and limited bandwidth experienced between clients. Client-side resources, both computational and network must be sacrificed in order to attempt to reach consensus on the physics state, however, inconsistencies will always be present. Gameplay aspects of games, such as life/death and score decisions require a high-level of consistency in order to maintain fairness [**?**]. Due to these consistency problems, physics is unable to influence gameplay without sacrifices to fairness being made.

This study aims to address the problems of consistency in distributing real-time physics between servers in multi-server networked games. Solving this problem would allow for physics to play a larger role in networked games, including having significant influences on gameplay.

2.6 Distributed Virtual Environments

Although there has been prior research utilising multiple servers to distribute the task of solving physics-based problems (e.g., [50]), there is no known literature describing real-time interactive physics exploiting the addition of servers to gain scalability. The closest work to this challenge is that carried out to seek scalability in terms of player numbers in online gaming in the field of Distributed Virtual Environments (DVEs), of which online gaming can be considered a subcategory.

Some key aspects to consider are the goals and requirements of real-time DVEs, which are different from other distributed simulations. DVEs only need to be "sufficiently realistic", which differs depending on context. This is known as selective fidelity, where different levels of realism are applied selectively to different aspects of the simulation, depending upon how much the aspect affects the outcome of the simulation. For example, in a ship simulator, realistic-looking clouds will not contribute to the simulator's effectiveness as a training device. DVEs must execute in real-time, unlike non-real-time simulations. Repeatability and synchronisation requirements are often relaxed as they may not contribute to the overall goal of the DVE, whereas they are essential for non-real-time simulations that require a high degree of accuracy. As a result of these different goals and requirements, many of the algorithms and techniques used in non-real-time simulations are not suitable for use in real-time DVEs [30].

DVEs also share the same architecture categories as online gaming [30] i.e. centralised server (referred to as client-server in 2.3), distributed server and distributed serverless (referred to as peer-to-peer in 2.3). Online gaming also includes the hybrid peer-to-peer architecture.

A brief overview of some approaches to implementing systems for DVEs will now be given.

2.6.1 Distributed Interactive Simulation

One example of an implementation of a DVE is Distributed Interactive Simulation (DIS), developed in the early '90s [30]. DIS followed the following design principles:

- Autonomous simulation nodes. Each node is responsible for simulating one or more entities and generates messages, referred to as Protocol Data Units (PDUs), to notify other nodes of updates to states of entities.
- Transmission of "ground truth" information. Each node sends absolute truth concerning the state of the entities it represents.
- Transmission of state change information only. To economise on communications, simulation nodes only transmit changes in behaviour. (Referred to as delta-encoding and discussed in 2.3.9).
- The use of "dead-reckoning", as discussed in 2.5.

The principle of autonomous simulation nodes has some significant benefits, particularly in the case of heterogeneous simulations (distributed simulations using more than one type of simulation). It simplifies the development of simulators (simulation instances) as each simulator is not concerned with the details of the other simulators and allows for easy integration of legacy simulators. It also allows for simple joining or leaving of the distributed simulation by simulators. Autonomous simulation nodes advance the simulation time independently of each other, meaning there is no need for clock synchronisation between nodes. Additionally, autonomous nodes are not responsible for determining recipients of messages, which is instead performed by the underlying infrastructure [30].

2.6.2 High Level Architecture

HLA is a successor to DIS and was originally developed in the mid-'90s for military simulations. In HLA a distributed simulation comprises of individual simulators called federates, which combined are referred to as a federation. The main goal behind HLA is to support interoperability and reuse of simulations (i.e. for use with heterogeneous simulations) [30].

All entities in the simulation are represented in the HLA and each entity instance contains:

1. A unique identifier
2. Attributes that indicate those state variables and parameters of an object that are accessible to other objects
3. Associations between objects.

HLAs include a non-runtime and runtime component. The non-runtime component is responsible for specifying the definitions of the object types to be used by the federation. The runtime component is responsible for providing the federates with a means to interact with one another, known as the Run-time Infrastructure (RTI). Each federate is responsible for a set of entities and the state variables for entities are stored within the federates and not the RTI. Each federate is responsible for informing the RTI of any changes to state. Federates register interest with the RTI to receive updates for entities for which they are concerned [30].

It should be noted that different attributes for a single entity can be owned by different federates and the RTI has no knowledge of the semantics of the information that it is transmitting. As a result of this lack of semantic knowledge, no optimisations (e.g. dead-reckoning) can be performed by the RTI and any techniques must be implemented only on the federates [30].

HLA is ideal for heterogeneous distributed simulations and for re-use with different types of simulations. However, there are some disadvantages to this architecture:

1. It increases latency between simulations as communication must go through the RTI.
2. It does not allow for any optimisations to be performed in the RTI as the RTI is not aware of the semantics of the simulations.

2.6.3 SpatialOS

SpatialOS uses a very similar architecture to HLA which also uses an RTI and 'workers' which are equivalent to federates [41]. A key difference between HLA and SpatialOS is that in SpatialOS the full entity data of all entities is stored within the RTI (as opposed to just attribute metadata and associations in HLA) and each worker stores a copy of a subset of the entities, which it is responsible for [40]. Each worker communicates updates for the entities it is responsible for, which then get disseminated by the RTI to other workers with registered interest in those entities.

In the case of real-time physics, we can assume the use of either homogeneous simulations or at least simulations that will be simulating physics with a large degree of shared semantics between simulations (as all simulations are dealing with physical simulation). As a result of this, there is little to no advantage gained in using HLA or similar architectures, furthermore direct communication between servers has the benefit of lower latency between servers.

2.6.4 Migratory vs Non-Migratory

There are primarily two ways in which server-side resources can provide scalability in online gaming (e.g., DVEs): (1) Migratory; (2) Non-migratory. In migratory approaches, a server will assume responsibility for handling in-simulation objects within a region. When objects traverse region boundaries into a region that is the responsibility of another server, they will be handed over to the other server. In a non-migratory approach, in-simulation objects are allocated to the responsibility of a particular server at instantiation time and stay with that server until they are deleted.

The benefit of a migratory approach is that tightly coupled objects (interacting frequently) can be co-located on the same server, reducing interaction latencies. However, the act of moving such objects may be costly in terms of time required to resolve the hosting requirements of an object. The benefit of a non-migratory approach is that servers are rarely exhausted but network traffic will result in higher latencies that will inhibit the fidelity of interaction between objects.

Migratory and non-migratory approaches are now described in greater detail.

In the migratory approach, a single game world exists, but is divided into geographical regions. Each region is maintained by a separate server (e.g. [25, 38, 23, 58, 41]). The main drawback of this approach is the complexity of handling interactions between objects in different regions/servers while maintaining consistency [91]. A technique to minimise these issues is to use overlapping regions between spatial partitions. Servers share state information about objects in the overlapping region (examples include 'zoning' as described in [25] or 'sub-regions' as described in [38]). Examples of games using this technology include [64] and [11], which use the SpatialOS platform [41].

In the non-migratory approach, the game world is not divided into geographical regions and players are split between servers in one of two ways: (1) Several instances of the game world run with complete independence from one another (known as shards e.g.[9]) and players have no interaction across shards [91]; (2) Players are distributed amongst servers by some other non-geographical method and interactions with players on other servers requiring servers to share messages [47].

Although shards allow a degree of scalability in the number of players, it is not suitable for use in scaling real-time physics simulations as all entities within a real-time physics simulation, in the same geographical region, may interact with each other.

In the case of architectures not using shards, Interest Management is required to prevent message passing growing polynomially as players increase (e.g. [6] and [47]).

Despite DVE being a popular area of research, the literature is restricted to modelling player interaction across servers and balancing their support on different servers. Clearly, the interaction patterns of players are significantly less demanding in terms of timeliness than that of interacting physical objects.

2.6.5 Interest Management

The technique presented in this paper, Aura Projection, makes use of the aura concept from Interest Management. Interest Management is a term used to describe any method of restricting message dissemination between objects within a virtual space, with the aim of reducing the network overhead when large numbers of participants or players are connected to a virtual world [62].

Interest Management can be broadly classified into two categories: Regions and Auras [62, 81].

“In the region based approach the virtual world is commonly, but not always, divided into well defined uniform sized regions that are static in nature (i.e. their boundaries are defined at virtual world creation time)."[81]

“In the aura based approach each object is associated to an aura that defines an area of the virtual world over which an object may exert influence. Ideally, an object may potentially communicate their actions to only objects that fall within their influence."[81]

Interest management is a well researched topic and enhancements to it exist, such as [6] which proposes the A3 algorithm. A3 uses a combination of a circular area of interest and field of view combined with a relevance gradient. [47] proposes a Behavioural Interest Management Technique that allocates resources based on player interactions. Auras (an area of interest/influence) are used to determine player message exchanges, reducing message passing while promoting player number scalability.

In addition to reducing the network traffic between servers, interest management can also be utilised to reduce the required network traffic between client and servers. In terms of physics being simulated on server(s), clients will be most concerned with objects within close proximity to them and less concerned with objects beyond the range in which can be physically interacted with. In addition, clients will be more concerned with objects in their field of view than objects that can’t currently be seen. An example of an interest management technique using distance and field of view is the A3 algorithm [6]. Through the use of client interest management it would be possible to simulate a persistent world far larger than could be simulated or rendered on a single machine, yet allow a player unrestricted, seamless access to the entire world.

Interest Management is relevant to this study as it enables the minimisation of network overhead in distributed virtual environments. It is important to note that objects within a virtual space are only affected by other objects in their region and neighbouring regions or objects within their area of influence. AP uses the concept of auras to minimise message passing between servers while still allowing time-space consistency to be maintained.

2.7 Distributed Real-Time Physics

In this section research into existing methods for distributing real-time physics across a cluster of nodes in a network will be discussed.

Distributed Real-Time Physics has attracted a lot of commercial attention recently, for many applications including games, multi-agent AI, city planning and VR [41]. The commercial interest in Distributed Real-Time Physics is likely due to the recent availability of cloud computing, which allows for scaling of the required computing resources on-demand. On-demand scaling provides a more cost-efficient approach as opposed to the traditional use of private server clusters [63].

Using Distributed Real-Time Physics removes the limitation of the computational power of a single machine (such as the limitation in server-centric architectures [36] and avoids the complexity and communication overhead of P2P architectures such as that described in [39]) thus allowing for greater scalability in both the number of users [38] and complexity and size of the Virtual Environment.

Distributed Real-Time Physics techniques fall into two main categories, those that use multiple instances of the same physics engine (as used by AP) and those that distribute particular aspects of the computational workload of a physics engine. Both techniques will now be discussed.

Examples of previous attempts at scaling real-time physics engine through the use of distributing a particular aspect of the workload include [62] and [3].

[62] describes a way of scaling physics simulations through deploying a real-time collision detection service across a cluster of servers. The simulation is spatially divided into regions and each node is responsible for all narrow phase collision detections within one region. A dedicated node is responsible for determining which region an object is contained within and informing the region's node that the object should be considered for collision detection. Objects intersecting region boundaries are discussed in [62]; objects intersecting a region boundary result in the same collision pair appearing in more than one region, these are dealt with by ensuring only one node will enact the narrow phase collision test for the pair. The process of identifying duplicate pairs is not described. The result of the response on a single object from collisions on two nodes is also not described.

In [3], a modular approach to physics simulations is described. A dedicated module is used for each type of object (rigid body, spring-mass, fluid etc.). Interactions are then handled between object modules through interaction modules. Each module can be run on a node in a cluster, allowing the simulation to be scaled.

Techniques that distribute a particular aspect of the workload have not gained any commercial attention recently, unlike techniques that use multiple instances of the same physics engine, which will now be discussed.

Examples of Distributed Real-Time Physics that use multiple instances of a physics engine include SpatialOS [41] and Aether Engine [37]. SpatialOS spatially partitions the world into regions, each region is a separate instance of a physics engine. Where regions meet, there is an overlapping area, allowing objects hosted on different servers to interact. However, the specifics of the techniques used by SpatialOS and Aether Engine are not described in any literature. The demonstration video of SpatialOS exhibits unnatural object "jitter", which is possibly a result of network latency and collision inconsistencies.

This thesis proposes an alternative technique to SpatialOS and Aether Engine, AP, that does not use overlapping regions and reduces the effect of communication delays between servers regarding states of physical entities. If the instability problems e.g. "jitter" can be solved, this would enable the use of Distributed Real-Time Physics for games and simulations that rely on stable and consistent physics.

2.8 Cloud Computing

In this section a brief overview of cloud computing is given, deployment and service models are discussed. This study’s implementation is deployed on the chosen cloud provider, Amazon Web Services (AWS). The latency and packet-loss conditions that affect AP are presented and these values will used to inform the correctness experiments performed on AP.

Cloud computing is a large scale distributed computing paradigm [29]. Cloud providers offer flexible, real-time on-demand services, including servers, storage and applications. Cloud Computing allows compute resources to be rapidly and elasticity provisioned [59].

Cloud deployment models are categorised into the following:

- Private cloud - Set up within an organisation’s internal data centre. This is much more secure than the public cloud due to its internal only exposure and access is limited to designated users within an organisation [59].

- Public cloud - Resources are provided by a third-party provider. Users request resources in a self-service manner over the Internet. Users are typically charged on a pay-per-use model. Public cloud services are less secure than private cloud as users have more responsibility in ensuring the security of their applications and data from potential malicious attacks from other users in the cloud. Public cloud services offer far greater resources than a private cloud, allowing spikes in demand to be provided for [59].

- Hybrid cloud - A private cloud linked to one or more public clouds. This provides solutions with more secure control over data and applications while still allowing public access over the internet [59].

Cloud Computing Service Models are categorised into the following:

- Infrastructure–as–a—Service (IaaS) - Provides basic hosting of resources, e.g. Virtual Machines (VM)s and storage [59].
- Platform–as–a–Service (PaaS) - Provide the ability to build or deploy applications on top of IaaS [59].
- Software–as–a–Service (SaaS) - Provides entire sets of applications running in the cloud. Users have no responsibility or management oversight for SaaS. Examples include Gmail from Google [59].

2.8.1 Latency Values

In order to validate the correctness of AP, experiments were conducted on AP in the cloud under differing conditions, including when subject to different levels of latency. Real world values of latency were used with values taken from a recent study that performed a series of exhaustive performance tests across all major cloud service providers, including our chosen provider, (AWS) [84]. Benchmark categories included:

- Global end-user network latency - measured between a variety of global “end-user” machines across the Internet and different geographical cloud availability zones (AZs)
- Inter-region latency - measured between cloud hosts located in different geographical regions within the same cloud provider
- Inter-AZ latency - measured between cloud hosts located within the same geographical region and within the same cloud provider.

This study is concerned with inter-region and inter-AZ latencies as these are the latencies that affect server-server communication and the correctness of the system. Lower latency is ideal for the deployment of a real-time distributed physics system, in the case of AP, lower latency means smaller auras leading to better partitioning and subsequent performance. However, inter-AZ deployment may not always be possible or desired (e.g. lack of availability of resources in one AZ or to reduce latency between users and some of the servers in the system in the case of geographically diverse users). Therefore the latency experiments used inter-AZ latency and low inter-region latency values. The lowest latency observed was from the inter-AZ category, i.e. servers hosted in the same geographical region at $< 2ms$. Inter-region latency measured in the range of $(7 - 302ms)$.

The chosen values for the latency experiment were as follows: $2ms$, $8ms$ and $16ms$. These values represent inter-AZ latency and low inter-region latencies within AWS, for regions that lie within same continent (eu-West-2 - eu-west-3) and (eu-West-1 - eu-west-2) [84].

2.8.2 Packet-Loss Values

Another factor that affects the performance of networked applications is packet-loss. An experiment was conducted in this study to determine the effects of packet-loss on the correctness of AP. The most common causes of packet-loss include network congestion [72] and bit errors in wireless transmission [43]. According to [83], carried out in 2018, cloud providers' networks have a very high level of reliability and exhibit only negligible packet-loss (measured at 0.01% of packets lost, on average). This holds true even when traversing inter-region backbone links and between different cloud providers. The only significant measured level of packet-loss was between some geographical "end-user" locations outside of the cloud providers' networks, and were typically $< 1\%$. However, this study is only concerned with packet-loss between servers in the cloud and not "end-users". The only notable exception to this and worst case for packet-loss was for traffic entering or exiting China, which experienced packet-loss as high as 8%.

The chosen values for the packet-loss correctness experiment range from $0\% - 20\%$. These values cover the entire range of expected packet-loss values while running in the cloud, as well as beyond those values, in order to determine what impact severe packet-loss would have on the correctness of AP and establish any trend increasing packet-loss has.

2.9 Real-time Physics for Scalable Simulations

In the past, real-time physics engines have not always been suitable for use in simulations [10]. However, in recent years significant effort has been put towards realism of real-time physics engines and the mathematical error has reduced. Commercial game engines (using real-time physics) are finding their use in other industries, including their application in virtual simulations (e.g. [90, 48, 77], collaborative VR [69] and in film production [54].

However, real-time physics for use in other applications has been bounded by the computational power of a single machine. For large, detailed and/or dynamic simulations, this may be insufficient [44]. Prior to the proprietary commercial software SpatialOS [41] and Aether Engine [37], real-time physics simulations have been limited to what can be achieved on a single machine. Scalability in simulations can be achieved by distributing a real-time physics engine across multiple machines, alleviating the computation limit of just one machine. This would allow many more applications to take advantage of real-time physics. This study aims to bring the methods for distributing real-time physics into an academic setting for the first time.

2.10 Microservices for Game Engines

Microservices have gained much recent attention in the software industry in the context of cloud computing. The reason for this, is the microservice architecture allows for greater leverage of the opportunities that cloud computing presents. Additional advantages include flexible horizontal scaling, as well as more efficient team structures during development [52].

Popular existing commercial game engines, such as Unity [86] and Unreal Engine [27], are based on the Monolith architecture. However, for game engines to take greater advantage of cloud deployment a move to a microservice architecture will be needed. This would allow for independent horizontal scaling and resource provision management of the different systems within a game engine, such as graphics, physics and AI. Despite the recent commercial interest in microservice architecture, there has been no academic research into the use of microservices for game engines. The only notable exception to this is [87], however, this has a limited focus to the application of microservices to the genre of Massively Multiplayer Online Role-Playing Games (MMORPG).

A microservice architecture for a game engine would create a more cost-efficient approach to cloud-hosted games and simulations, while also creating the potential for the scalability of individual systems on an on-demand basis. This study concentrates on separating out the physics system from the game engine and distributing it across multiple servers while maintaining the real-time streamed service to players. Through this distribution, this study aims to demonstrate that the physics aspect of a game engine can be horizontally scaled through the use of cloud computing.

2.11 Streamed Gaming

Streamed Gaming (also know as Gaming as a Service or Cloud Gaming) consists of cloud servers streaming to a player's device with player input being returned to the cloud server. The player's device acts as a thin client. The main benefit is that a player does not require expensive, powerful hardware, and games can be played on any operating system (e.g. Android, Linux and Mac) [26]. However, these benefits come at the cost of bandwidth and latency requirements [89].

The first successful demonstration of game streaming was in 2001 by G-cluster, who publicly demonstrated game streaming over WiFi to a PDA in 2001 and launched a game-on-demand service in 2004 [13]. In the late 2000s, more companies introduced game streaming platforms such as OnLive [70], Gaikai, and GameNow [31]. Due to financial difficulty in 2012, OnLive sold their patents to Sony and Gaikai and in 2012, Gaikai was bought by PSNOW [80]. Streamed Gaming services currently available include NVidia Geforce Now [67], PSNOW [80], Google Stadia[34] and Microsoft xCloud[56]. NVIDIA GRID hardware technology is targeted specifically at Streamed Gaming [65].

A drawback to streamed gaming is the requirement for a significantly more powerful machine at the server-side than what would be required if the game was played solely at the client side. This is because the server not only has to run the game, but has to process the video and audio stream into a suitable format for streaming. In addition, real-time player interaction requires low latency and high bandwidth resulting in networking infrastructure more expensive than would be expected for regular streaming services.

In recent years, streamed gaming has become an active area of research, with many studies focusing on the effects of latency on the quality of service, e.g. [79, 17, 42, 46, 18, 15]. The challenges in streamed gaming that remain are both technical as well as economical [13].

In streamed gaming, each game instance resides on a single server. There is no technology to balance the real-time requirements of the game across multiple servers. The core problem is that all gaming technology is built and designed for single console/PC install and the greatest bottleneck is the inability to share physics calculations across machines.

The technique presented in this study, AP, is intended for use in streamed gaming. This study aims to be preliminary research into the division of game engines into microservices, allowing streamed gaming to take better advantage of cloud infrastructure, such as through elastic scaling of resources as needed. AP allows multiple servers to simulate physics, meaning physics simulations are no longer limited in size and complexity to that which can be achieved by only a single server. This allows for game streaming servers to make use of multiple physics servers, the numbers of which can grow and shrink as needed by the game and greatly increase the limit on size and complexity of physics within the game.

2.12 Background Summary

The reader has been provided with a brief overview of the working of real-time physics and introduced to important terms and concepts. The current state of online gaming has been discussed, including the main challenges faced. In addition, aspects of online gaming that are relevant to this study have been presented in greater detail. The challenges in physics in online gaming are those of latency and consistency i.e. maintaining consistent states while latency exists between servers. Techniques for handling consistency on longer time scales (100s of milliseconds), such as interactions between players, already exist and compensation techniques (based on dead-reckoning) exist to reduce the perceived differences in physics state due high latency between clients and servers. However, neither of these are appropriate for distributing the simulation across multiple machines. Therefore this study intends to address this challenge through the use of a novel technique called AP.

The applications of AP in both scalable games and simulations, as well as in cloud gaming, have been discussed. AP is a move towards real-time physics as a microservice for a microservice-based game engine, which can better take advantage of cloud infrastructure, ideal for cloud gaming. AP also provides a means of scaling real-time physics simulations beyond what can be achieved on a single machine. For the remainder of this research, we describe our approach to solving this algorithmically, present how a working implementation was achieved, and present results evidencing our work. This is the first presentation of literature that can demonstrate real-time scalable server-side physics modelling and is a significant contribution to reducing the cost of commercialised streamed gaming.

Chapter 3

Problem Definition and Proposed Solution

In this chapter the requirements of a solution to scalable real-time physics and the challenges of distributing a real-time physics engine will be discussed. A naive approach will be used as an example. Some potential solutions to distributing real-time physics and their trade-offs will be considered. Finally, this study's proposed solution, AP, (originally presented in [12]) will be described, along with high-level overviews of the challenges with this approach and the implemented solutions for the solved problems and proposed solutions for the remaining problems.

3.1 Solution Requirements

In order for any solution to scalable real-time physics to maintain scalability, the extra processing overhead must be less than that gained by distributing the simulation workload. In addition, when scaling, a solution must maintain the correctness and fidelity of the simulation, i.e. errors should not be introduced.

In general terms, distributing work between servers has the following overheads: the logic of determining messages to be sent; the processing overheard of sending the messages themselves; and the logic of handling received messages from other servers. To achieve the greatest scalability, a solution must minimise these overheads while maximising useful work done by each server.

The types of errors that can occur in distributed real-time physics simulations will now be discussed. Types of errors include:

- Time-space inconsistencies, in which two different objects occupy the same space at the same time, but on different servers.
- Late collisions and missed collisions, in which after an object migration prior time-space inconsistencies lead to collisions either being detected late or missed entirely.

- Object state conflict, in which the same object has different states on different servers. This can lead to objects appearing to 'jitter' and can be the result of diverging results of collisions between the two servers.
- Objects not being simulated (missed updates), which could occur when objects are migrating between servers.
- Objects being duplicated and objects being lost.

Examples and consequences of the above errors will now be described.

3.1.1 Time-space inconsistencies

Objects being simulated on different servers may end up occupying the same space and the same time, which is a physical inconsistency. This error could occur due to latency issues between servers or if a naive approach is taken to handling objects near server-region boundaries. For example, in a naive approach, object ownership may be determined by the centre of the object and no special consideration is given to objects near server-region boundaries. If two objects on different servers are overlapping the boundary at the same place, but their centres remain in their owning server's region, the two objects will have no interaction, despite occupying the same space at the same time. In a centralised version of the simulation, the two objects would collide and interact as soon as the two began to overlap with each other.

3.1.2 Late collisions

Due to latency between servers, object positions on different servers will never be agreed upon exactly by both servers. The delay between servers means positions of objects on another server is always out of date. Even in solution where a single object is simulated on both servers, due to the non-determinism of real-time physics engines, the states will diverge.

As a result of the out of date positions of objects, two objects hosted on different servers that would collide with each other, are unaware of the latest position of the other object. This means collisions between objects that are hosted on different servers will always occur late and at different positions on different servers, leading to differing results on different servers.

If objects that are going to collide are migrated between servers, they need to be migrated and arrive before a collision occurs. If the arrival takes place late, this may result in the two objects overlapping significantly (occupying the same space at the same time). The consequence of these late collisions are unstable collision response. To understand how late collisions lead to unstable collision response, how real-time collision detection works must first be understood. In real-time physics engines, objects move in discrete steps, as a result of this, when two objects collide, they overlap each other i.e. penetrate. The collision is resolved by calculating the forces resulting from the collision and moving the objects so they no longer overlap (in PhysX if penetration is high, the latter may be done over several physics time-steps). If the penetration is high, such as in the case of a late collision response or if two objects are moving at high speed towards one another, the collision response can no longer guarantee stable results.

3.1.3 Missed collisions

In cases of higher latency, or small or fast moving objects, collisions may be missed entirely. A typical example of this is the 'bullet-through paper problem', where the discrete updating of objects means that a fast moving object, the bullet, never overlaps a thin object, the paper, and a collision is never detected. The distance the bullet is updated each step is wider than the collision volume of the paper. High latency can mean the position of a remote object is received so late that it misses the overlap with a local object that would have collided entirely. The larger or slower moving the objects involved are, the less likely this situation is to occur. Alternatively, if objects are migrated between servers, the object must arrive before the collisions happens, as discussed in 3.1.2, if a delay in migration is severe enough, the collisions will be missed entirely.

3.1.4 Diverging results

PhysX is non-deterministic, as a result of this, repeating the same scenario with the exact same starting parameters does not necessarily lead to the exact same end state. The order in which collisions are resolved are randomised to achieve greater stability of the simulation. When there is a queued execution of collision calculations in the collision response phase, the objects considered in the latter stages of the phase are prone to cumulative errors from the previous calculations within the same phase. By randomising the order of the collision calculations each time step, the cumulative error is not propagated into the same objects over multiple time steps.

Due to this non-determinism, if the collision response is performed on different servers with the same group of objects, they will not lead to the same results. PhysX has an option for increased determinism, but as servers will have different states, as they will be hosting objects that the other server is not, this will not guarantee the exact same results. The results of a single differing collision response can create a significantly different state of objects, for example if a stack of cubes is dropped onto a flat, level plane, the cubes land in such a way where the stack is on the verge of collapsing. In one iteration, the stack of cubes may remain standing; in a second iteration, the stack of cubes may collapse.

As a consequence of non-determinism and diverging results, it is not possible to simply simulate the same objects on two different servers without some means of the two servers reaching consensus on the object state. In addition, it is not possible to measure correctness of a distributed simulation by comparing it with a single centralised simulation, i.e. run both solutions with the exact same starting parameters and compare the final state of the distributed simulation to the centralised simulation. A distributed solution can produce a significantly different final state compared to a centralised solution, despite handling all collisions correctly.

3.1.5 Object state conflict

The challenges involved with two servers reaching a consensus on the state of an object if the two objects have conflicting states will now be discussed. If a solution is used in which two servers simulate the same object (such as in an overlapping region between servers), diverging results from collisions can result in the same object can end up in different states on different servers. These conflicts in state may be small, in which case they can be resolved by either choosing the state from one server or taking the average of the two objects, however, this may appear as 'jitter' if the differences between states is large enough. 'Jitter' refers to objects appearing to instantaneously snap back and forth between positions. Furthermore, cases may exist when the two states have diverged significantly, such as the stack of cubes example in 3.1.4, in which case averaging of the two states cannot be used. One server's state would have to be chosen, which would appear on the other server as though the objects have 'teleported' (instantaneously changed position). In addition to this, prior to the server having its object's states overwritten, those objects may have affected other objects in only that server. For example, continuing with the stack of cubes example in 3.1.4, suppose the stack of cubes fell over on one server and one of the cubes collided with a sphere, that is only being simulated by that sever (for example, if it is far from the boundary). The sphere is given a velocity as a result of the collision. If the cubes in the stack are then updated to be the same as on a remote server, essentially rewriting the history of the cubes, the end result would be a stack of cubes that is upright and sphere with a velocity from a collision that has been effectively erased from the simulation history.

3.1.6 Missed updates

Objects not being simulated (missed updates), can occur when objects are migrating between servers. When objects are being migrated between servers, they don't exist on any server, as a result the position and rotation of the object will not be updated. This would appear as though the object has frozen, how noticeable this would be to an observer would depend on the situation and the number of updates missed. For example, if the object is stationary or moving slowly, several time steps could be missed without being noticeable. Alternatively, in a scenario where a large object was in a fast free-fall, very few update steps could be missed before an observer would notice that the object is not being simulated correctly.

3.1.7 Object duplication or loss

Any solution in which objects are migrated between servers is vulnerable to object duplication or loss. Networks do not provide absolute guarantee of message deliveries or order of message arrival. As a consequence of this, network messages for the migration of objects may be lost, in which case the object could be lost. For example, if one server sends a migrate message for an object and deletes the object from its simulation and the message is lost and the object never gets received and created on the receiving server. Alternatively, a server may not delete the migrating object until an acknowledgement is received from the server it is migrating the object to. In this case, the acknowledgement may be lost and the object kept on the sending server, despite a duplicate of the object now existing on the receiving server.

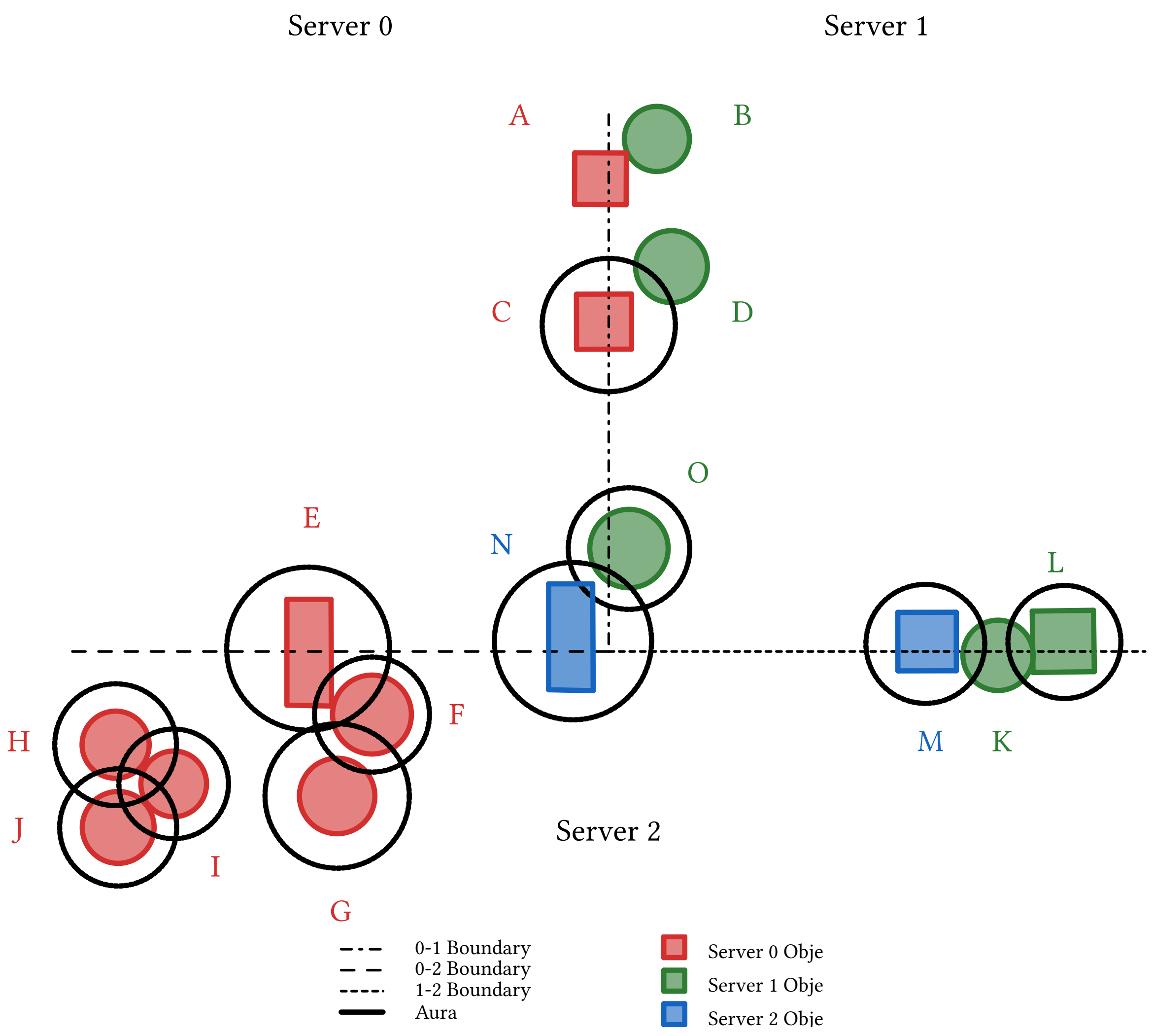


Fig. 3.1 Aura Projection Scenarios

3.2 Naive Approach

To provide a comparison and to demonstrate this study's contribution to the field, a naive approach will now be described. The naive approach is used to highlight the challenges faced when solving the problem of distributed real-time physics.

Assuming a migratory approach in which servers are responsible for a geographic area, when one object crosses a boundary between geographic areas, such an object's hosting must be transferred to the appropriate server. It is handling this transfer that is the underlying problem.

Accomplishing a transfer with a simple message from one server to another is charged with many problems:

- The message may be lost and the object disappears.
- The message may be delayed and the object disappears and reappears.
- There may be more than one candidate server for hosting (if object travelling quickly) resulting in duplication of an object.
- The server an object leaves may be unaware of when it should stop simulating the object (resulting in duplicated objects [22]).
- An object may be colliding with another object across the boundary and the collision would be missed if both objects are hosted on different servers (objects may pass through each other).

For additional clarity consider objects A and B in Fig. 3.1. Each server is aware of the objects they host but not other objects. As such, objects A and B would not interact and not collide with each other, creating a time-space inconsistency. A time-space inconsistency is defined as two objects occupying the same space at the same time. If Object B has a velocity directed towards the boundary, it would continue at its initial velocity until it traversed the boundary, at which point it would migrate to server 0. Upon completion of the migration, objects A and B would overlap far more than would be expected in a typical simulation, resulting in a late and inappropriate collision response.

3.3 Considered Solutions

There are several solutions that should be considered to distribute a real-time physics simulation across multiple machines over a network:

3.3.1 Both servers simulate objects on boundary

Simulate objects overlapping the boundary on both servers, however, results would diverge significantly as objects interact with objects not overlapping the boundary and there is the extra computational overhead of needing to be simulate objects twice.

3.3.2 Both servers simulate objects in an overlapping region

Simulate objects overlapping the boundary and objects interacting with those on both servers, e.g. by using an overlapping region. Object may interact with objects not in the overlapping region, so the results would diverge or the overlapping region would have to dynamically change in order to keep all interacting objects within the overlapping region. Even if all objects interacting were in the overlapping region, results would still diverge due to non-determinism of real-time physics engines. The divergence of results due to non-determinism could be corrected by periodically synchronising the objects (ensuring the objects have the same state on both servers), however, this would appear as jitter as objects would be moved from one position to their corrected position instantaneously. In addition, this would require additional network overhead for the syncing of objects between servers and extra computational overhead of needing to simulate objects twice.

There would be a trade-off with this approach, either:

1. Frequent messages are sent with small corrections for objects, appearing as jitter. This would require lots of computational and network overhead.

2. Infrequent messages with large corrections for objects, appearing as objects moving large distances instantaneously but would require less computational and network overhead.

For new objects entering the overlapping region, there will be a delay before the new object appears on both servers, network delays would affect this and could lead to the two simulations diverging significantly. For example, if there are two fast moving objects on a collision course with each other that are expected to collide in the overlapping region. Due to network delays, each server only becomes aware of the other object once the objects have already passed each other, as a result the collision is missed entirely and the result is incorrect compared to a centralised simulation.

3.3.3 Over network object collisions

Objects in the overlapping region interact over the network. This would be done with the two physics in lock-step, i.e. neither progresses until all computation divided between the servers has been completed for that physics step. Servers could divide the computation of collision detection and/or response between servers and then exchange results. This solution would have to simulate objects twice and would require multiple round-trips of messages over the network per physics time-step, but the collision response for objects would be identical on both servers. This approach would be affected by network delays significantly, as not only are multiple round-trips of messages required between server each physics time-step, the simulation won't advance until all messages for that physics time-step have been completed, meaning the frequency of the simulation would be negatively affected. This solution also means the entire simulation can only run as fast as the slowest node as all nodes need to wait for the slowest node to complete each step of the simulation.

3.3.4 Remote objects as static

Treat objects being simulated on another server as static. This would not lead to correct collision response, if compared to a centralised simulation. Momentum is conserved in collisions and no momentum is transferred to static objects, therefore the resulting momentum on the colliding object would not be the same as if the two objects were dynamic (not static) on a single simulation. Half of all collisions would be missed, as collisions would be resolved for the object not being treated as static, but on the remote server, the collision never appears to happen, as the collision has already been resolved when receiving the updated state of the interacting object. This could also lead to jitter, if two objects are in contact (i.e. colliding over multiple frames, such as a box stacked on top of another box), the server in which the collision is resolved on could be different every physics step. A cube stacked on top of a static cube may have a different outcome to a cube with a static cube stacked on top of it.

This approach would be affected by network delays. For example, using the stacked cubes example. Say the two cubes are free-falling and the bottom cube is being simulated on a different server, the position of the bottom cube would only be updated whenever a message was received from the remote server. If there was a long delay between messages from the remote server, for example, from a spike in latency or a sudden large loss in packets, the cube would not be updated and appear to have stopped falling. As the bottom cube has stopped on the receiving server, the top cube remains stationary, as it is in contact with the bottom cube. On the remote server, if messages are being received, the cube would be stationary and appear to be floating, as from its server's perspective, it is in contact with the bottom cube. As a result, both servers have incorrect states compared to a centralised simulation.

3.3.5 All object interactions on the same server

Simulate all objects that are interacting with each other on the same server. Auras could be used to determine which objects are potentially interacting. Auras of objects would need to be exchanged. Solution would need to manage the number of auras exchanged to reduce this overhead.

3.4 Proposed Solution

We term this study's approach Aura Projection (AP). AP tackles the problem of maintaining consistency across regions while ensuring timeliness of a simulation. In particular, AP provides the building blocks of a scalable solution to server-side physics simulations by handling all configurations of boundary cases in migratory approaches.

In AP, an object maintains an aura, an area of interest around the object, that indicates its possible future location and, therefore, aids in identifying near future interactions. In other words, auras act as a request for objects to be migrated. Any object colliding with the aura is migrated to the server the aura came from. The presence of an aura allows:

- The prediction of future hosting requirements allowing time to transfer objects.

- A narrowing of interest in only considering a subset of objects promoting a scalable solution.
- An unhindered physics simulation for existing objects.
- Limiting communications requirements between servers based on focusing message passing overhead on interacting objects (promoting scalability).

In essence, AP ensures that any two objects that may be interacting are always being simulated on the same server. For example, objects C and D illustrated in Fig. 3.1. Object C projects its aura into Server 1, Object D collides with the aura and is migrated to Server 0, allowing continued interaction to occur.

Messages are sent between servers where aura overlap occurs across a geographic boundary. We term this phase of the algorithm as an object projecting its aura onto another server.

In the context of real-time physics simulations, auras appear on the receiving server as a trigger volume (non-physically interacting volume). For each simulation step that an object is projecting an aura, the object's position and calculated aura are sent. As a network optimisation, if the boundary object is not moving, no details are sent and the aura on at a receiving server is unaltered.

When an aura is no longer being projected for an object (boundary not overlapped), the server simulating that object sends a message to any servers receiving the aura to remove the aura of that object.

In summary, objects that are being hosted by a remote server are consistency-aware, through the use of auras. The system doesn't know exactly where the object will be, but can guarantee it will be within the aura by the time an interaction takes place. A small amount of availability is sacrificed (in the region of 10ms) as objects migrate between servers, in order to achieve eventual consistency while maintaining network partitioning. In terms of collisions between objects, AP provides strong consistency. All collisions between objects only take place on a single server, removing the possibility of inconsistencies between servers. Network partitioning is sacrificed in order to achieve strong consistency.

3.5 Aura Calculation

Calculating an aura size is key to creating a scalable solution balanced against consistency:

1. Auras that are too small will result in a faster simulation but with more missed interactions.
2. Auras that are too large will include more calculations that are simply not required, increasing server load and decreasing scalability.

Therefore, how auras are calculated will be carefully described and justified.

Spheres are used to represent auras. This bounding volume is computationally efficient as rotational calculations are not required in determining its correct alignment. Auras accommodate the displacement of objects using an estimated network latency based on historic monitoring, in addition to the distance a remote object may be predicted to penetrate an aura (considering velocities).

Rather than every object maintaining an aura, which would add overhead to the physics simulation, when an aura is exchanged it accounts for a potential remote object's aura. When an object comes into the aura distance of the region boundary, an aura is created for the object and the aura starts being broadcast to neighbouring servers.

Fig. 3.2 demonstrates the sequence of events involved when an aura is sent, received, and collided with, resulting in an object migration.

To lower inconsistencies during migration, the following user-defined tolerances are used for the aura calculations:

- V_t - maximum speed of simulated objects.
- T_L - maximum network latency time.
- T_F - maximum frame-time (time allowed for combined frame update and network update).

If velocities, latencies, or frame-time are above these tolerances, then stability is no longer guaranteed. This enables AP to deterministically indicate to the overall simulation when latency may be influencing the mutually consistent views of the servers, which in turn manifests as errors in the physics.

To calculate the radius of an aura, the maximum distances travelled by objects within the delay time between simulations must first be calculated. To calculate the maximum distance an object may travel for a given time, the following formula is used: $\triangle s = v \cdot \triangle t$, where $\triangle s$ is distance, v is speed, and $\triangle t$ is time. In this context, $\triangle v$ is substituted with maximum speed tolerance, and $\triangle t$ will be a multiple of the physics step time (to accommodate discrete time step calculations in the physics engine). $\triangle t$ can be calculated using the maximum frame-time and maximum latency tolerance, discussed below.

There is a time delay between the aura being created on a host and being created on a receiving server, which is made up of: up to one frame before sending the aura creation message; inter-server latency; up to one frame from the message being received and acted upon; the time delay between the aura being created and the detection of the collision between a potential remote object and the aura by the physics update step. This last time delay is accounted for by rounding the previous delays up to the nearest physics time step.

Using the time delays mentioned above and substituting the relevant tolerances, the maximum displacement time (Max_{DT}) of a remote object is calculated using the following equation:

$$T_R = \left\lceil \frac{2 \cdot T_F + T_L}{T_P} \right\rceil T_P \tag{3.1}$$

T_R is the Max_{DT} of a remote object, T_F is the frame-time tolerance, T_L is the latency tolerance and T_P is the physics step time.

The aura has to account for the maximum displacement of both the object on the host server and a potential object on the remote server. The total Max_{DT} will be the sum of: a) the Max_{DT} of a remote object, subtracting one physics time step, as the aura only needs to account for the displacement of the host object after the creation of the aura; b) The following time delays: a time delay of up to one frame time between the physics step of the remote server and the remote server update loop sending the migration message from the migration buffer; a time delay of the latency between servers; a time delay of up to one frame time between the message being received by a host and a host acting on the migration message and creating the migrated object in the physics engine; a delay between the migrated object being created and the collision being detected by the physics update step. This last time delay is accounted for by rounding the previous delays up to the nearest physics time step.

The Max_{DT} of the object on the host that is projecting the aura is therefore calculated using the following equation:

$$T_H = T_R - T_P + \left\lceil \frac{2 \cdot T_F + T_L}{T_P} \right\rceil T_P \tag{3.2}$$

T_H is the Max_{DT} of the object on the host.

The total Max_{DT} is therefore $T_R + T_H$, which can be simplified to the following equation:

$$T_T = (3\left\lceil \frac{2 \cdot T_F + T_L}{T_P} \right\rceil - 1)T_P \tag{3.3}$$

T_T is the total Max_{DT}.

The aura of an object can then be calculated using the following equation:

$$R_a = R_o + (V_t \cdot T_T) \tag{3.4}$$

R_a is the radius of the aura in m, R_o is the bounding sphere of the object, V_t is the speed tolerance in $m{\cdot}s^{-1}$ and T_T is the total Max_{DT} in s.

3.6 Sub-optimal object hosting

As objects within each others auras need to be simulated on the same server, this can lead to objects being deep within another server's region (objects that are far from the region boundary).

Clusters of objects can therefore end up being simulated with all but one of their objects in their host server's region. This is sub-optimal, as the more auras that are being maintained/exchanged, the higher the computational cost.

The solution to this problem is to dynamically move the server-region boundaries to prevent clusters of objects having the majority of the objects outside of their host server region or the server-region boundaries could be moved in such a way as to avoid intersecting large clusters of objects entirely. However, this solution is not yet implemented in AP.

3.7 Thrashing

The common term 'thrashing' will be used to describe the following two scenarios that must be considered.

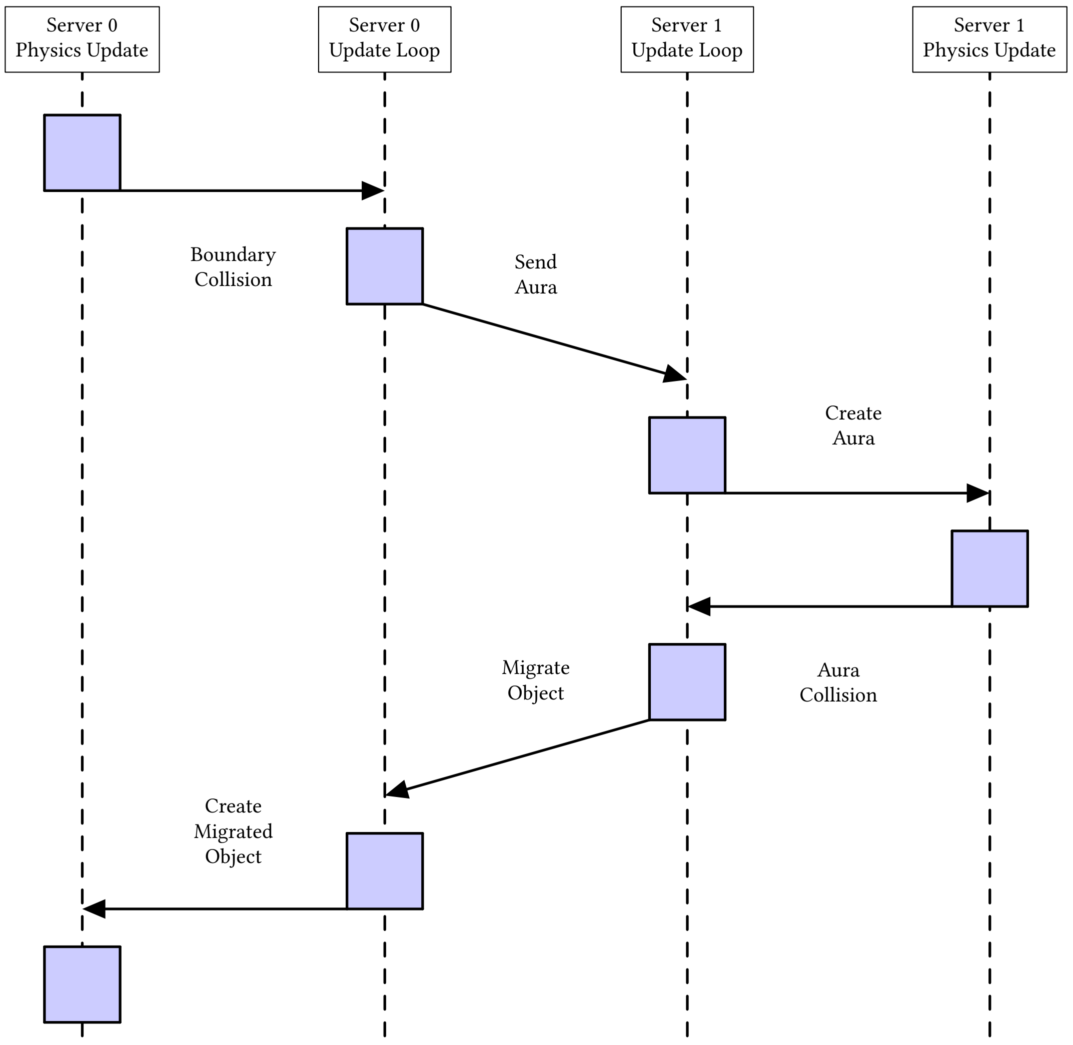


Fig. 3.2 Aura and migration sequence diagram

1. 3-Object Thrashing - An object may be overlapping two auras from different servers. Given appropriate velocity, this could result in object migration, followed by aura collision, followed by migration and then repeating the process again. For example, Objects K, L and M in Fig. 3.1. Object K lies inside the auras of both Object L and M.

In order to prevent this thrashing, when an object is migrated, it is also migrated with any objects found that lie within the aura of the object or have an existing aura which overlaps the migrating object's aura. This is carried out recursively to ensure all objects that are overlapping are migrated at the same time.

2. 2-Object Thrashing - Two objects on different servers may both trigger the migration of the other. For example, Objects C and D in Fig. 3.1. Suppose the two objects are moving towards each other. Object C collides with the aura of object D and begins migrating to Server 1. Before Object C is received (and the message processed), object D collides with the aura of Object C and begins migrating to Server 0. The two objects remain on different servers and the process can repeat again, indefinitely.

When Object C is received, the object is created in Server 0's physics simulation and the aura for that object is removed from Server 0's physics simulation. The next physics time-step on Server 0, the object is within the aura distance of the boundary with Server 1, a message for the aura of object C will added to the send buffer for Server 1. The next network update, the message will be sent to Server 1. If Object C intersects an aura from Server 1, in this case Object D's aura, it will be added to the object send buffer for Server 1. The next network update, the message will be sent to Server 1. The message for Object C's aura is received by Server 1 and processed on the next network update, when processed, a trigger volume for Object C's aura will be created in the physics engine of Server 1. The next physics time-step on Server 1, any objects within that aura will call a trigger callback, in this case Object D, indicating that object D has collided with an aura and so should be migrated to Server 0. Object D is added to the object send buffer for Server 0 and the next network update, Object D is migrated.

In order for this process to repeat, both objects must have their migrations triggered (during a physics update) before the message of the other object migration has been received and processed and a physics time step has occurred to remove the aura trigger volume from the physics simulation.

In order for this to occur, each object must have its aura created (first physics time step after arrival) before the aura of the other object or the other object itself has been received (if the aura is received, it will trigger the migration immediately, so an aura will not be sent, if the other object is received, the aura will be removed). Each object has its aura received by the other server, triggering the migration of the other object. The cycle is broken once an object is received and its aura removed before the other object has its migration triggered when a physics time step occurs.

3.8 Islands

When an object traverses a region boundary, the object is migrated only if it does not overlap an aura projected by an object from the same server. In the context of migrations, an island is defined as two or more objects located outside of their host server's region (i.e. have traversed a region boundary) but are each within the aura projected from objects owned by the same host. For example, Objects H, I, and J illustrated in Fig. 3.1. No objects in the island are migrated as each object is within the aura of another object in the island. This should be prevented as it causes processing and networking overhead and is unnecessary as all objects lie within the region of Server 2 and are not interacting with any other objects from Server 0.

To prevent islands, a search is performed at each time step to determine if an object is part of an island. A search is performed to determine if a potential migratory object is within a group of objects with overlapping auras, of which none are positioned within the hosting server region. If the group of objects has no members within the hosting server region then the object is part of an island and the entire island of objects is migrated. Otherwise, no action is taken. For example, in Fig. 3.1 Object I has overlapping auras with Objects H and J. H and J are checked for overlap with the Server 0 - Server 1 region boundary and for overlap with auras from Server 0 that are intersecting the Server 0 - Server 1 boundary. In this scenario, H and J are found to be part of an island, so H, I, and J are all migrated to Server 2. If objects are found to not be part of an island, no action is taken. For example, Objects E, F, and G in Fig. 3.1. Object F is found to have an overlapping aura with Object E, but Object E intersects the Server

0 - Server 1 boundary, so E, F, and G are found to not be part of an island and therefore no action is taken. This solution is shown in Algorithm 2 and the problem of islands can therefore be considered solved.

3.9 Corner Case

This section discusses how AP handles corner cases, i.e., where the boundary of more than two servers meet.

An example of the corner case is illustrated in Fig. 3.1. Object N, hosted on Server 2, sends an aura to Server 0. Object O, on Server 1, sends an aura to Server 0. Object N's aura overlaps with Object O's on server 0. Server 2 is unaware of Object O and Server 1 is unaware of Object N. As the two auras overlap, there is a potential interaction between the objects, yet the two objects remain on separate servers. To solve this problem, auras from boundaries between two neighbouring servers are shared. In the region layout in Fig. 3.1, auras from a boundary are received by interested servers. In the Object N and O example, Server 2 would send Object N's aura to Server 1; when Object O collides with Object N's aura, migration to Server 2 occurs.

A server has to receive the auras of all objects from neighbouring boundaries, as objects being simulated by a server can exist anywhere in a neighbouring server's region. Our assumption is that a region a server simulates is sufficiently large enough to prevent the aura overlap of objects from non-neighbouring servers.

3.10 Benefits of Proposed Solution

Aura Projection has the following benefits over other considered solutions. AP guarantees strong consistency of collisions, meaning diverging results on different servers are prevented. Objects are only simulated once on one server, instead of being simulated on multiple servers, requiring more computation time per object. No synchronisation is required between servers; as each server is running an entirely autonomous simulation, there is no need for any synchronisation to take place between servers.

3.11 Limits of Proposed Solution

Performance of the system will be measured using the highest cost across all servers, which will be referred to as *costmax*. If the addition of one or more servers (and/or re-partitioning of servers) cannot be done without causing a higher *costmax*, the system cannot be scaled up further.

It is possible to predict whether or not a new configuration has a higher or lower *costmax* using the following process. Let B:N be the performance cost ratio of boundary to non-boundary objects (this cost ratio will differ for different simulations and may either be a constant ratio or a function, but we will assume it is a constant for this estimate). Due to boundary objects having additional processing and networking overhead, we can assume the B:N ratio will always be greater than 1. The performance cost can then be estimated for each server in the new configuration by summing the boundary and non-boundary objects and using the B:N cost estimate. If this results in a lower *costmax* the performance of the system will be increased.

Let $cost(x)$ be the performance cost of a server x. Let Obj_{bdry} and $Obj_{nonbdry}$ be the number of boundary and non-boundary objects respectively for a given server x.

$$cost(x) = (Obj_{bdry} \cdot B : N) + Obj_{nonbdry} \tag{3.5}$$

Let N be the number of servers. The $cost_max$ of a server can be calculated using the following equation:

$$costmax = max(cost(1), ..., cost(N)) \tag{3.6}$$

Let $costmax_n$ be the current *costmax* of the system and $costmax_{n+1}$ be the *costmax* of a potential new configuration.

$$costmax_{n+1} < costmax_n \tag{3.7}$$

The $costmax_{n+1}$ must be less than $costmax_n$ in order for a performance gain to occur. If no possible $costmax_{n+1}$ with a lower value exists, the overall performance of the system cannot be increased and the system will have reached the limit of scalability.

Example 1: Let 1 non-boundary object have the cost value of 1 and suppose the B:N ratio is a constant 2:1. The example begins with 1 server with 1000 non-boundary objects and the $costmax_n$ is 1000. 1 server is added, and the simulation is partitioned so that 500 non-boundary objects exist on each. The $costmax_{n+1}$ is 500. 500<1000, so the new configuration results in better overall performance of the system.

Example 2: The example begins with 2 servers with 1000 non-boundary objects each and the $costmax_n$ is 1000. 1 server is added, and the simulation is partitioned. The best partitioning that can be achieved results in 1 server having 250 non-boundary and 400 boundary objects. The $costmax_{n+1}$ is (400*B:N)+250 = 1050. 1050>1000, so the new configuration results in worse performance. As this is the best partitioning that could be achieved the system cannot be scaled up further.

Using the logic above and the assumptions made, it can be deduced that the main limiting factor of Aura Projection is the number of unavoidable boundary objects present on a single server in the simulation.

3.12 Problem Definition Summary

In this chapter, the following points have been addressed:

- The requirements of a solution to scalable real-time physics.
- The challenges of distributing a real-time physics engine and potential solutions.
- A description and justification of AP, including the aura calculation.

- The implemented solutions to the problems of: 3-object thrashing; islands; and the corner case.
- The proposed solutions to: sub-optimal object hosting and 2-object thrashing, which are not yet implemented.

Chapter 4

Implementation

The implementation of AP will be discussed in this section. The following work is based on the implementation of AP used in [12], however, with many newer additions. Full details of the implementation of AP have been included to give a complete and whole solution.

The following points will be looked at in detail:

- An overview of the system architecture: How the servers are deployed in the cloud and how they communicate with a client; a breakdown of the components of each server; and potential deployments to be investigated in the future.
- Aura Implementation
- The algorithms used by the servers to implement AP.
- The life cycle of a simulated object as a finite state machine (FSM).
- The features and implementation details of the visualiser, including a description of its features and future work to be carried out on the visualiser.

4.1 System Architecture

The simulation space is partitioned into regions, with each region consisting of its own instance of PhysX running on a dedicated GPU-enabled machine in the cloud. The boundary between regions are defined as vertical planes, two-dimensionally dividing the simulation space into one region per server. The network library RakNet is used for all message passing between servers. RakNet ensures messages exhibit best effort and are received in sent order. For the purposes of simplicity, the network tick rate (the rate at which RakNet is updated) is the same as the main update rate (the rate at which the main logic of the program is updated, such as processing user input and handling callbacks from PhysX). This removes the need for the aura calculation to account for an extra time delay between network update and main update.

When objects project auras they are added to a send aura buffer that is sent to all servers associated to the boundary of concern. Each object has a unique identifier (ID). When an aura is received by a server, an aura is created if it does not already exist, otherwise the aura is updated using the data received. When an object is no longer projecting its aura, the ID of that object is added to the delete buffer which is then sent to all servers neighbouring the boundary.

When objects fully traverse region boundaries, they are added to a migration buffer with all information required to duplicate an object at a neighbouring server. The contents of each migration buffer are sent to the associated server, which will then be responsible for hosting the migrated objects. When migration messages are received, the migrated objects are created within the receiving server's simulation.

Clients connect to all servers and are provided with a streamed visualisation of the simulation in real-time. Clients may also interact with and influence the simulation through client-created replicas and static objects, and RPCs, providing a comprehensive solution for real-time interactive physics. The client system was built using the Unreal Engine. Once a client is connected, the position and states (replicas) of all objects in the simulation are sent from each server to the client via the RakNet Replica Manager.

An overview of the system architecture is illustrated in Fig. 4.1.

For the implementation of this project, Amazon Web Services (AWS) was selected as the cloud provider, giving access to GPU-enabled instances (G2 instances). PhysX makes us of GPU-acceleration, including for the simulation of rigid-bodies [68]. The PhysX Samples project (a publicly available project for demonstrating the features and capabilities of PhysX) was used as a starting point for this project, as this was a relatively minimal project that uses PhysX but with some useful features for use in development of this project, such as a basic renderer and many helper methods for using PhysX. For the purposes of automated testing and for running experiments, the testing library, Gtest, was used.

4.1.1 Server-Region Topology

Server-region topologies and how they affect the communication requirements between servers will now be discussed. AP allows for the simulation space to be partitioned into any topology. In terms of communication, all servers need to communicate with all of their neighbours in order to satisfy the solution to the corner case, as described in 3.9. We define a neighbour as any server that shares a region boundary, including at a corner.

The simplest topology that can be used is a column layout, as shown in Fig. 4.2. In column layout, servers have at most 2 neighbours, this topology therefore has the fewest server connection requirements. It should be noted however, that fewer server connection requirements doesn't necessarily mean less network traffic as the traffic between 2 servers is dependent on the activity of the simulation near the server boundary.

Any topology more complex than columns will result in corners being present in the topology. The simplest corner that can exist is between three servers, as shown in Fig. 4.3a. All three servers must communicate when a corner between three servers exists, this is required for the solution to the corner case (as described in 3.9). For example, in Fig. 4.3a, a cluster of objects being hosted by server 2 may exist across the boundary between server 0, objects on the edge of the cluster may be positioned deep into server 0's region. In addition a cluster of objects being hosted on server 1 may exist across the boundary between server 1 and server

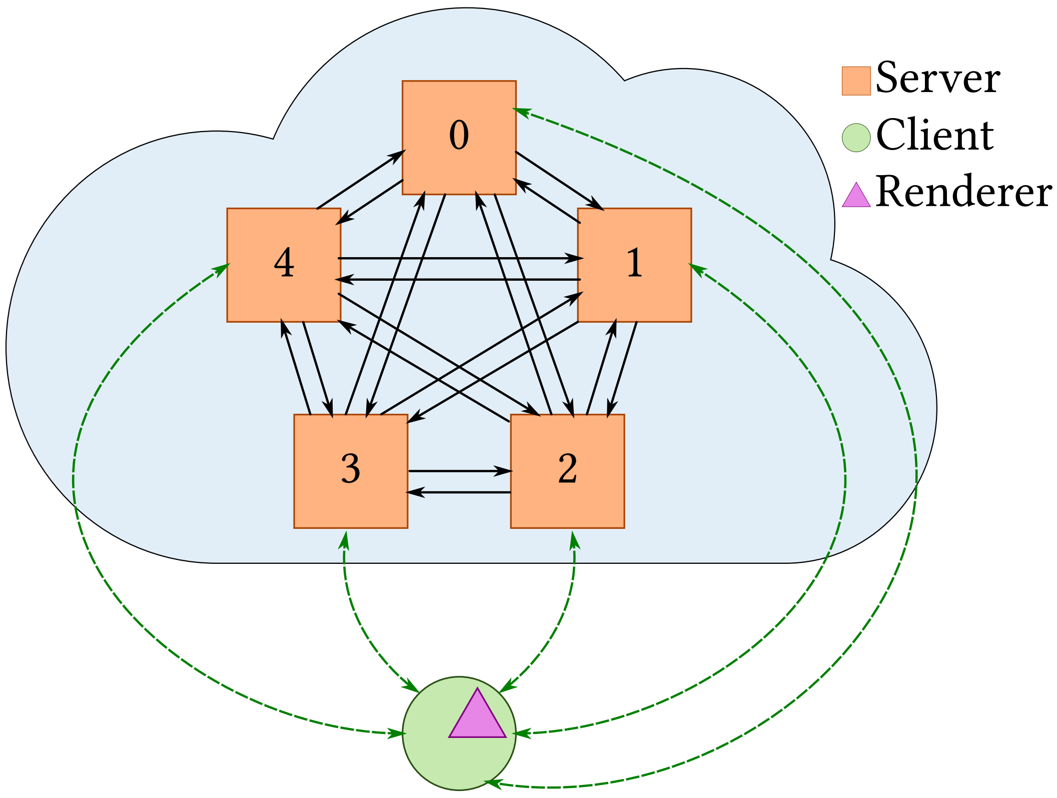


Fig. 4.1 Network Topology. Each server is an EC2 instance in the cloud, running AP, which includes an instance of PhysX. The client is a local machine running the visualiser built in Unreal Engine. Servers connect to each other in order to exchange messages for aura creation/update and deletion and the migration of objects. The client receives replica updates from each server as well as sending client-created replicas, static objects and RPCs.

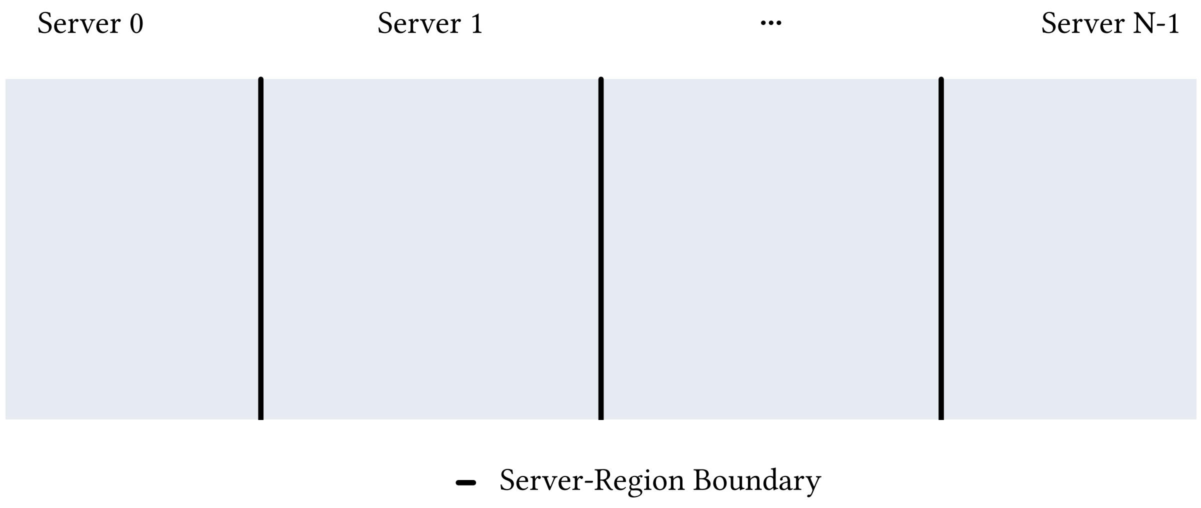
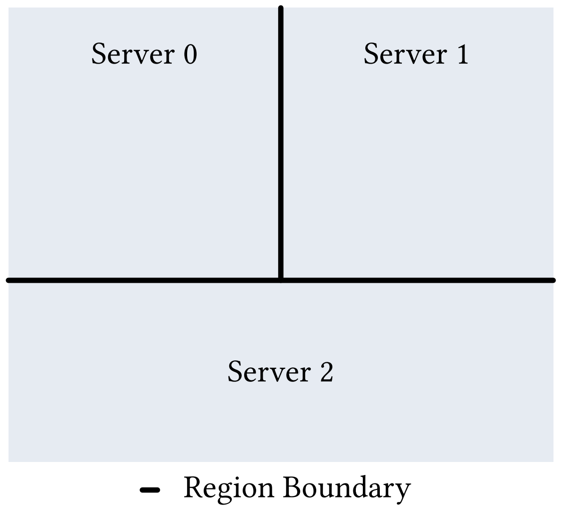
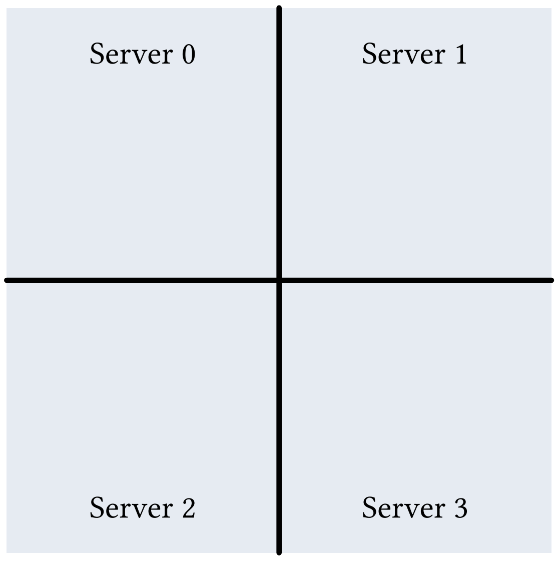


Fig. 4.2 Column topology of server regions

0 with some objects in the cluster being deep into server 0's region. Without the auras from the server 0 - server 1 boundary being sent to server 2 and vice versa, the two clusters will be unaware of each other's existence. From Server 0's perspective, the two clusters may have overlapping auras, and so should be being simulated on the same server. In order to prevent this, the auras from the boundaries of all neighbours need to be exchanged.



(a) 3 server corner topology



(b) 4 server corner topology

Fig. 4.3 Corner topologies of server regions

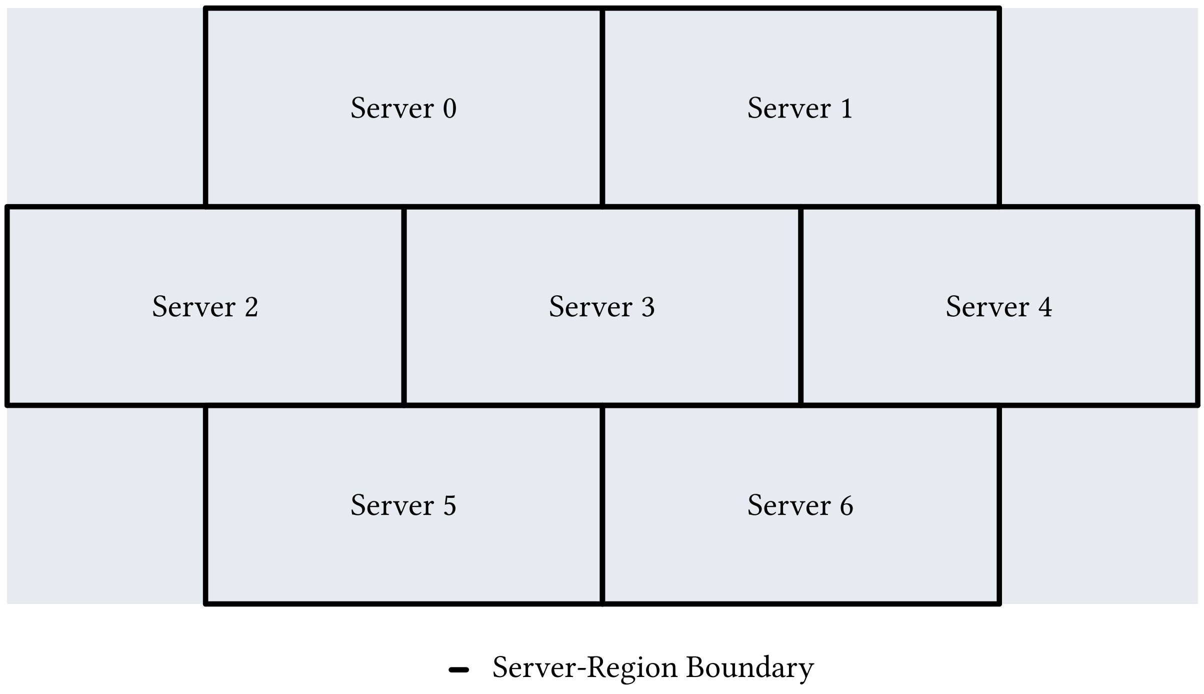


Fig. 4.4 An example topology using only 3-server corners

An example of a topology that includes corners with no more than 3 servers neighbouring is illustrated in Fig. 4.4. In this topology, the maximum number of server connections required would be 6, as server 3 has 6 neighbouring servers. It should be noted that not all servers need to communicate with each other, for example server 0 does not neighbour with servers 4 or 6, so no communication is needed.

An example of a topology that uses 4-server corners is illustrated in Fig. 4.5. In this topology the maximum number of server connections required would be 8, as server 4 has 8 neighbouring nodes. It should be noted that not all servers need to communicate with each other, for example server 0 does not need to communicate with servers 2 or 5 as they are not neighbouring servers.

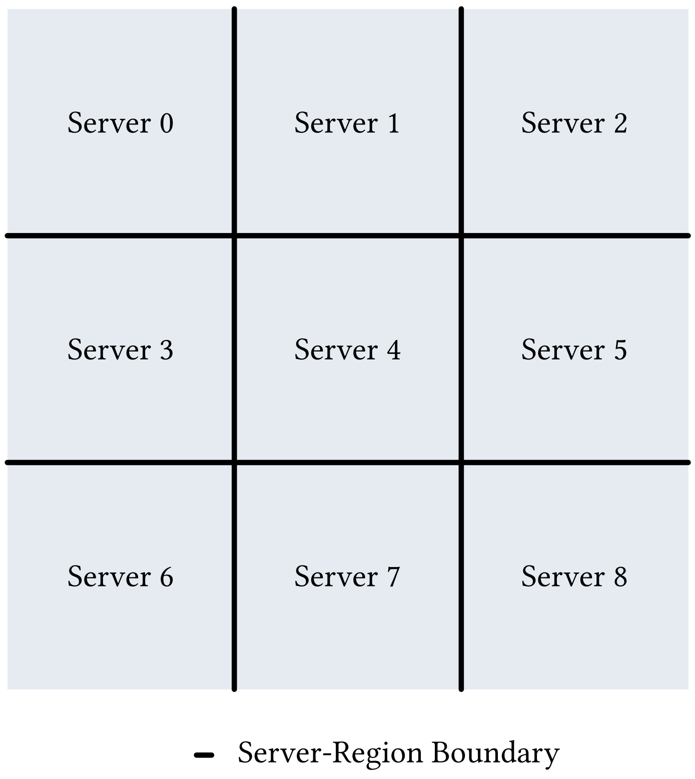


Fig. 4.5 An example topology using 4-server corners

4.1.2 Individual Server Breakdown

The components that make up an individual server, as illustrated in 4.6, will now be discussed. Each server is connected to the client (C_1) and all neighbouring servers $S_1..S_N$ (the definition of neighbouring nodes is discussed in 4.1.1). The server connects to both of these types through the RakNet library. RakNet includes the Replica Manager and RPCs used by client.

For each boundary with a neighbouring server, a server has three message buffers:

1. A migration buffer for sending objects to the server corresponding with that boundary. Each element in the buffer is a message for one object migration. Each migration message contains all data necessary to reproduce the object on the receiving server.

2. An aura creation/update buffer, each element of the buffer is a message for the creation or update of an aura and includes ID, position and radius of the aura. The contents of this buffer are sent to all neighbouring servers, not just the server corresponding with this boundary. The messages structure for both creation and update are identical as the receiving server creates a new aura if one does not already exist for the corresponding object (determined using the ID), otherwise the message is just used to update the state of the existing aura.

3. An aura delete buffer, each element of the buffer is a message for the deletion of an aura, which consists of the aura's ID. The contents of this buffer are sent to all neighbouring servers, not just the server corresponding with this boundary.

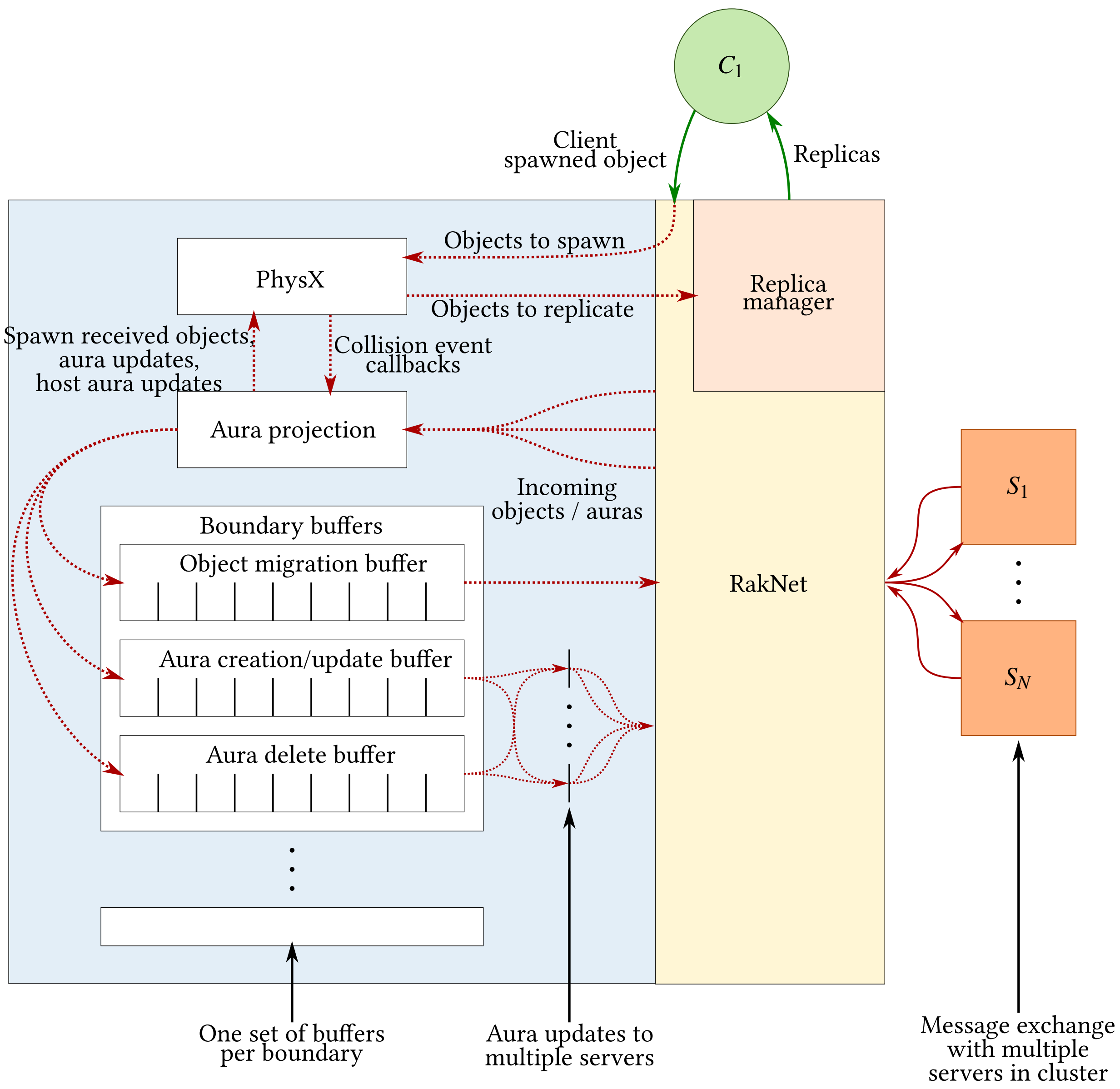

C_1
Client spawned object
Replicas
Objects to spawn
PhysX
Objects to replicate
Replica manager
Spawn received objects, aura updates, host aura updates
Collision event callbacks
Aura projection
Incoming objects / auras
Boundary buffers
Object migration buffer
Aura creation/update buffer
Aura delete buffer
RakNet
S_1
S_N
One set of buffers per boundary
Aura updates to multiple servers
Message exchange with multiple servers in cluster

Fig. 4.6 Server Breakdown. This diagram shows a breakdown of the components of an individual server.

Aura Projection receives and processes the messages received from the neighbouring servers. This includes the creation of migrating objects in the PhysX simulation, the creation and updates of the trigger volumes for auras and the deletion of auras in the PhysX simulation.

PhysX provides callbacks for collision events, such as when an object collides with an object's aura or collides with a trigger boundary aura.

The Replica manager tracks the states of objects that are being replicated and sends changes of state to the client. The Replica manager also receives replica creation messages from the client (as described in 4.5.5)

RPCs allow the client to call procedures on the server, including the creation of server-created replicas (as described in 4.5.3). RPCs enable the client to interact with the servers through various features described in 4.5.6

4.1.3 Future Architecture

System architectures which could be investigated in the future will now be proposed and discussed. A limitation of the current architecture is that a client must connect to all servers; servers that are responsible for running the simulation also have the overhead of connecting to many clients. An alternative to this is to make the use of an ‘overseer’ node, which acts as a middle-man for communication with all the servers, as illustrated in Fig. 4.7. The overseer node is co-located in the cloud with the simulation servers. The clients all connect to the overseer node, but all the servers only connect to one overseer node, reducing the overhead of many clients connecting to a server, freeing up computational resources on each simulation server, allowing more resources to be used for the simulation itself. An additional advantage of this approach is it allows for some fault-tolerance. If one or more simulation servers were to fail, the overseer node has a copy of the states of all objects from those servers, meaning those spatial partitions of the simulation could be restored. Another advantage of overseers is it can potentially allow some computational work to be carried out on the overseer nodes instead of the simulation nodes, such as simulation queries and non-simulation work, such as game logic, interest management.

The disadvantage of using an overseer node is the additional latency between client and simulation server, messages now have to be processed by the overseer node between the simulation servers and clients. For example, if a replicated object moves in one of the simulation servers, the update message for that object has to be processed and sent by the simulation server, sent over the network to the overseer, received and processed by the overseer node and then sent over the network to all connected clients. The additional latency is due to the communication latency between simulation server and overseer node and the time taken to process messages on the overseer node.

The limitation of the use of an overseer node is the number of clients that can connect is limited to the amount that can be handled by a single overseer node. In order to enable a scalable number of clients, multiple overseer nodes could be employed, dividing the workload between overseer nodes, allowing for many more clients to connect. This is illustrated in Fig. 4.8.

Overseer nodes could be located at the edge, geographically closer to the clients, as illustrated in Fig. 4.9. The advantage of this approach is, there is a reduced latency between clients and overseers, meaning lower delay times between clients and the workload being carried out by the overseers, such as game logic. The disadvantage of this approach is that there is higher latency between overseer nodes, which are located far apart, geographically.

In the current system architecture, the rendering work is carried out on the client machine. An alternative to this is to have the rendering performed on dedicated render nodes in the cloud, as illustrated in Fig. 4.10. The results video could then be streamed to clients. This has the advantage of enabling low computational power devices, such as mobile devices, to display higher quality graphics, beyond what would be achievable on low computational power devices. Additionally, it means there is very low latency between the physics simulation and the renderer, meaning that what is displayed is closer to the actual current state of the physics

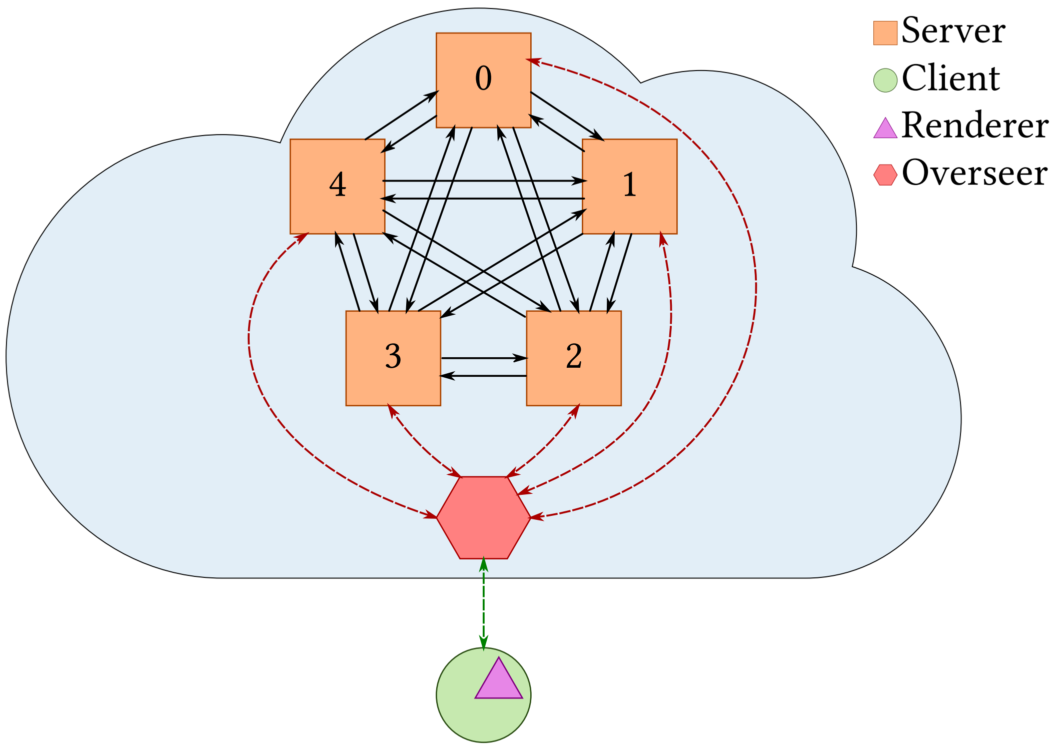


Fig. 4.7 Overseer node.

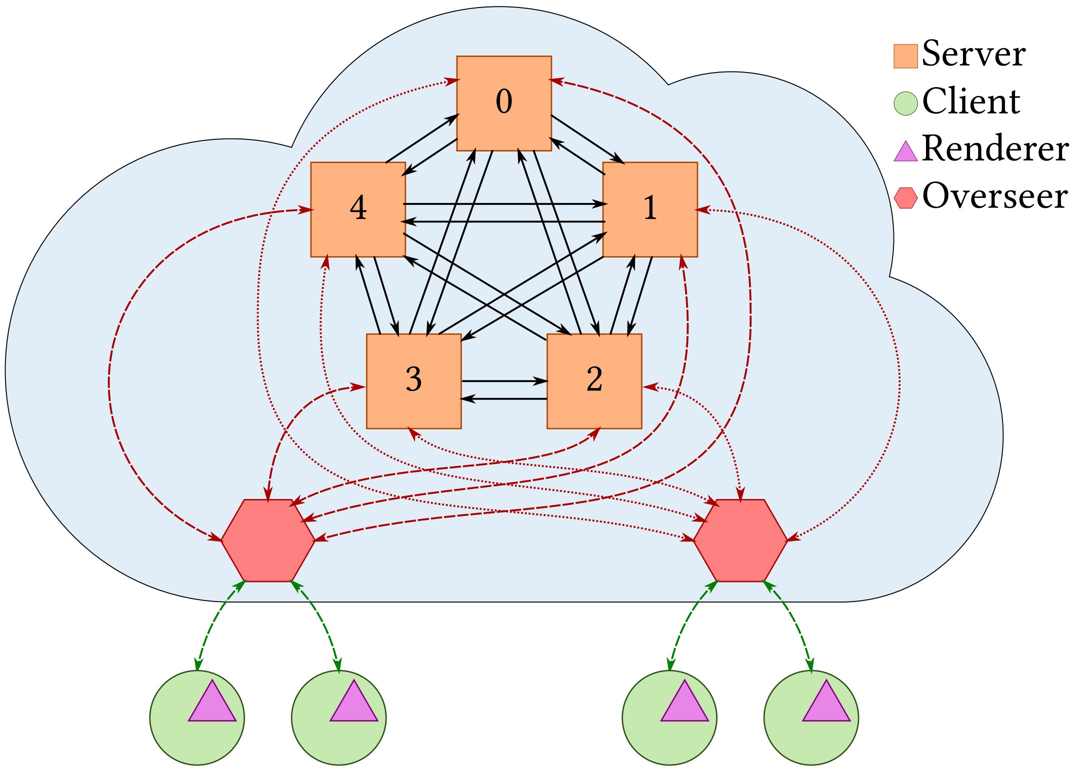


Fig. 4.8 Multiple overseer nodes for scalable number of clients.

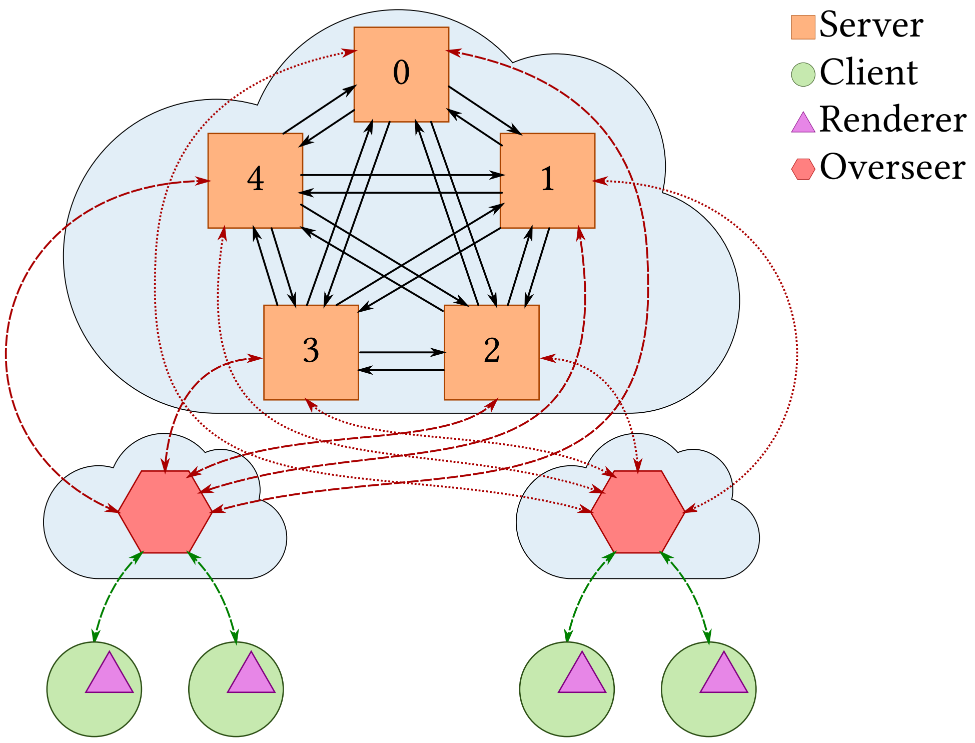


Fig. 4.9 Edge-based overseer nodes.

simulation. The disadvantage of this approach is the latency time between the client and the renderer.

4.2 Aura Implementation

How the auras in AP are implemented will now be described at a high-level, before a formal description of the algorithms using pseudo-code are given in 4.3. Auras are implemented through the use of trigger volumes (non-physically interacting volumes, that generate trigger collision callbacks). Two trigger volumes are used for different purposes. The first is used for auras that are being received from remote servers. When an aura is received, a trigger volume, consisting of a sphere, is created at the corresponding position with a given radius. The second trigger volume used is by the server-region boundaries. Each server-region boundary has a trigger volume that extends inside the server's region, it extends the same distance as the aura radius, minus the bounding sphere of an object. This is used to determine when an object should begin and stop having its aura sent to neighbouring servers and avoid the need for per-object checks for proximity to the server-region boundary. When an object collides with the aura trigger boundary, the aura begins to be sent, when an object leaves the aura trigger boundary, an aura delete message is sent. Each server-region boundary has a trigger volume, objects may collide with multiple aura trigger boundaries, meaning auras can be sent to multiple servers, as illustrated in Fig. 4.12.

It should be noted that an object's bounding sphere may be intersecting the boundary trigger, but the object itself may not, resulting in no aura being broadcast. However, this will not lead to problems with auras not being broadcast when a possible interaction could occur, as

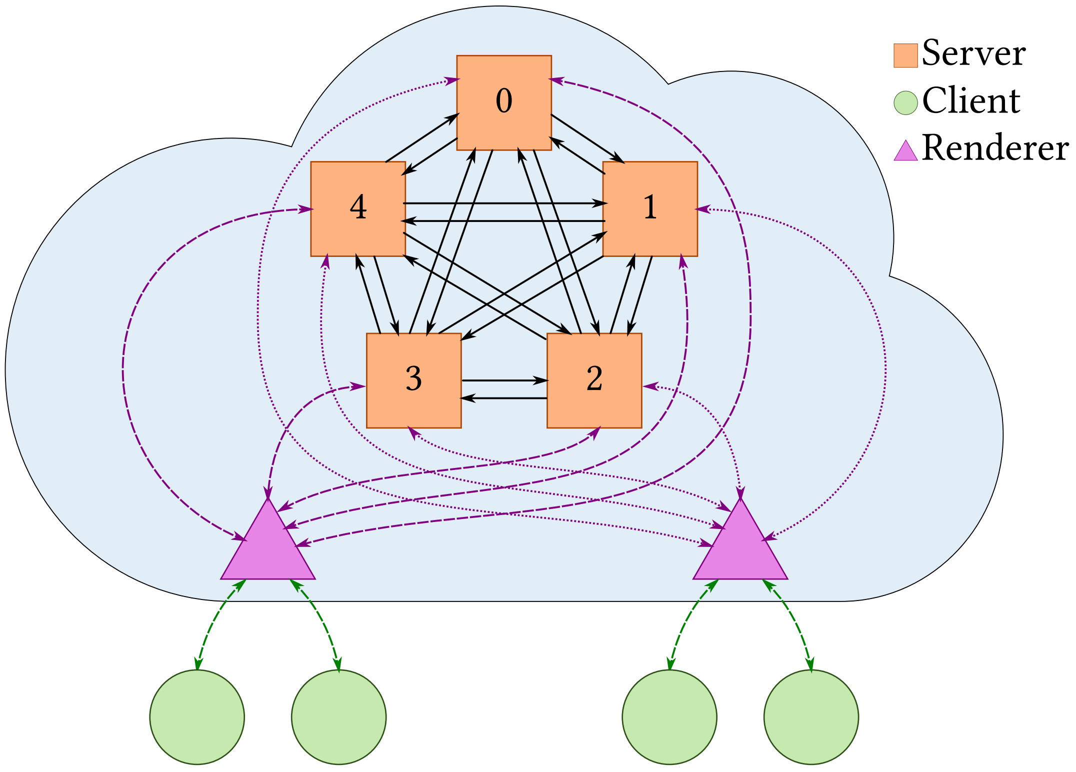


Fig. 4.10 Dedicated render nodes in cloud.

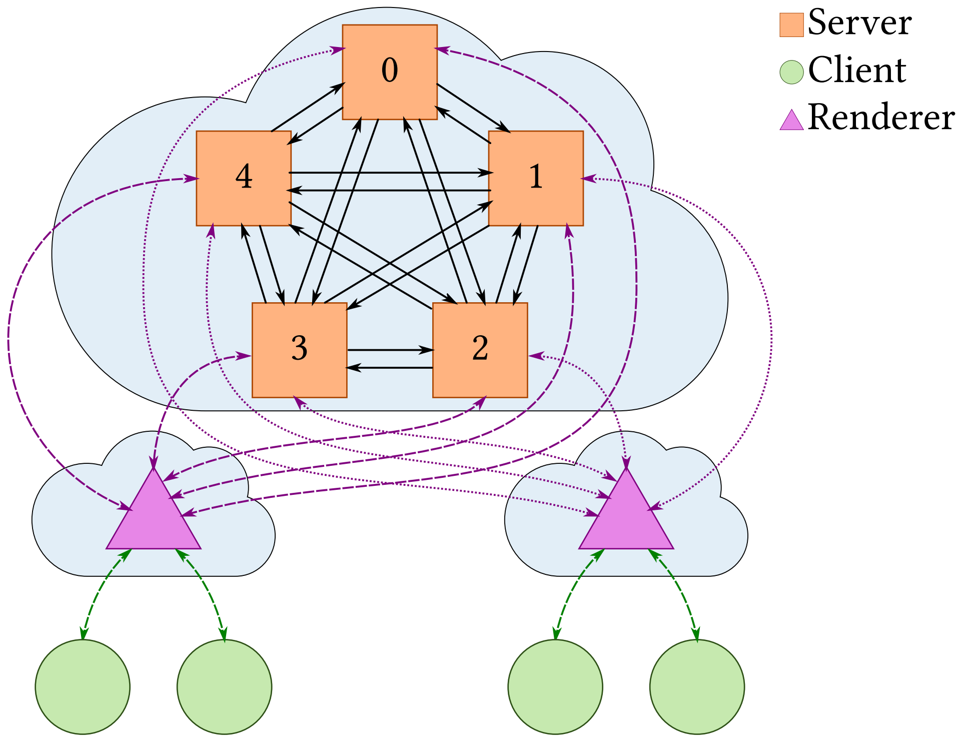


Fig. 4.11 Dedicated render nodes in edge.

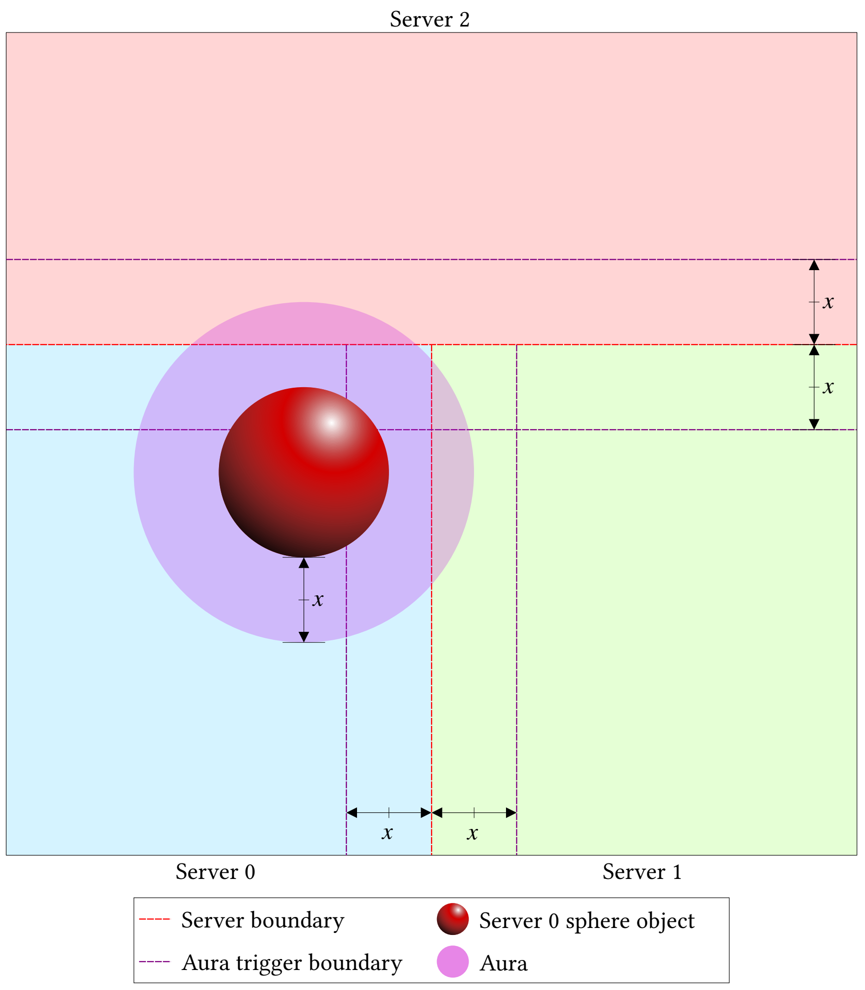


Fig. 4.12 Aura Projection at a 3-way corner intersection. When an object collides with a server aura trigger boundary, the aura begins to be sent to the corresponding neighbouring server. In this scenario, an object from server 0 is colliding with two aura trigger boundaries from servers 1 and 2, meaning the aura is sent to both servers 1 and 2. x is the aura distance beyond the boundary sphere of an object and the distance used by the aura trigger boundaries.

the bounding sphere is a simplification used for the aura. Using the actual shape of the object to determine when to begin broadcasting the aura is more accurate than the bounding sphere simplification.

When a server begins communicating an aura, a message is added to the aura creation/update buffer. This message consists of the position of the centre of the aura and the radius of the aura. The contents of the buffer are then broadcast to all neighbouring servers. When aura creation/update messages are received and an aura for that object does not already exist, a new aura, a sphere trigger volume, is created at that position with the given radius. Whenever an object projecting an aura moves, an update message is added to the aura creation/update buffer, containing the new position and radius of the aura (aura radius is currently fixed, but may become dynamic in the future). The contents of the buffer are broadcast to all neighbouring servers. When the message is received, the positions of the auras are updated. If an object leaves aura trigger boundary, either by moving away from the boundary or by being migrated to another server, a delete aura message is added to the aura delete buffer. The contents of the buffer are then broadcast to all neighbouring servers. When the message is received, the aura trigger volume is removed from the simulation.

When objects collide with an aura trigger volume from an object on a remote server, a callback from PhysX is generated. When these callbacks are processed by AP, the colliding object is added to the migration buffer for the server the aura is being sent from. In addition, a recursive search is carried out to identify all objects within the cluster (a group of objects with overlapping auras). The entire cluster is added to the migration buffer, so that all objects in the cluster are migrated at once. The contents of the buffer are then sent to the corresponding server.

In addition to auras existing as trigger volumes representing auras of remote objects, servers also track the auras of objects that they are hosting, that are having their auras broadcast. These are used to determine clusters of objects, so that clusters of objects are migrated together, at the same time. When an object begins broadcasting an aura, a query-only shape is attached to the object, which represents the object's aura. When an object is a migration candidate, a scene query is carried out, using this query-only shape, to determine which auras overlap the migration candidate's aura. Using a recursive search, involving a scene query for each object found from the prior query, it is possible to determine the objects in the cluster. If the entire cluster is outside of the host server's region, the entire cluster is added to the migration buffer and sent to the server in which the cluster resides. If any object in the cluster is still within the host server's region, no migration takes place. This prevents clusters being repeatedly migrated back and forth between servers.

It is important to note that message order for the state of auras is important. For example, an aura exists for an object hosted on server 0 and is being broadcast to server 1. The object moves away from the boundary and after multiple frames is outside of the boundary trigger. Server 0 sends update message to server 1, to update the position of the aura and finally sends a delete aura message to server 1, when the object exists the boundary trigger. However, if the messages are received in the wrong order and an update message is delivered after the delete message is received and the aura deleted on server 1, server 1 will interpret the update message as a new aura being created, leading to an aura that should not exist. This aura may never be deleted, as the sending object might never send an aura update in the future. In order to prevent this, the messages use the ordered message feature of RakNet, which ensure the messages are read on the receiving server in the same order they are sent from the sending server.

4.3 Algorithms

In this section the algorithms used by AP will be discussed in detail and pseudocode presented. AP can be broken down into 6 main algorithms:

- *AC* - Object migration as a result of an aura collision.
- *BT* - Object migration as a result of boundary traversal.
- *OBC* - Aura creation, when an object collides with a server boundary.
- *OBU* - Aura update, which occurs every update for all existing auras.
- *OBE* - Aura destruction, when an object exits a server boundary.
- *BNU* - Boundary network update, occurs every update for each network connection between servers and exchanges object migrations and aura messages for all boundaries.

AC is called when an object collides with an aura. A recursive search is performed in order to find all objects that would lie within each object's aura, preventing thrashing as discussed in 3.7. Once the recursive search is complete, all objects are added to the send buffer.

Algorithm 1 Object Migrate - Aura Collision (*AC*)

```
1:  procedure OnAuraEnter                                  ▷ A callback on an object
2:      ▷ Track visited objects to prevent infinite recursion
3:      visited := {}
4:      ▷ Recursively send object with objects that would lie within each object's aura
5:      SendWithOverlaps(thisObject, visited)
6:
7:  procedure SendWithOverlaps(object, visited)
8:      ▷ Get objects whose auras overlap this object's aura
9:      overlaps := GetAuraOverlaps(object)
10:
11:     for each object ∈ overlaps do :
12:         if object ∉ visited then:
13:             visited := visited + object
14:             SendWithOverlaps(object, visited)
15:
16:     AddToSendBuffer(object)
```

BT is called when an object traverses a boundary. In order to prevent 'islands' forming (for example Objects H, I and J in Fig. 3.1), a recursive search is carried out to determine if an object is part of an island or not. If an object is found to not be part of an island, the entire cluster of objects is added to the send buffer, otherwise no action is taken. Note that a cluster may consist of only one object, meaning the algorithm handles migration of both single and groups of objects.

Algorithm 2 Object Migrate - Boundary Traverse (*BT*)

```
▷ Update called on each boundary
procedure BoundaryUpdate
    checked := {}                                   ▷ Used to prevent duplicate checks

    ▷ Loop over fully traversed objects from latest update
    for each $object \in traversed$ do :
        if $object \in checked$ then:
            continue
        island := {}
        isIsland := IslandQuery(object, island)
        ▷ IslandQuery() returns true if object is part of an island and a list of objects in the island
        if $isIsland = true$ then:
            SendGroup(island)
        checked := checked + island

function IslandQuery(object, visited)
    visited := visited + object
    if object.overlapsHostRegion = true then
        return false

    ▷ Get objects whose auras overlap this object's aura
    overlaps := GetMutalAuraOverlaps(object)

    ▷ If all objects with overlapping auras are islands, then this object is an island
    isIsland := true
    for each $object \in overlaps$ do :
        if $object \notin visited$ then:
            isIsland &= IslandQuery(object,visited)
    return isIsland
```

OBC is called when an object collides with a boundary. The object's aura is calculated and added to the boundary's send aura buffer. A host aura is also created, to allow for the checking of mutual aura overlaps and prevent thrashing, if this is the first boundary intersection.

OBU is called once per frame that an object is intersecting a boundary. If the object is not 'sleeping', a new aura is calculated and added to the boundary's send buffer and the host aura is updated.

The isSleeping() function returns true if an object is sleeping. From the PhysX documentation: An object is considered 'sleeping' when an actor does not move for a period of time. (The default PhysX period of time is $0.4s$ and this is the value used in our approach). Objects are 'woken up' when they are touched by an awake object.

Algorithm 3 Create Aura - Object boundary collision (*OBC*)

```
1: ▷ A callback on an object, called when an object collides with a boundary
2: procedure OnBoundaryEnter(boundary)
3:     AddToAuraBuffer(boundary, this)
4:
5:     ▷ Create ‘host aura’ so GetMutalAuraOverlaps() will detect this object’s aura
6:     if boundaryIntersections = 0 then
7:         CreateHostAura()
8:     boundaryIntersections := boundaryIntersections + 1
```

Algorithm 4 Update Aura - Object boundary update (*OBU*)

```
1: ▷ A callback on an object, called per step per boundary the object is colliding with
2: procedure onBoundaryUpdate(boundary)
3:     if this.isSleeping = true then
4:         return
5:     ▷ Send Aura Delta
6:     AddToAuraBuffer(boundary, this)
7:     UpdateHostAura()
```

OBE is called when an object is no longer intersecting a boundary. The object is added to the boundary’s remove aura buffer. If the object is no longer intersecting any boundaries, the host aura is deleted.

Algorithm 5 Destroy Aura - Object boundary exit (*OBE*)

```
1: ▷ A callback on an object, called when an object exits a boundary
2: procedure onBoundaryExit(boundary)
3:     AddtoDeleteAuraBuffer()
4:
5:     ▷ If object is no longer sending an aura, no need to keep a ‘host aura’
6:     boundaryIntersections := boundaryIntersections - 1
7:     if boundaryIntersections = 0 then
8:         DeleteHostAura()
```

BNU is called once per network connection between servers. It is responsible for sending and receiving object migrations and auras between servers, including the sending and receiving of auras from boundaries between other neighbouring remote servers.

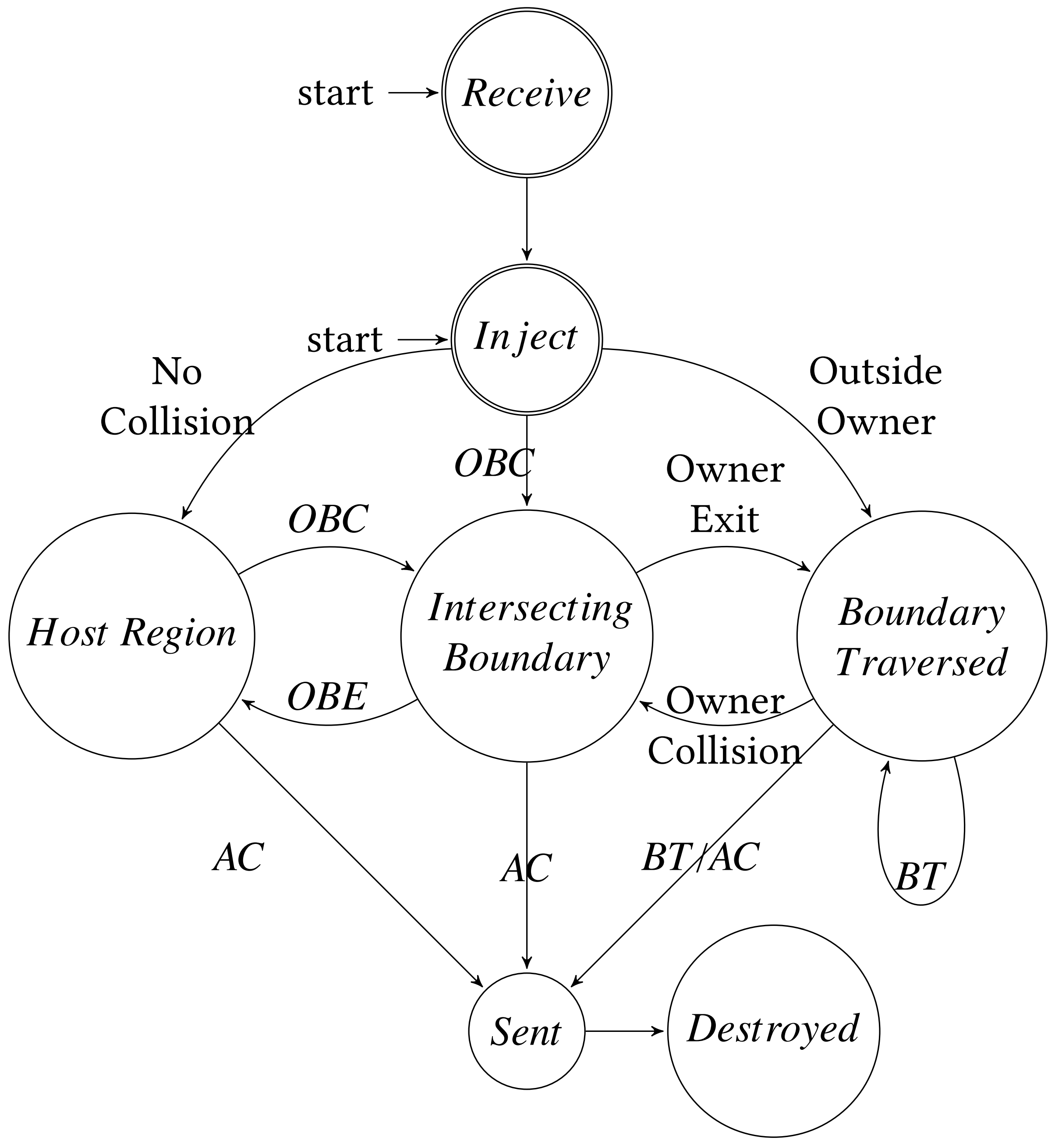
4.4 FSM

The state of an object in the context of AP will now be discussed, i.e. not the physical state of an object (position and rotation), but the state of object as it is being treated by AP.

Objects project auras in two states, the Intersecting Boundary and Boundary Traversed states. When entering these states and every update in which the object has moved while in these states, the object’s aura is added to the send aura buffer, which is broadcast to all neighbouring servers, allowing those server’s to create the object’s aura. When objects leave these

Algorithm 6 Boundary Network Update (*BNU*)

```
1:  ▷ Update called once per network connection
2:  procedure NetworkUpdate
3:      ▷ Exchange migrations with target server
4:      SendObjectsInBuffer
5:      ReceiveObjects
6:
7:      ▷ Send aura state updates to all neighbours
8:      for each $neighbour \in neighbours$ do :
9:          SendAurasInBuffer
10:         SendDeleteAurasInBuffer
11:         ReceiveAuras
12:         ReceiveDeleteAuras
```



states, the object ID is added to the remove aura buffer, which is broadcast to all neighbouring servers and the receiving servers then delete the aura with corresponding ID.

An object has two entry states, it is either received from another server via the network or it is injected into the simulation. Upon receiving an object, the object is then injected into the simulation and is moved into the inject state. Immediately the object's position determines its next state. If the object is not intersecting a server boundary and is not colliding with an aura, the object moves into the Host Region state. If the object is intersecting a sever boundary, it moves into the intersecting boundary state. If the object is outside of the server's region, such as is if the object is injected into a server at position the server is not responsible for, the object changes states to the boundary traversed state.

From the host region state, an object remains in this state until it either collides with a server boundary or a remote object's aura. If colliding with a server boundary, it is moved into the Intersecting Boundary State (triggered through a PhysX callback, the OBC algorithm is run). If colliding with a remote object's aura (triggered through a PhysX callback, the AC algorithm is run), it is immediately moved into the Sent State.

From the Intersecting Boundary state, an object remains in this state until one of the following occurs: the object is no longer intersecting the server boundary as is contained entirely within the host region (triggered through a PhysX callback, the OBE algorithm is run), in which case the object moves to the Host Region state; the object is no longer intersecting the server boundary and has fully traversed the boundary, in which case the object moves to the Boundary Traversed state; or the object collides with a remote object's aura (triggered through a PhysX callback, the AC algorithm is run), in which case it is immediately moved into the Sent State.

From the Boundary Traversed state, an object remains in this state until one of the following occurs: the object begins intersecting the host's boundary again (triggered through a PhysX trigger collision callback), in which case the object moves to the Intersecting Boundary state; the object collides with a remote object's aura (triggered through a PhysX callback, the AC algorithm is run), in which case it is immediately moved into the Sent state; or the object cluster (group of one or more objects with overlapping auras) becomes an island (no object in the cluster is intersecting the host's boundary) determined using the BT algorithm, in which case the object is moved to the sent state. The object cluster island check (BT algorithm) is performed every update, per boundary.

When an object enters the Sent state, the object data is added to the object migration buffer for the destination server and is immediately moved to the Destroyed state, in which it is removed from the host's simulation.

4.5 The Visualiser

The details of the visualiser, both the features and implementation details, will now discussed.

The visualiser is built using Unreal Engine and the RakNet plugin.

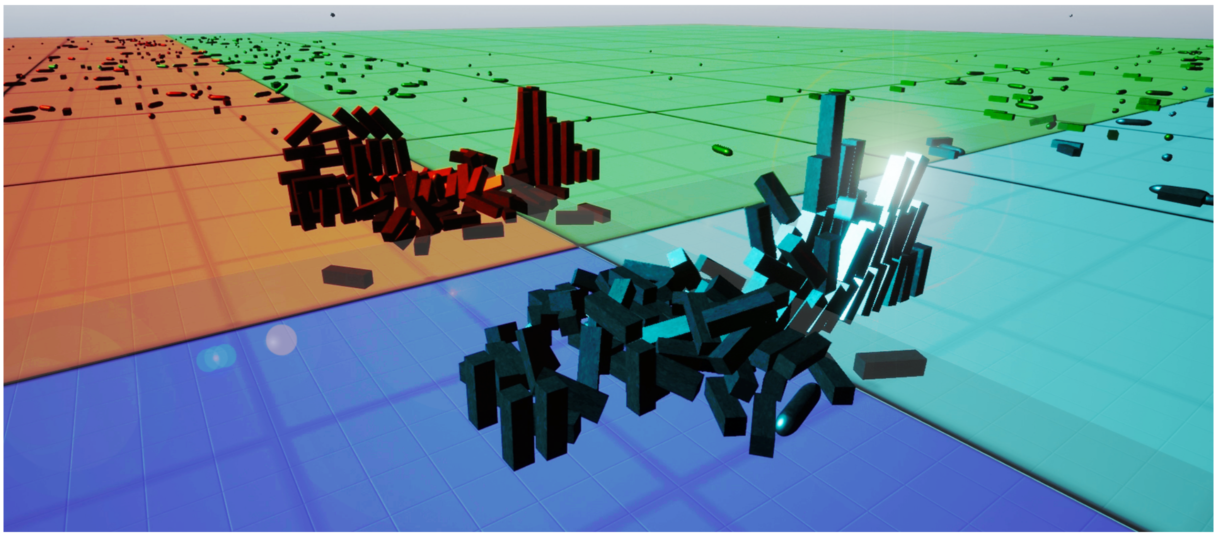


Fig. 4.13 Stacks of objects near and on a corner boundary. This demo scene consists of two large stacks of server created cuboid replicas near the corner of four server regions. Either side of each server boundary are random injection volumes, in which server created objects are injected into the simulation, randomly, within a volume. Each colour represents a different owning server.

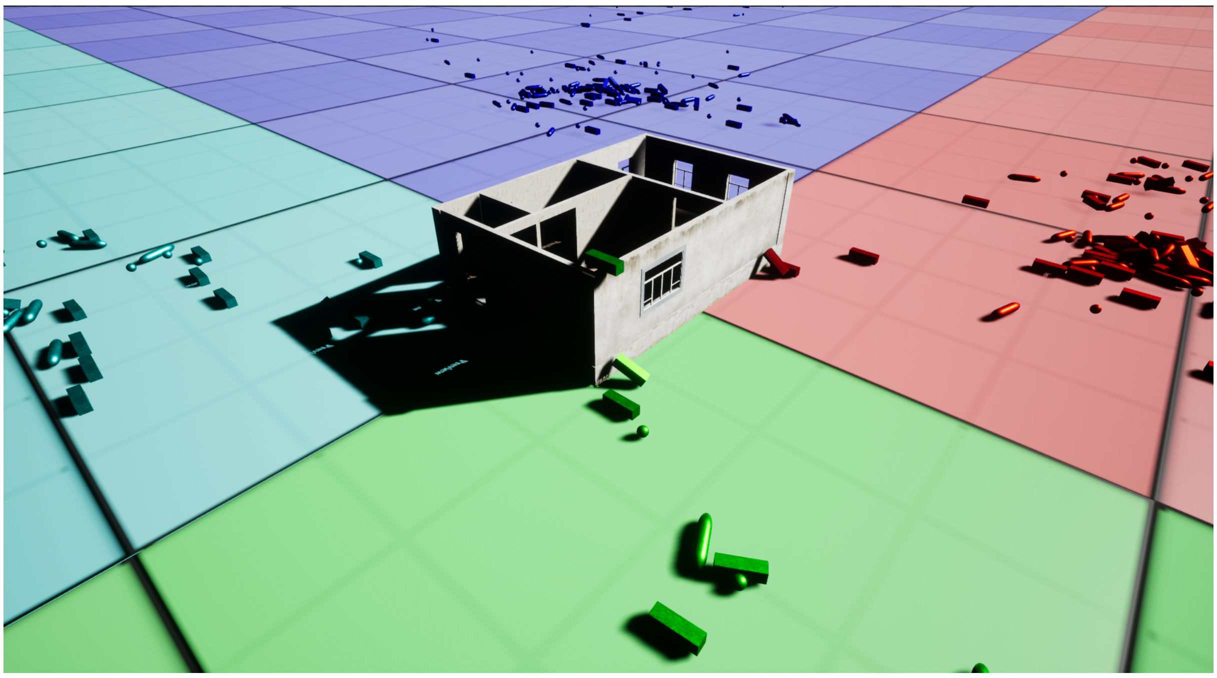


Fig. 4.14 A static object on a corner boundary surrounded by objects being simulated on different servers. This demo scene includes a client-created static object (a building), which is being simulated on all servers. Numerous server created objects surround the building, which have been triggered from within the client.

4.5.1 Terminology

Before discussing specific details of the visualiser, the terms used must be defined.

- The visualiser: The software application that is capable of connecting to AP servers and display replicas of objects on the servers.

- The editor: This refers to the Unreal Engine Editor. The version of the editor used for the visualiser is UE4.19.2
- Edit-time: When scenes are being edited in the Unreal Engine Editor, before the visualiser has been packaged.
- Run-time: When the visualiser is running
- Player: The in-visualiser representation of the user. Has a position and rotation. The camera for rendering is attached to the player
- Replica object: A representation displayed in the visualiser of an object being simulated on a server, which is intended to replicate the physical state of the object on the server.

4.5.2 Overview

The visualiser connects to multiple servers, objects are simulated on server and those objects from all servers are replicated on the visualiser. These are referred to as replica objects. The visualiser uses RakNet's ReplicaManager, which is responsible for: the creation of replicas objects on both the server and client; communicating the state of objects from the server to the client, in order for the replica objects to be replicated; and the deletion of replica objects. Replica objects are replicated on the visualiser by the server sending the state (position, rotation) to the client, the replica representation on the client is then updated with the newly received state from the server. The object state is optimised to reduce processing and network overhead, by only sending updates of the position and rotation to the client, when they change on the server. In the visualiser, replica objects are kinematic objects, i.e. they are not physically simulated in the visualiser, their physical states are updated entirely by messages received from the server, such as their movement from velocity and no interactions, such as collisions, are simulated in the visualiser.

Replica objects can be created in one of two ways. Either by the server or by the client. The implementation details of each type of replica creation will now be discussed.

4.5.3 Server-created replica objects

The visualiser supports server-created replica objects, which are limited to pre-defined basic objects (cuboid, sphere & capsule). A different render material per server is used to identify the owning server of each object. These server objects have pre-defined representations on the client. The server sends an identifier for the type of object, which is then created on the client. Using pre-defined objects means the whole geometry of the object does not need to be transferred to the client. This is useful for when there are a large number of identical objects as it avoids having to re-send the same geometry for each object. However, this method requires the client to know of these pre-defined types before packaging.

4.5.4 Client-created replica objects

The visualiser also supports arbitrary client-created objects. Entire scenes can be created in the client at edit-time, in the Unreal Engine Editor, and then re-created on servers when connections to the servers are made. This works for objects with any mesh, i.e. the server does not need to know about the geometry of an object when the server is built and deployed, the client communicates all data required to create the physical object on the server. All objects marked for replication on the servers are sent to all servers, if an object exists outside of a server's region, the creation is skipped, meaning the replica object is created only on the server containing the replica object. The central point of the object is used, avoiding conflicts between servers when an object exists that intersects a server region boundary.

4.5.5 Client-created static objects

In addition to replica objects, static objects can also be created by the client. As these objects are static, there is no need for their states to be replicated as they do not change over the course of the simulation. Static object work simply by the client sending the geometry, position and rotation details of the static object to all servers and the servers then re-create the static object. These static objects then remain unchanged throughout the simulation, requiring no updates to the state to be communicated to the client. Static objects are created on all servers. This could be optimised so only static objects within a server's region are created, however, AP can lead to clusters of objects being simulated outside of their host server's region, which may interact with static objects that do not exist on their host server. In addition, the cost of simulating static objects tends to be very low. They do not require any updates as they are not dynamic. In the future, region boundaries may change; a server may become responsible for a static object previously residing in another server's region, which would then have to be communicated when the region boundaries changed, by all servers simulating all static objects. This avoids the need for extra communication when server boundaries change. With these trade-offs in mind, the decision for AP was made to create all static objects on all servers, when static objects are used in a simulation.

4.5.6 Interactivity

The visualiser also allows for interactivity, the creation of server-created objects can be triggered from the client. This feature uses RakNet's Remote Procedure Calls (RPCs), which enables clients to execute procedures on the server, including any required parameters. The client sends a RPC (triggered by a key press) with the parameters of player position and forward direction. The RPC message is received by all servers, only the server which is responsible for the player position creates a server-created object at that position and the object is given a velocity in the direction of the player's forward direction. This allows the player to create new objects at run-time, which are then able to interact with all objects in the simulation.

The user is also able to reset or change scenes that are being simulated on the server. In the visualiser, the scene can be reset on a key press. This sends an RPC to all servers, which either resets all simulations or resets the current test scenario. As all objects are deleted on the servers, any replicas on the visualiser are deleted. Also available to the player is a menu which can be opened that allows the user to select a scene to switch to. This allows the user to change the scene in the visualiser at run-time while maintaining connections with all servers.

4.5.7 Client-defined server regions

Through the editor at edit-time, it is possible to define the regions that each server should be responsible for. Using Unreal Engine's box colliders, a user can specify the area of each server region. Upon establishing connections with all servers, the client then communicates the area for each region with the servers through RPCs. In addition to this, servers are able to automatically calculate their neighbouring regions and establish appropriate communications with them; linking boundaries with corresponding servers and determining which servers to broadcast auras to.

4.5.8 Future work

Currently, when server created objects are migrated between servers, the client destroys the replica and a new replica is created. This is because the simulated object is destroyed on one server and re-created on another. A more efficient way of dealing with this, would be a procedure for handing over server-ownership of replicas, removing the need for replicas to be destroyed and recreated. This would reduce processing overhead on the client, as there is no longer a need to destroy and recreate objects when server migrations take place. In addition, this would also prevent any chance of objects appearing to 'flicker' due to any delays between being destroyed and recreated.

The visualiser can create rigid-bodies, including those with multiple shapes, however, joints (constraints) are not yet supported. Joints have the added complexity of needing references to multiple objects. In order for these to be replicated on the server, these references would need to be kept and correctly refer to the corresponding objects after being transferred to the servers. This could be achieved through the joint replica containing unique references to the object replicas, which are identical on both client and server.

An alternative approach to client-created simulations/scene is PhysX's scene-serialisation. PhysX has support for serialisation of a physics scene. This could be used to transfer the whole physical state of a scene to servers. This includes relationships between objects, i.e, joints and objects with multiple shapes.

Currently all objects from all servers are replicated on the client with equal priority and frequency. Interest management could be employed for large complex scenes as the visualiser can only replicate and render a limited number of objects.

Mesh deformation is a technique sometimes used in video games, in which the geometric mesh representation of an object is deformed, often as a result of physical interaction. Mesh deformation is not currently supported in AP and would require significant network traffic, especially if done over a period of time (a series of many deformations), rather than in a single instance. Position and rotation can be replicated with small amounts of data, however, geometry data can be arbitrarily large. In mesh deformations 100s or even 1000s of vertex points can change, all of which would have to be serialised and transmitted over the network in order to be correctly replicated on the client.

The implementation of AP will be discussed in this section. The following work is based on the implementation of AP used in [12], however, with many newer additions. We include full details of the implementation of AP to give a complete and whole solution.

4.6 Implementation Summary

In this chapter the following points were discussed:

- The system architecture of AP, including how the servers are deployed in the cloud and how they communicate with a client.
- A breakdown of the components of each server.
- Topologies of server regions, including examples and the communication requirements between servers.
- Potential server deployments to be investigated in the future, such as the use of overseer nodes and dedicated render nodes for game streaming.
- A high-level description of the aura implementation including justifications for design decisions made.
- The algorithms used by the servers to implement AP.
- The life cycle of a simulated object as an FSM.
- The features and implementation details of the visualiser, including a description of its features and future work to be carried out on the visualiser.

Chapter 5

Experiments and Results

In this chapter, the design for the experiments for testing both the scalability and correctness of AP are presented and justified, the results of which, are analysed and evaluated. Scalability experiments are carried out with increasing numbers of servers in both column and corner topologies. Correctness experiments are carried out using two servers and two objects, one from each server, colliding with each other under different conditions and with different tolerances used for the aura calculation.

5.1 Scalability Experiments

This study aims to measure and demonstrate scalability of AP. When more servers are added, the timeliness of the simulation improves and more objects may be supported. The performance measure of interest is the maximum frame time of a server as this indicates if the simulation can be maintained when object numbers increase (keeping a low frame-time is the goal). Therefore, an injection rate is used that spawns moving objects into the simulation as time passes.

Experiments were performed on two layouts of servers: column layout and corner layout. The column layout experiment was performed using an increasing number of servers from 1 to 10. The regions were laid out in a column configuration as shown in Fig. 5.1. Objects are injected at a constant rate, both near and far from boundaries. Injected objects are randomly selected from the following types: sphere (radius: $0.3m$), cuboid ($0.3m \times 0.3m \times 1.0m$) and capsule (radius: $0.3m$, height: $2m$) and start with a random velocity from the uniform distribution of: $(-10{<}x{<}10, -10{<}y{<}0, -10{<}z{<}10)m{\cdot}s^{-1}$. 50% of objects are injected in a volume of $20m \times 20m \times 150m$ centred $12m$ away from a boundary and $15m$ above the ground plane. 50% of objects are injected in the centre of a server's region in a volume of $20m \times 20m \times 20m$, $15m$ above the ground plane.

The experiment is designed to test the performance of the system with a reasonably even distribution of objects between servers with a ratio of 1:1 between objects close to and far from server boundaries.

The experiments were run for 60 seconds with an injection rate of 160 objects per second. For all experiments, a speed tolerance of $32m{\cdot}s^{-1}$ was used, which was the maximum expected speed for any object (based on maximum injection height and velocity, and gravity). The

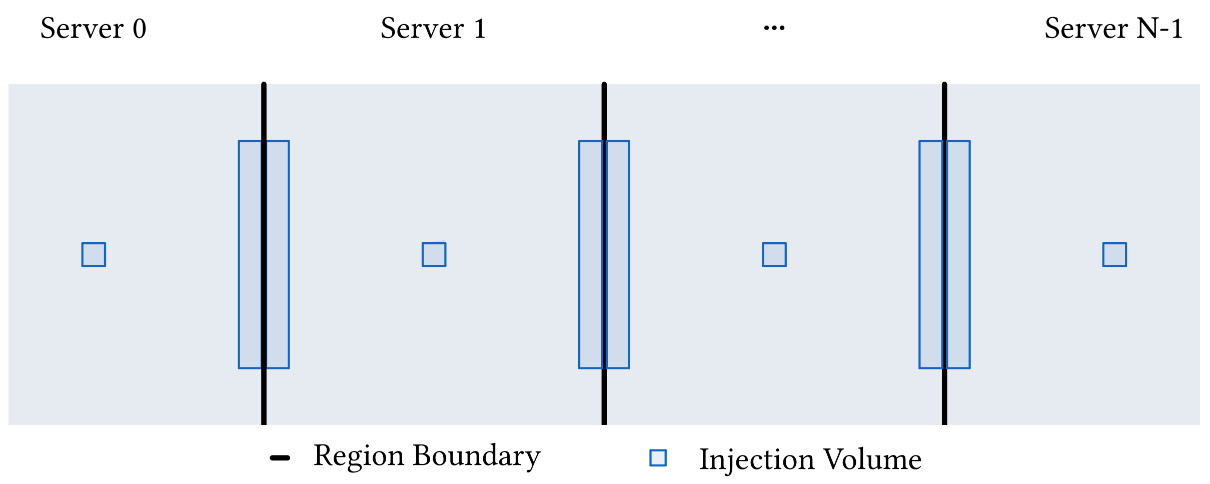


Fig. 5.1 Column Layout Experiment Setup

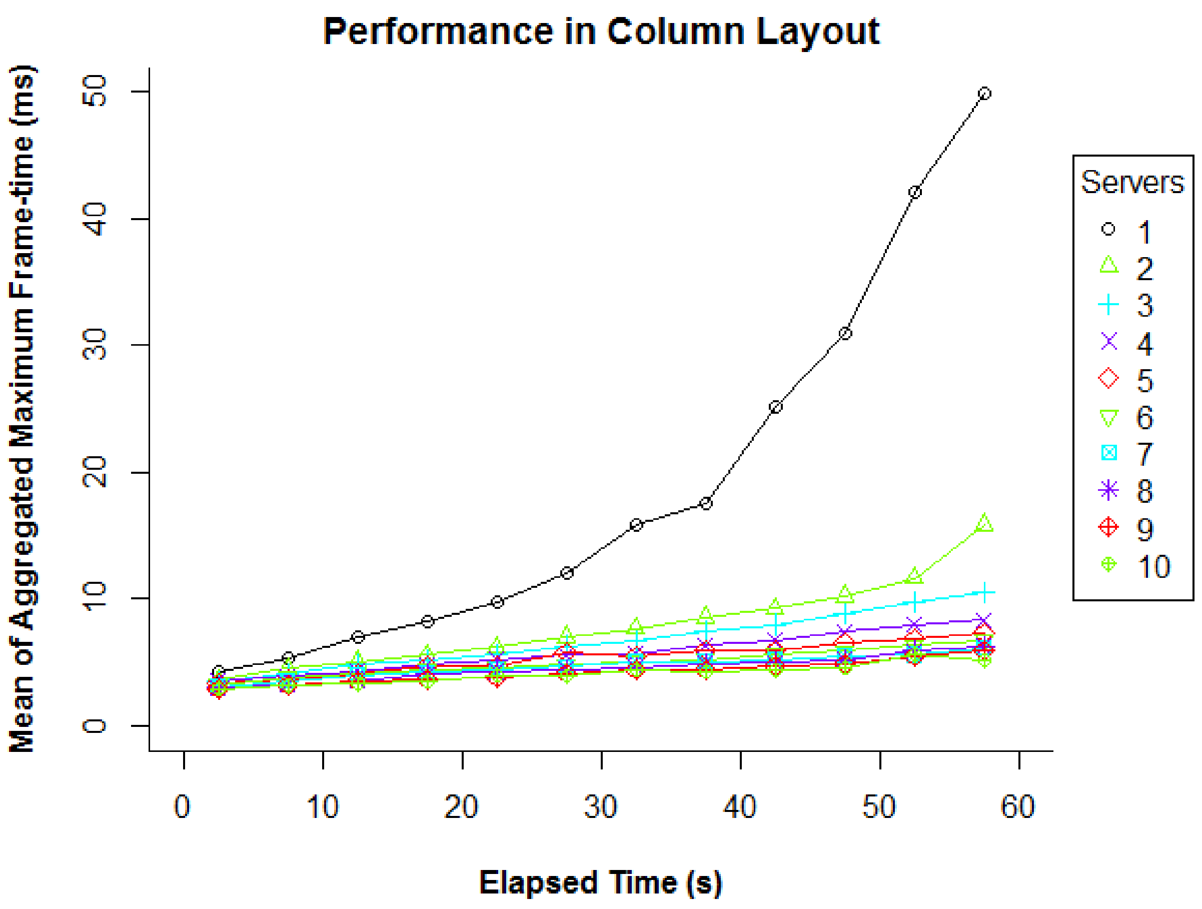


Fig. 5.2 Performance of increasing numbers of servers with an accumulating number of objects (in column layout)

latency tolerance for these experiments was set to 2*ms* (based off measurements of latency between servers) and frame-time tolerance was set to 15*ms* (based off preliminary performance results). A time of 16*ms* was used for the physics time-step. Each experiment was repeated 50 times at various times of day to account for differing performance in cloud resources at different times. For each iteration, the maximum frame time of any server was aggregated for each 5s period. The mean of the aggregated maximum frame times of the iterations was then calculated and plotted.

Experiments were conducted using AWS G2.2xlarge servers located within the same geographical region. A G2.2xlarge instance uses a 2.60GHz Intel Xeon E5-2670 CPU with 16GB RAM and an NVIDIA GRID K520 (Kepler) GPU running Ubuntu 18.04 LTS.

Fig. 5.2 shows the performance of our system with rising server numbers from 1 to 10. The graph clearly shows that the addition of servers lowers the frame time throughout the experiment. This is increasingly noticeable later in the simulation when a greater number of objects are present. For higher server numbers the reduction in maximum frame time is less than for lower numbers, which is as expected as in the ideal case the workload per server would be $1/n$ of the total workload, where n is the number of servers.

From these observations it can be concluded that this system is scalable in column configuration as the addition of servers results in increased performance.

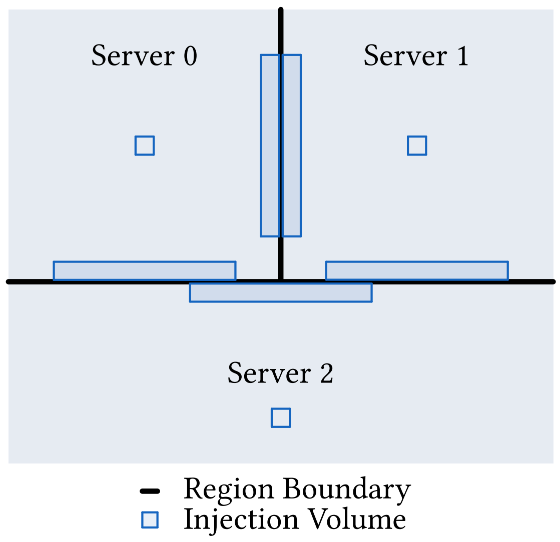
Experiments were also carried out using servers in a corner layout. The layouts of 3 and 4 servers are shown in Fig. 5.3, in which the 4 server case is a 2x2 grid with a single corner intersection in the centre. The 9 server case is a 3x3 grid with 4 corner intersections. Injection rates and volume dimensions remain the same as in the column-based experiments. Fig. 5.5 shows the graph illustrating performance.

As discussed in 4.1.1, topologies with corner boundaries between server regions have higher communication requirements. This is because servers must communicate with all neighbouring servers. The highest number of expected neighbours for a given server in a grid layout is 8 servers, which applies to the server in the centre of the grid illustrated in Fig. 4.4 and any server not on the edge of the grid. The graph in Fig. 5.5 demonstrates that increasing the number of servers lowers the frame-time. This remains true even in the presence of corner boundaries and with the maximum number of expected neighbours, 8, in a grid layout (meaning maximum communication requirement). The additional processing overhead of the higher communication requirements (as messages must be exchanged with all neighbouring servers) on the server with 8 neighbours is outweighed by the performance gains of partitioning the simulation using additional servers. In other words, 9 servers in grid layout outperform (have a lower frame-time than) a single server, despite the server in the centre of the grid needing to communicate with the maximum number of neighbouring servers for a grid layout.

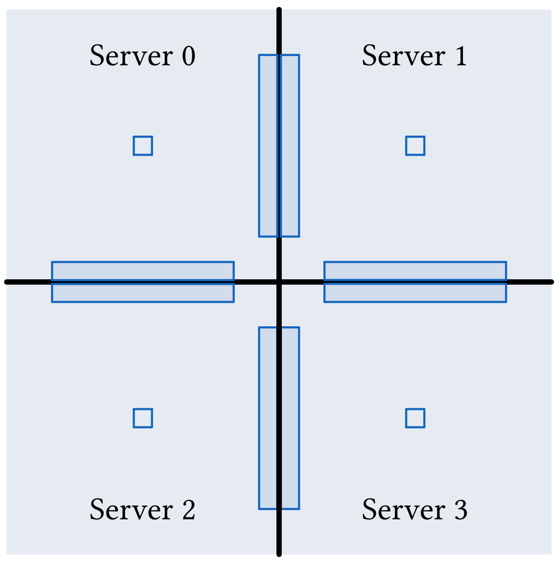
The performance cost of the higher communication requirements for corner layouts is reflected in the performance graphs as the respective number of servers in column layout outperform those in corner layout, i.e. 3 servers in column layout have a lower frame-time than 3 servers in corner layout.

As 8 neighbours is the maximum number of neighbours a server can have in a grid layout (and therefore has the most processing overhead), it can be concluded that any number of servers in a grid layout will outperform a single server (assuming reasonably even distribution of objects between servers and reasonable ratio of objects near and far from server boundaries).

From these observations it may be declared that this system is scalable in corner and grid layout configurations as the addition of servers results in increased performance.



(a) 3 Server Corner Layout



(b) 4 Server Corner Layout

Fig. 5.3 Corner Layouts

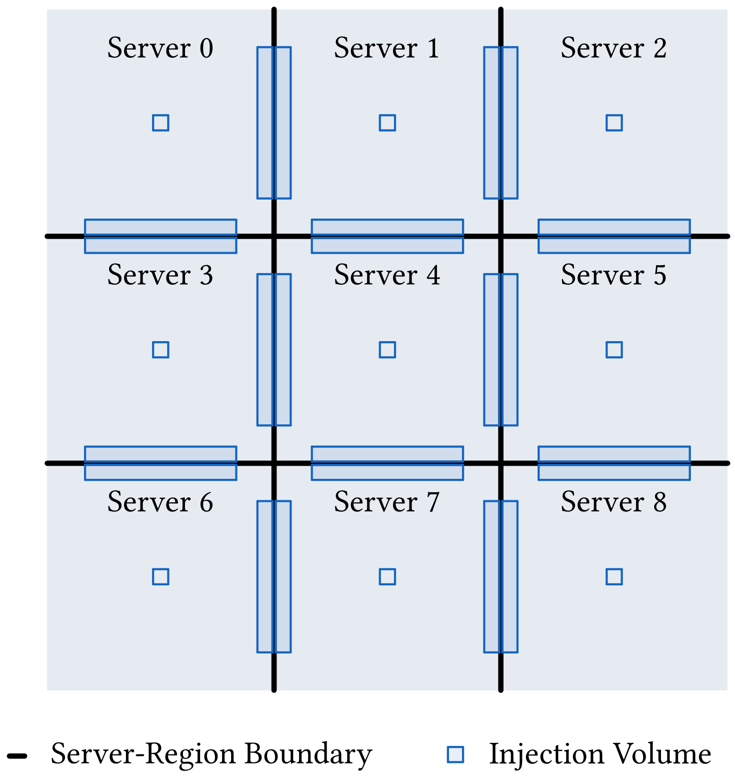


Fig. 5.4 9 Server Corner Layout

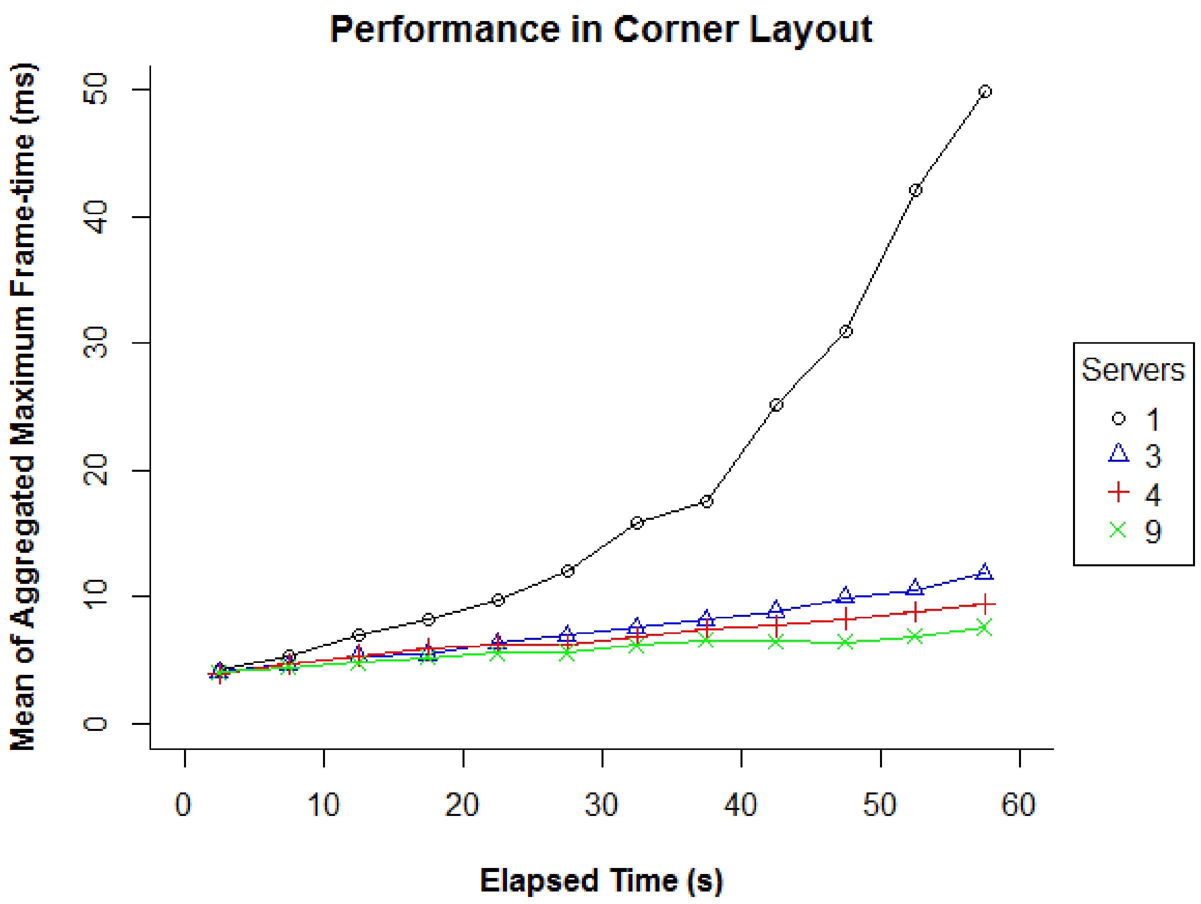


Fig. 5.5 Performance of increasing numbers of servers with an accumulating number of objects (in corner layout)

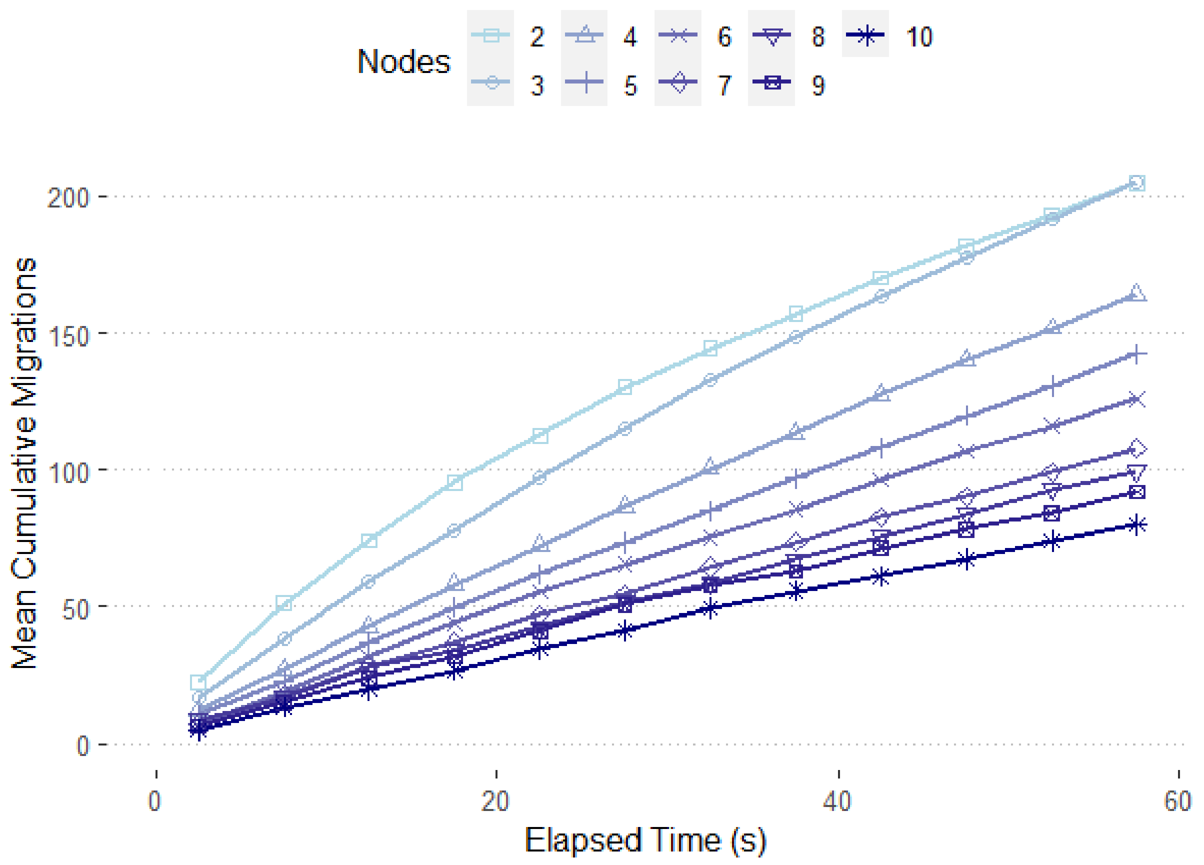


Fig. 5.6 Object Migrations over time (in column layout)

5.1.1 Object Migrations

In this sub-section we show the number of migrations of objects between servers. This is done using the average cumulative migrations between servers over time for the different numbers of servers. For each iteration of the experiment, the maximum number of cumulative migrations across all nodes for each time interval is taken and the mean of this value for each time interval is then taken across the 50 repetitions. This was done for both the column (see Fig. 5.6) and corner experiments (see Fig. 5.7).

In the experiments, objects are injected at a fixed rate across all servers into random locations within regions near and far from the boundary. As the number of servers increases, the number of objects injected per server decreases. As a result of this, experiments involving more servers will have a lower number and density of objects leading to fewer object migrations.

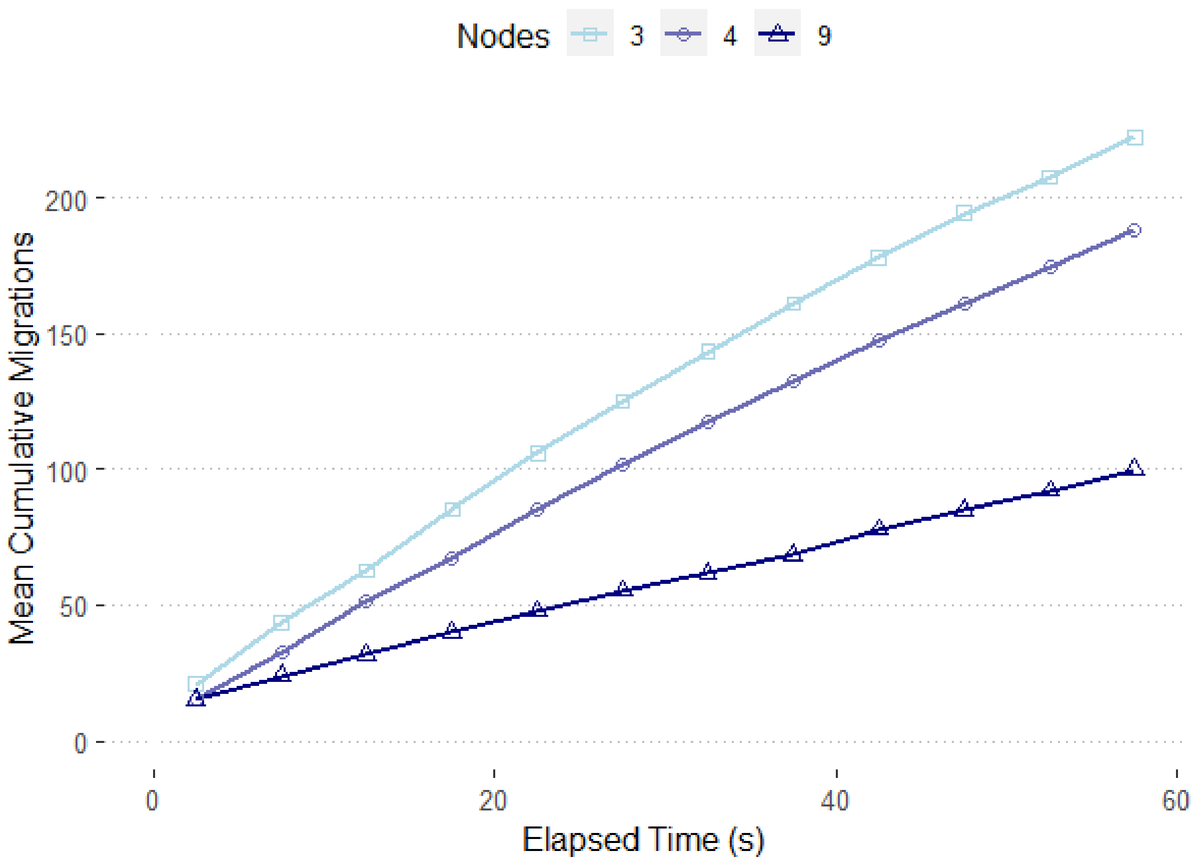


Fig. 5.7 Object Migrations over time (in corner layout)

5.1.2 Object Updates

In this sub-section we show the number of object updates sent between servers. An update is sent from the host server to any neighbours with interest in that object (i.e. when an object is within the aura distance of a boundary), whenever the object's position changes. The number of updates shared between servers will be shown in graphs produced using the following process: for each time interval of 5 seconds, the maximum number of updates is taken across all servers for each iteration of the experiment. The mean of these maximums is then taken for each number of servers and plotted over elapsed time of the experiment. This was done for both the column (see Fig. 5.8) and corner experiments (see Fig. 5.9).

The graphs demonstrate that as elapsed time increases, so does the number of object updates being sent in both the column and corner experiments. However, the relationship is not a linear one. The density of objects around the boundary will affect the number of updates being sent in two ways: (1) Updates are not sent for objects at rest; a higher density of objects means a newly injected object is more likely to collide with an object at rest, resulting in updates needing to be sent for both objects (2) A higher density of objects means a newly injected object is more likely to collide with another object and come to rest sooner than if no object were there.

Comparing the column and corner experiments, the corner experiments have a lower object density in the boundary region as objects are spawned in multiple boundary regions on all servers. As a result of this, the corner experiments result in fewer object updates for all numbers of servers (3, 4 and 9) compared to the same number of nodes in column layout.

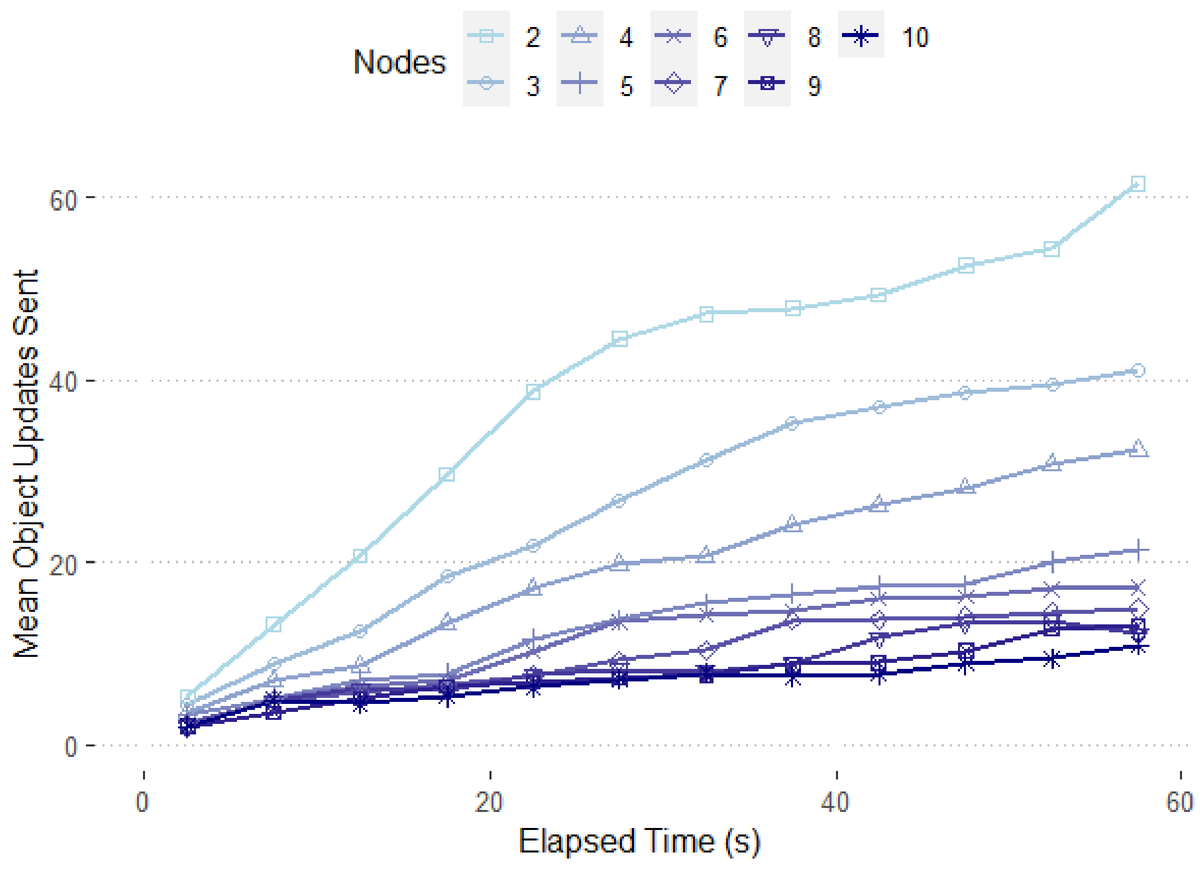


Fig. 5.8 Object Updates Over Time (in column layout)

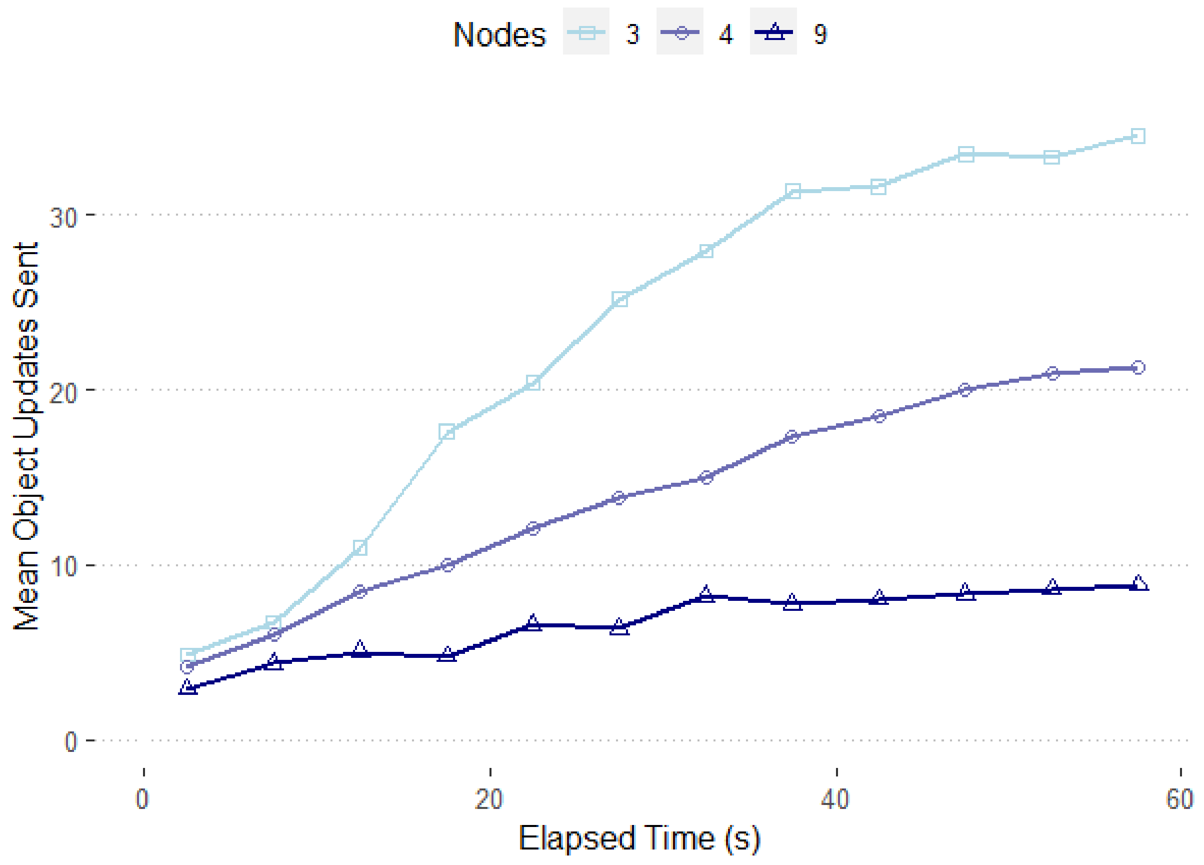


Fig. 5.9 Object Updates Over Time (in corner layout)

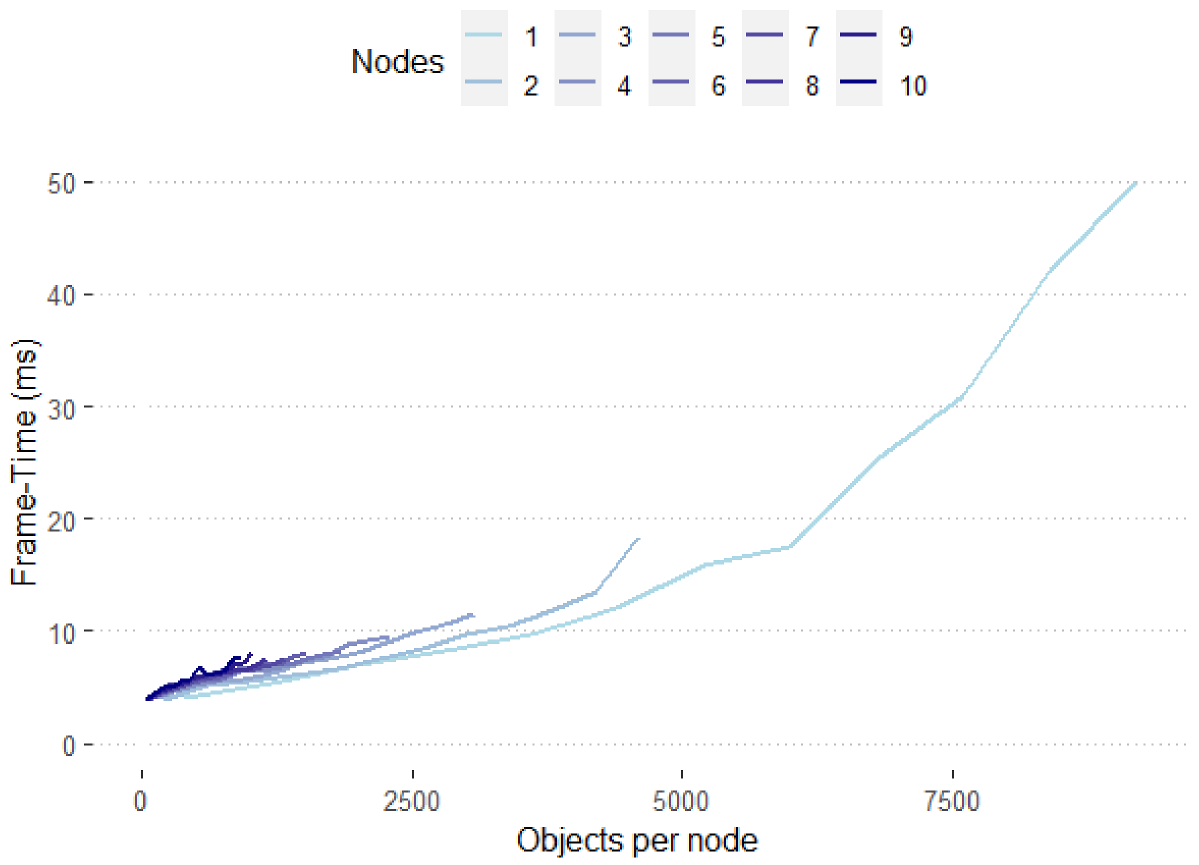


Fig. 5.10 Performance Cost of Increasing Objects per servers (in Column Layout)

5.1.3 Performance Overhead

In this sub-section we address the performance overhead of Aura Projection. This overhead is calculated by first measuring the performance cost (frame-time) in terms of objects per server (the number of objects is known because the injection rate of objects is known). This is shown in Fig. 5.10. A curve can then be fitted to the performance cost of the single server (centralised) experiment with increasing numbers of objects. This provides a baseline cost for non-boundary objects.

The relationship between objects per node and frame-time is approximately exponential and can be seen more clearly when plotting the measurements of frame-time on a log_{10} scale (shown in Fig. 5.11). The curve is fitted using linear regression on the log_{10} of the frame-time (shown in Fig. 5.12).

The cost of non-boundary objects can then be subtracted from the performance cost of each number of servers using the baseline from the fitted curve for the centralised experiment (the ratio of boundary to non-boundary objects is 1:1, so the number of boundary and non-boundary objects is known) leaving just the cost of boundary objects. To calculate the overhead of boundary objects, the cost of the same number of non-boundary objects is subtracted, leaving just the overhead of boundary objects relative to the cost of non-boundary objects (shown in Fig. 5.13).

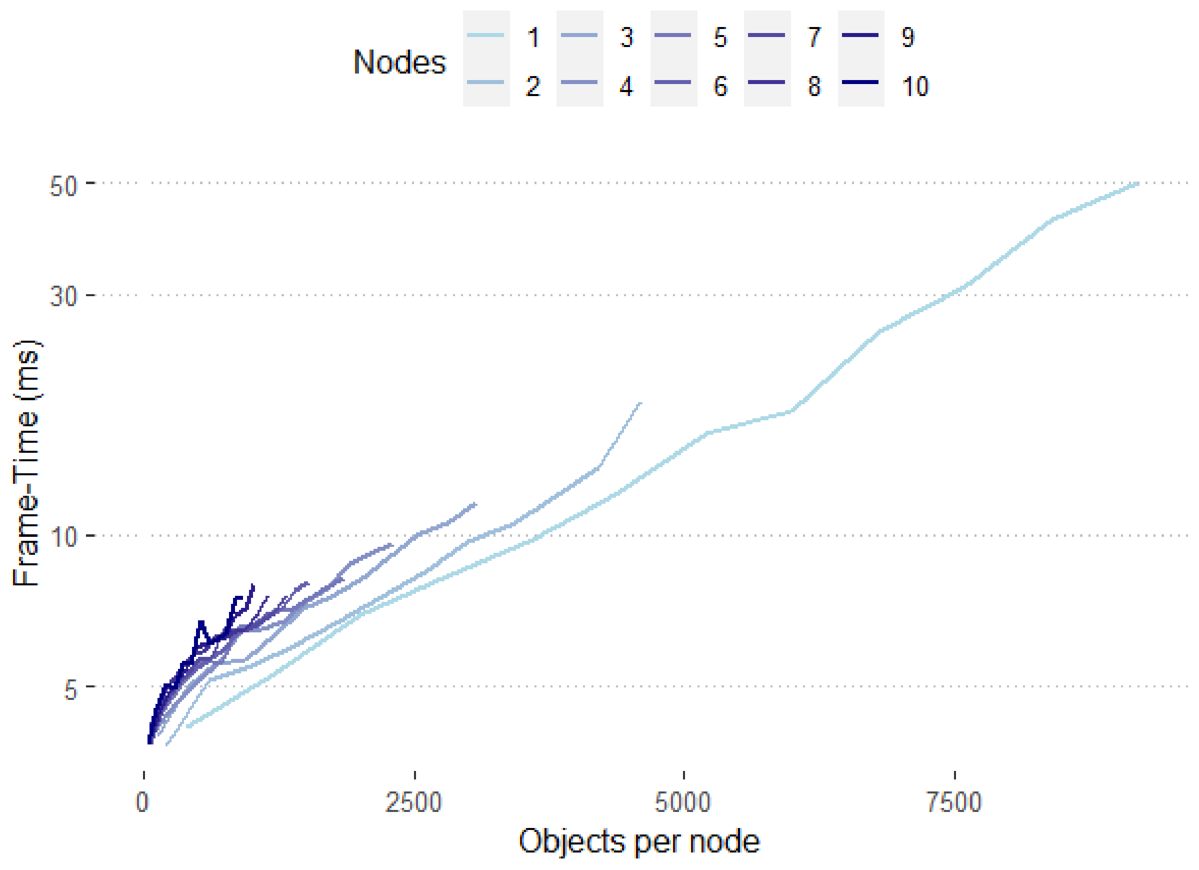


Fig. 5.11 Performance Cost (log_{10}) of Increasing Objects per servers (in Column Layout)

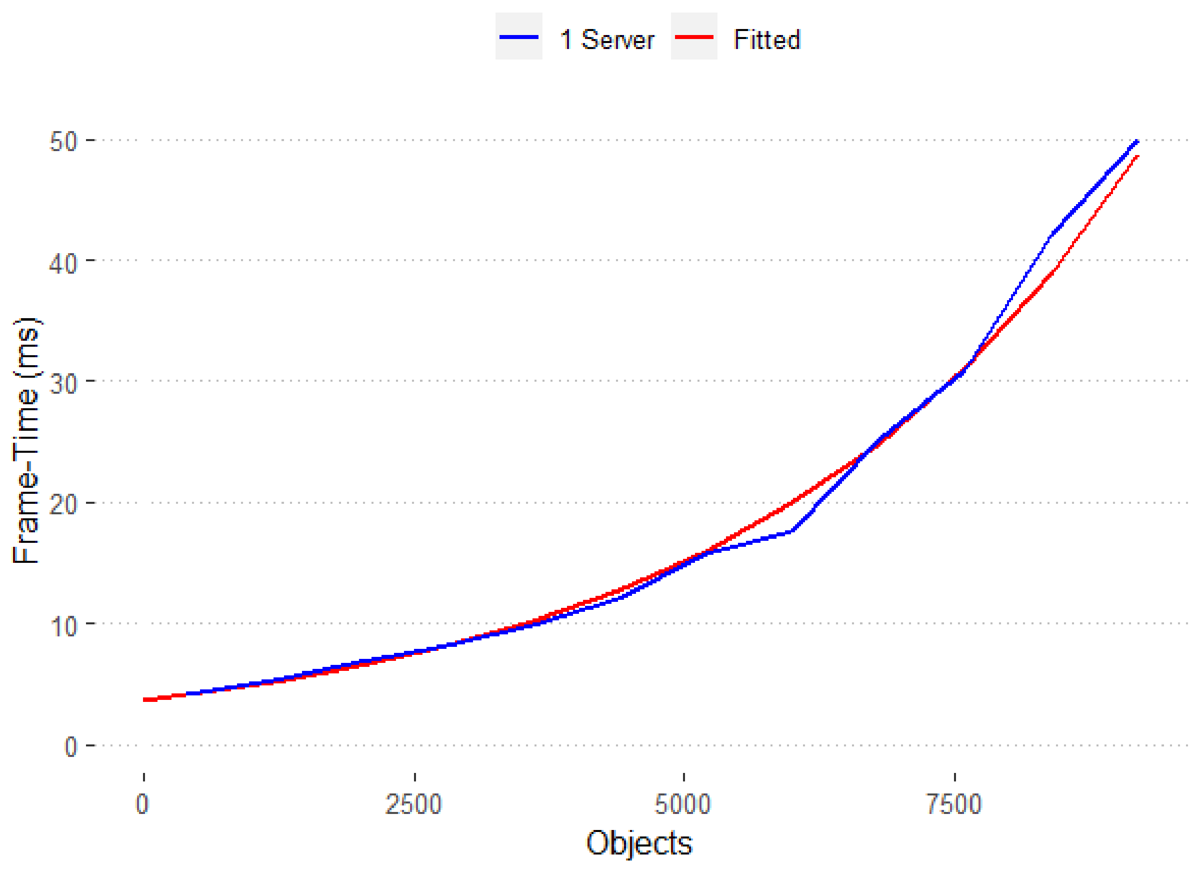


Fig. 5.12 Fitted Cost of Increasing Objects on a Single Server

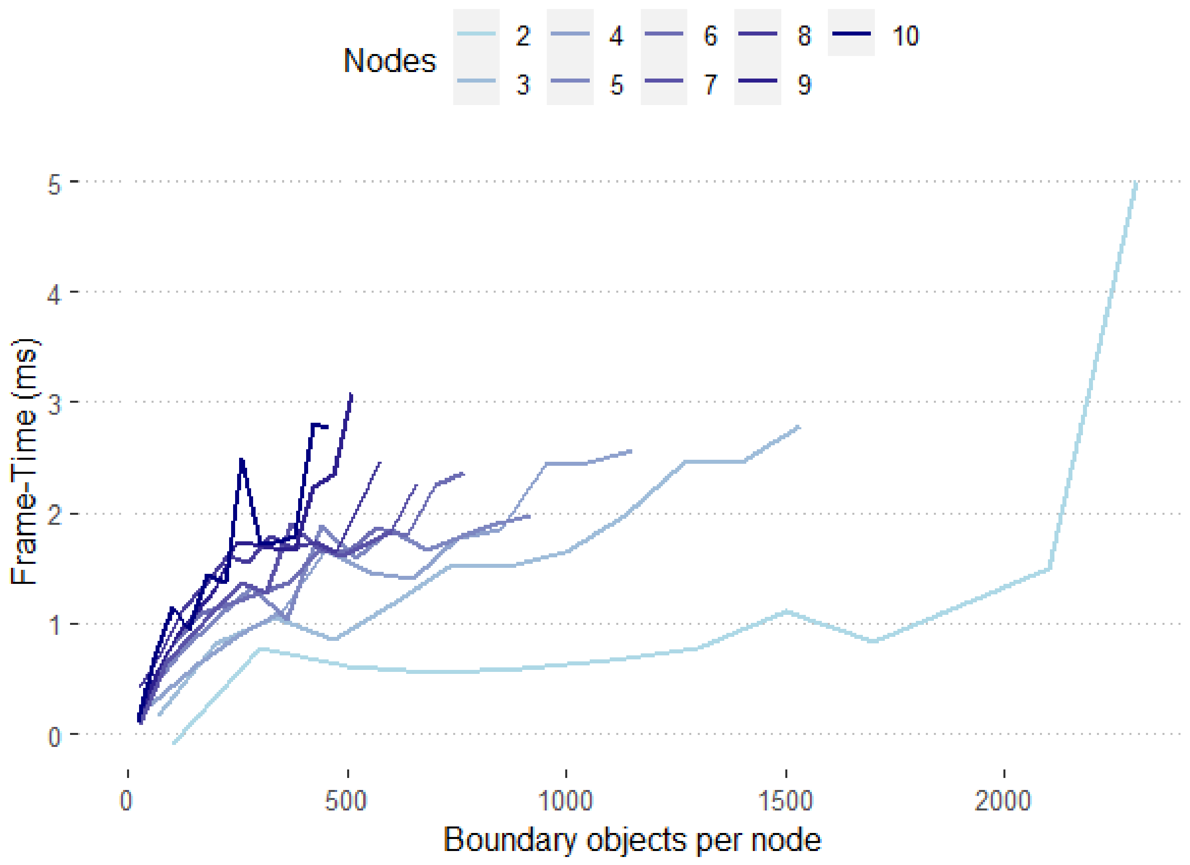


Fig. 5.13 Overhead Cost of Increasing Objects per Node in Column Layout

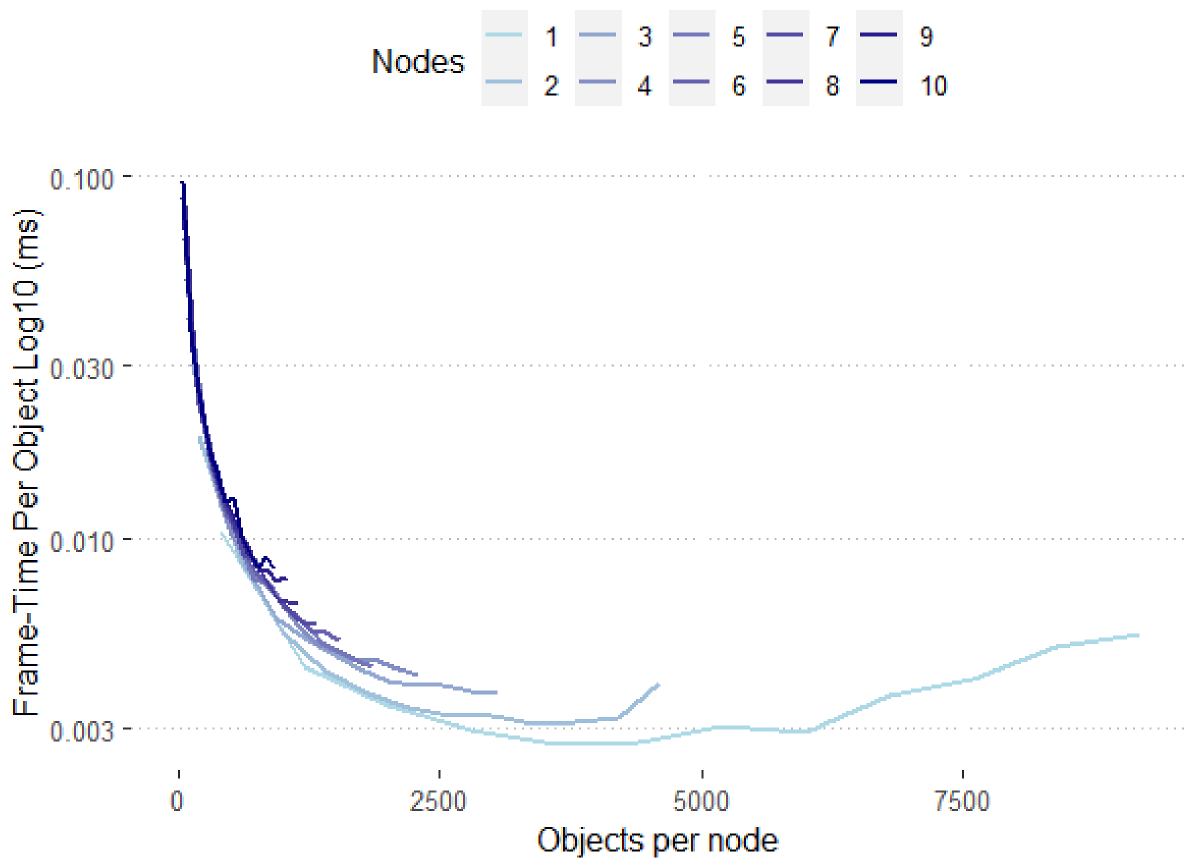


Fig. 5.14 Cost per Object per Node (in column layout)

5.1.4 Optimal Scale

The relationship between cost per object and increasing objects is shown in Fig. 5.14. The cost per object steeply decreases until the lowest cost per object occurs at approximately 4000 objects. Above 4000 objects, the cost per object begins to increase again.

This means, for example, if 1000 objects took 6ms to perform a single physics time-step, 500 objects would take more than half of 6ms. This has important consequences when distributing the workload as it means splitting objects between two servers may not double the performance even without accounting for overhead. Likewise, splitting the objects between two servers could also potentially lead to more than double the performance. For example, if 8000 objects exist on one server and the simulation takes 80ms to perform a physics time-step, then if the objects are split across two servers, resulting in 4000 objects on each, it would take approximately 12ms to perform a physics time-step. This would be a 6.67 factor increase in performance (not accounting for overhead).

The relationship between number of objects and frame-time is approximately exponential (shown in Fig. 5.15). In order to determine the optimal scale for this experiment, we can calculate the coefficients for approximate exponential functions for the performance of each number of servers. Ideally, a double in the number of servers should lead to a double in performance and therefore double the exponential coefficient. First, approximate functions were fitted for each node using linear regression on the log_{10} of the frame-time (shown in Fig. 5.16). This gives a coefficient for each node. This coefficient can then be used to work

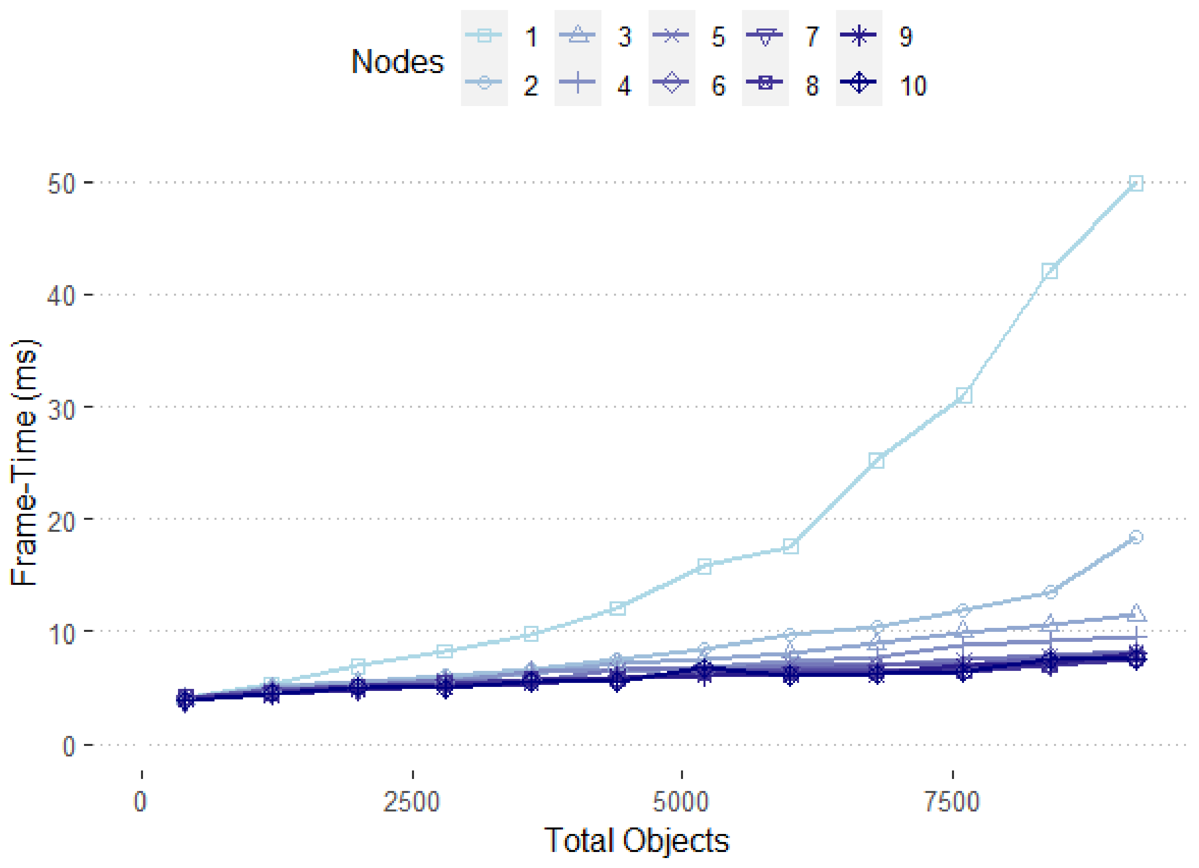


Fig. 5.15 Cost of Increasing Objects (in column layout)

out relative performance. 2 servers led to an increase in performance by a factor of 1.74, 4 servers a factor of 3.40 and 8 servers a factor of 4.22. It should be noted that the accuracy of the coefficients for higher numbers of servers (i.e. 8, & 10) will be low as the relationship appears linear when there are low numbers of objects per server. As a result of this, it is not possible to accurately predict the performance of the system for more than 8 servers with more than 10,000 objects (the total used in this experiment).

The coefficients used in the approximate functions can then be plotted and compared to the ideal line, show in Fig. 5.17. A curve was then fitted using a self-starting asymptotic regression model to the curve of coefficients, to approximate the point at which the performance factor increases converge. The result of this is 4.26.

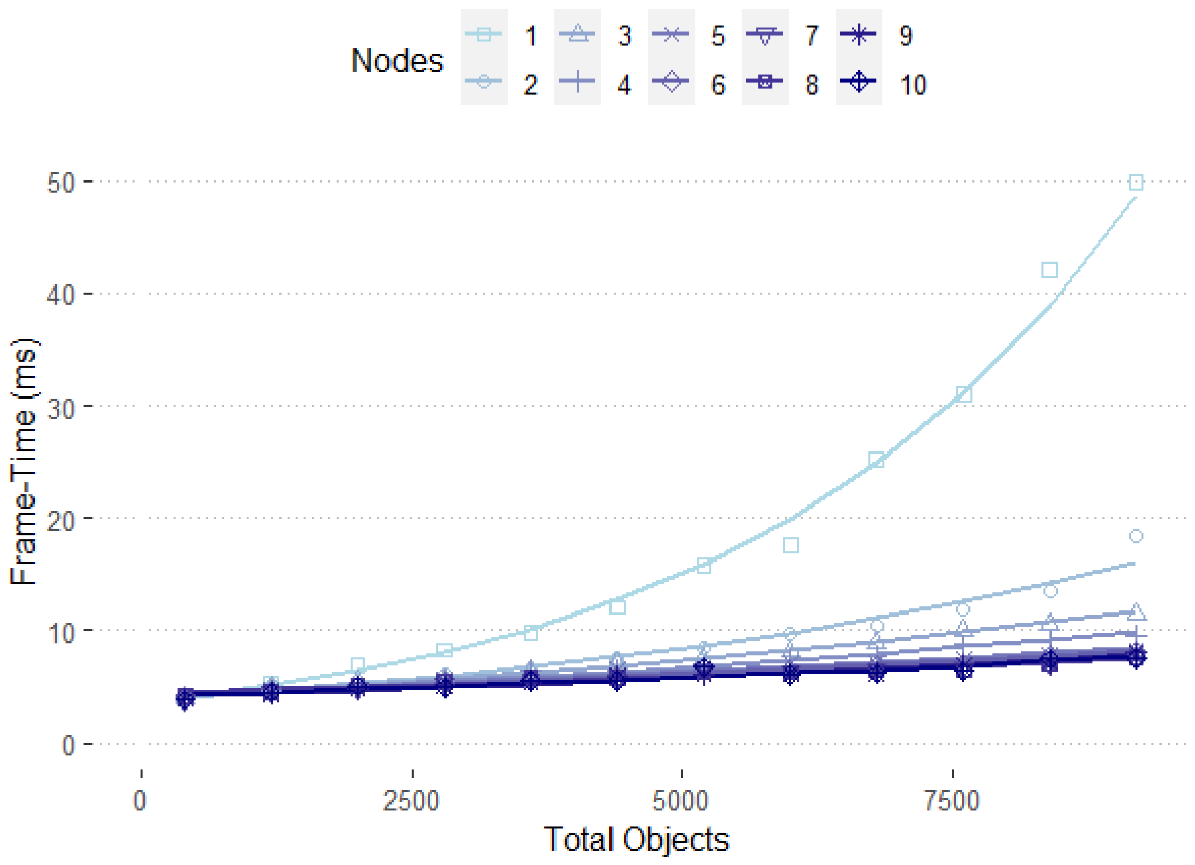


Fig. 5.16 Fitted Cost of Increasing Objects (in column layout). The shapes indicate the results from the experiment, the curves indicate the fitted functions

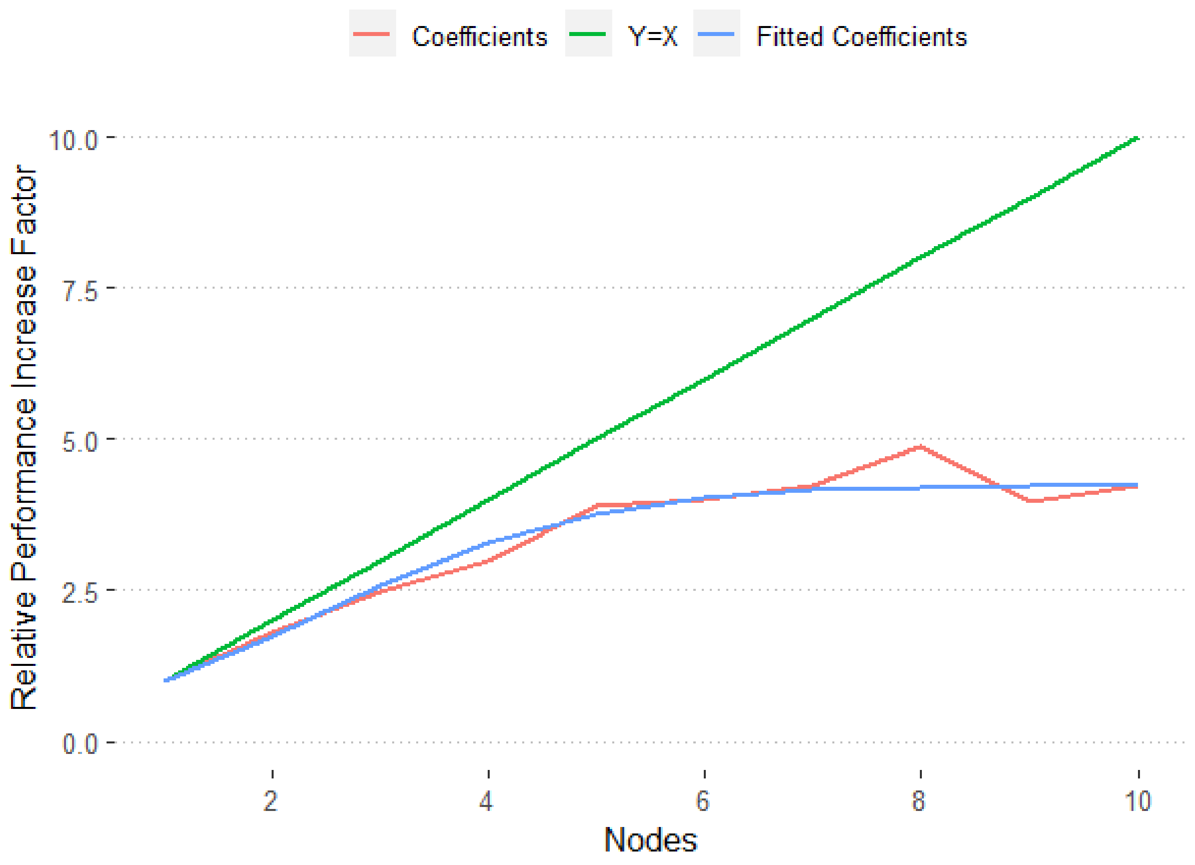


Fig. 5.17 Fitted Cost of Increasing Objects (in column layout). Coefficients used in Fig. 5.15 are in red, the ideal line of Y=X (i.e. doubling servers doubles performance) is in green and a curve fitted on the values of the coefficients is in blue.

5.2 Collision Correctness Experiments

In the following section the correctness of collisions between objects is explored, i.e. do objects have the same collision behaviour in a AP as they do in a centralised system.

The following points highlight the contributions of the correctness experiments:

- Demonstrate that collisions between objects and migrating are handled correctly as long as the factors (speed, frame-time, latency) remain within their tolerances. Collisions can be handled correctly under worsening conditions (higher object speed, frame-time and latency), so long as the tolerances account for these.

- Demonstrate that late collisions begin to occur once tolerances are exceeded or as predicted by the aura calculation, i.e. auras are the size they need to be, and no larger, which would impact on performance.

- Demonstrate the severity of each of the factors being exceeded, i.e. when different factors are exceeded, how badly are collisions handled?

- Measure the frequency of two-object-thrashing

Experiments were undertaken to detect erroneous collisions under different conditions and are concerned with objects that are migrating and/or interacting with an object that has recently migrated. Erroneous collisions fall into two categories:

- Missed collisions - collisions that should have taken place but were not detected by the physics engine e.g. Bullet through paper problem.

- Late collisions - collisions that were detected later than expected in a centralised simulation and result in not only different collision results, but also unstable collision response.

A recap of how a naive system can lead to erroneous collision behaviour and how AP can lead to erroneous collision behaviour if speed, frame-time, or latency exceed the user-defined tolerances will now be given.

In a hypothetical naive system, a system that does not use AP, objects are migrated solely based on their position and objects on a remote server are not accounted for. Two objects close to the server region boundary could be occupying the same space at the same time. If one object were to be migrated to the other server, for example, if the centre of the object has traversed the region boundary, then the object would be migrated into an overlapping space as the other object on the receiving server. This penetration could be much greater than would be expected in a normal collision detection and could lead to a highly erroneous collision response.

AP works by predicting the future bounding sphere of both the object projecting the aura and a potential object on the remote server. When an object collides with an aura from a remote server, the object is migrated to the owner of the aura. However, the aura projection is not instantaneous and has to account for the following three factors: the speeds of both objects; the frame-times of the two servers; and the latency between the two servers. These three factors need to be known through the duration between the aura projection being initiated and a remote object being received and collided with. As it not possible to know these three factors in advance, tolerances must be used. These three tolerances are used to determine the size of the aura with the aim to ensure a remote object is migrated before a collision happens. If the speed tolerance is exceeded, an object can penetrate deeper into the aura than the maximum penetration to ensure it is migrated in time before a collision occurs. In the case of latency, the aura would potentially arrive too late on the receiving server, meaning the migration is triggered too late. In addition to this, the migrating object's arrival will be delayed, potentially leading to a late collision. In the case of frame-time, the extra delay between the messages being processed and physics time-step could cause the collision with the aura to be detected too late and potentially lead to a late collision. Furthermore, the extra delay between the migration message being received and the object being present in the physics simulation on the receiving server could cause the collision to be detected too late.

To understand how late collisions lead to unstable collision response, it must first be understood how in real-time collision detection works, which is discussed in 2.1.4. In summary, in real-time physics engines, objects move in discrete steps. As a result of this, when two objects collide, they overlap each other i.e. penetrate. The collision is resolved by calculating the forces resulting from the collision and moving the objects so they no longer overlap (in PhysX if penetration is high, the latter may be done over several physics time-steps). If the penetration is high, such as in the case of a late collision response or if two objects are moving at high speed towards one another, the collision response can no longer guarantee stable results.

5.2.1 Collision Error Detection

It is possible to detect late collisions between objects; Given the relative speed of the two objects and the physics time-step, it is possible to calculate the maximum expected penetration distance. If a collision is detected and it is above this value, it means it is a late collision. In addition, if no collisions occur, it means there is a missed collision.

Collision penetration can range from 0 to maximum penetration distance and depends upon the distance between the two objects' surfaces in the time-step before the collision. Maximum penetration for a given speed occurs when two objects are travelling directly towards each other and in the time-step before the collision occurs, there is no distance between the surfaces of the objects. In the next time-step, this results in the two objects overlapping each other with the maximum penetration for their relative speed. The overlap is then detected as a collision by the physics engine. Re-arranging $s = d/t$, to $d = s.t$ and substituting the physics time-step for t, gives us the maximum distance (penetration) for a relative speed, and we are able to plot the maximum expected penetration line.

In all the following experiments collisions were recorded, for each collision, the following data is contained: Relative Speed, the speed of the two colliding objects relative to each other; Penetration, the distance the two objects overlapped when the collision was detected. PhysX reports penetration of collisions as negative values and can report collisions before the two objects are intersecting, which is recorded as a positive penetration. In all experiments, the penetration is negated for simplicity. The relative speed and penetration can be used to

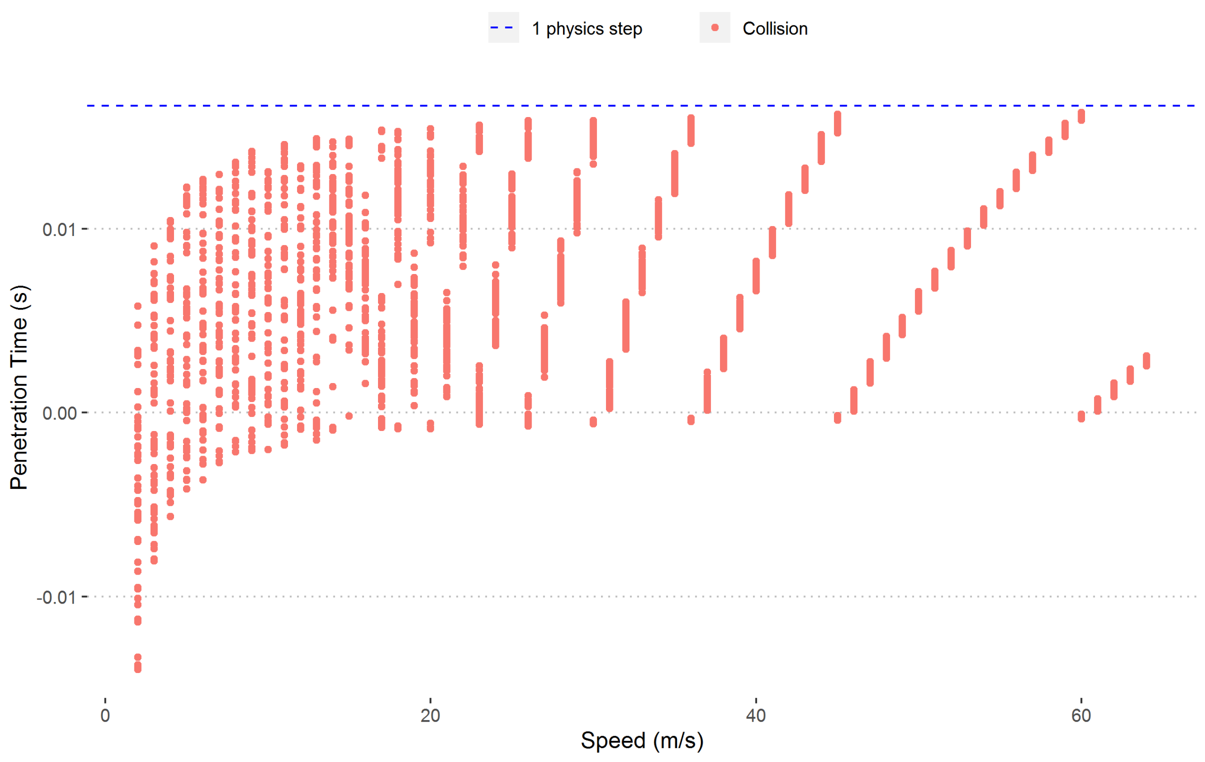


Fig. 5.18 Penetration time of collisions with varying speed. The maximum expected penetration time of 1 physics step is marked with a dashed blue line.

determine if the collision was a late collision. In all the following experimental scenarios only two objects are used, meaning if there was a missed collision there would be no collision output and multiple collisions from the same pair of objects can be ignored. The two objects start the scenario with a specified velocity (drag and gravity are disabled), the direction being directly towards the other object and the two objects were given the same speed and a random variation of between 0 and $-1m{\cdot}s^{-1}$ to prevent identical results between iterations. Spheres were used, so rotational effects, such as rotational velocity, do not need to be accounted for.

Thrashing between two objects on different servers is an unsolved problem in AP; thrashing occurrences can be detected by counting the number of migrations that occur before the first collision is detected, as only two objects are involved in each experiment, the number of migrations before a collision should only be one, otherwise thrashing has occurred. For the purposes of these experiments, collisions involving thrashing will be excluded, regardless of the correctness of the collision. For the remainder of this chapter, thrashing refers to only thrashing between two objects and not the solved 3-object-thrashing problem.

5.2.2 Control Experiment

A control experiment on a centralised system (single server, single simulation) was performed, in which the scenario was repeated multiple times. The speed of each object is increased from 1 to $64m{\cdot}s^{-1}$ in steps of $1m{\cdot}s^{-1}$. The experiment was repeated 50 times, for a total of 3150 collisions.

As speed increases, expected penetration increases. Using penetration time gives us a uniform maximum expected value regardless of speed. Re-arranging $s = d/t$, to $t = d/s$ and substituting the collision penetration for d and speed for s, gives us the penetration time for a collision. It should be noted that because physics engines work in discrete time steps, this time value does not actually reflect the real time the colliding objects were penetrating each other. It should also be noted that PhysX can report collisions before objects intersect, leading to the negative penetration distances shown. Correct collisions result in penetration times of up to 1 physics time step and any collision penetration times over this value are considered erroneous. Fig. 5.18 shows the penetration of objects with increasing speed in a centralised simulation (displaying speed without the random variation) and demonstrates that penetration time is never greater than the time of one physics step, regardless of the relative speed of objects. The purpose of this experiment was to confirm the predicted maximum expected penetration in a centralised simulation. This provides a benchmark of correctness to compare with results from experiments using multiple servers.

Using the above method to detect late collisions, experiments on a distributed configuration of a simulation were then performed. Experiments were carried out for both variations in all three aura tolerance factors and packet-loss. Experiments were conducted using AWS G2.2xlarge servers located within the same geographical region. A G2.2xlarge instance uses a 2.60GHz Intel Xeon E5-2670 CPU with 16GB RAM and an NVIDIA GRID K520 (Kepler) GPU running Ubuntu 18.04 LTS.

5.2.3 Experiment- Varying Factors

Experiments were carried out to demonstrate the effects of varying each aura calculation factor on the correctness of collisions between objects. The purpose of these experiments is to demonstrate that collisions in AP are always correct if the three factors that determine aura size remain within respective tolerance, i.e. there are no late or missed collisions. In addition, if the aura calculation is correct, we expect to see late collisions begin to occur as soon as tolerances are exceeded. If errors do not occur as soon as the tolerances are exceeded, this is evidence that the aura radius is too large. Large auras can lead to reduced performance in AP, therefore auras that are only as large as needed to be, for the given tolerance, are desirable to maximise performance.

In order to rigorously test the hypothesis "Can real-time physics simulations remain correct when scaled?", a ‘worst-case’ scenario was used for the experiments, in which two objects are moving directly towards each other and collide while one of the objects is intersecting the boundary between servers. This experiment setup is illustrated in Fig. 5.19. In this experiment two servers were used with one sphere being created on each server. The two spheres were given starting positions so the two would collide at a point when one of the spheres is intersecting the server region boundary, thus creating the most likely situation for spheres collide with each other in the first time-step after being migrated; this is necessary as AP can only lead to late or missed collisions when at least one sphere involved in the collision has just migrated (received since the last physics time-step). This is because non-migrating spheres behave the

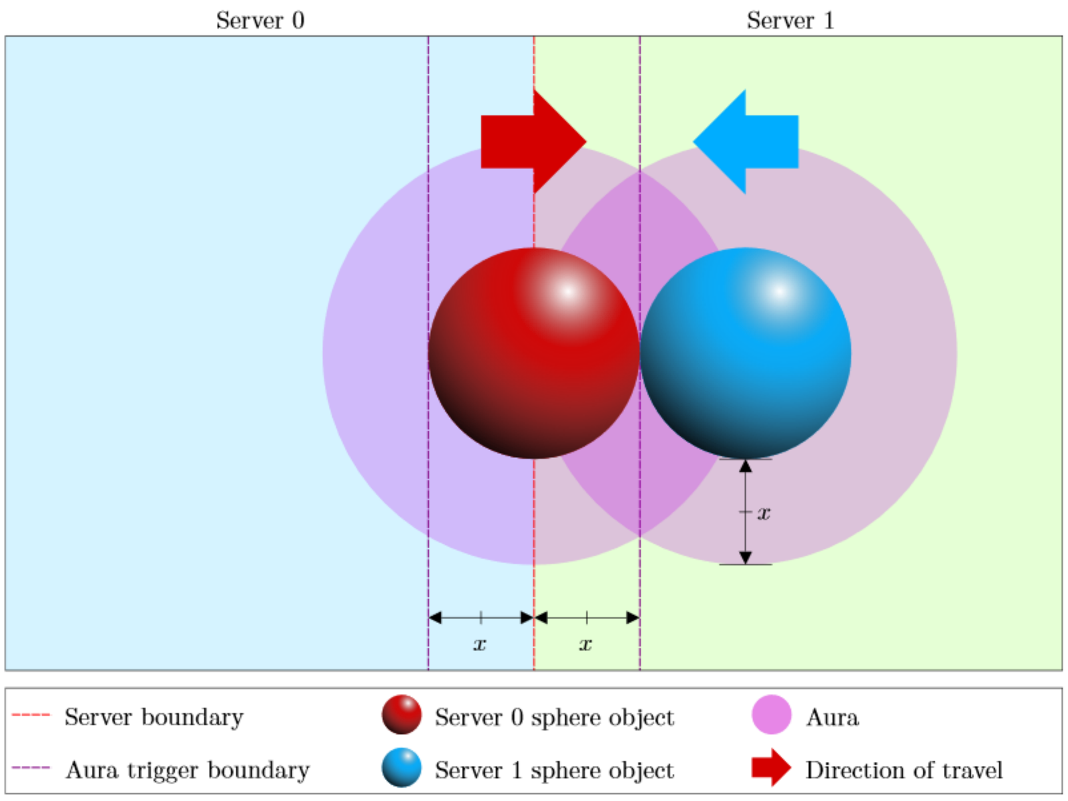


Fig. 5.19 Correctness experiment setup. The setup is the ‘worst-case’ scenario, in which two objects are moving directly towards each other and collide while one of the objects is intersecting the boundary between servers.

same as spheres in a centralised solution, therefore only collisions that involve a sphere that has just migrated are considered in the experiment results.

There are three factors that are used to determine aura size: speed; latency; and frame-time. In order to demonstrate that collisions always remain correct when within their tolerance, even under worsening conditions and the effect each factor has on collision correctness, one aura factor is varied per experiment and the other two factors are set to equal the tolerance values. For example, speed is varied, the latency between the servers is set to equal the latency tolerance value and the frame-time of each server is set to equal the frame-time tolerance value. The non-varied factors are set to equal the tolerance values as exceeding the tolerance value in one factor can be compensated for by having a value below the tolerance in a different factor. For example, the latency may exceed the tolerance by $5ms$, but the frame-time could be more than $5ms$ below the frame-time tolerance and the collision would still be expected to be correctly handled by AP.

Speed is controlled such that the two objects used in the experiment collide with each other with that desired speed. Latency is controlled using the Traffic Control tool to emulate latency for all communications between the servers. The desired latency is applied as a fixed delay to each server, resulting in that delay being applied to all packets outgoing from that server. Frame-time is controlled using a wait within the simulation’s update loop, which is limited to a target frame-time as the resultant frame-times are a normal distribution with the target frame-time as the median. This is illustrated in Fig. 5.20. As a result of this, half of all

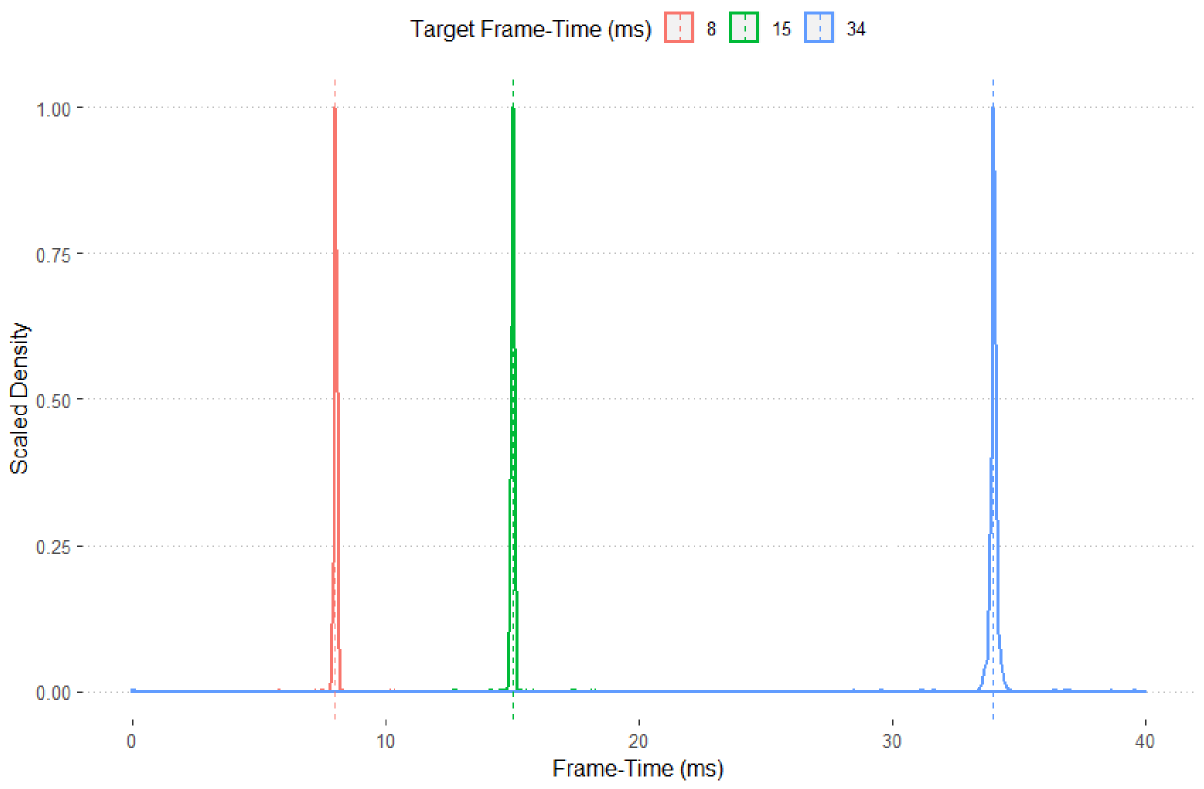


Fig. 5.20 Scaled Density of frame-times for the different target frame-times used in the collision correctness experiments. The median of each target frame-time is marked with a vertical dashed line. As target frame-time increases, the distribution of actual frame-time also increases. Frame-time is controlled using a wait within the simulation's update loop, which is limited to a target frame-time as the resultant frame-times are a normal distribution with the target frame-time as the median. As a result of this, half of all frames are expected to be above the target frame-time, which may cause false-positives in detecting late collisions.

frames are expected to be above the target frame-time, which may cause false-positives in detecting late collisions, in the following experiments. However, the following points should be considered: the variance in frame-time is low; longer frames have to occur exactly at the time of messages being exchanged; exceeding one factor can be compensated for by having a low value in another factor. Therefore, false-positives are expected to be rare up until the tolerance value is approached. It should be noted that variance increases with target frame-time, therefore more false-positives should expected with experiments using higher target frame-times.

For the varying factors experiments, the following parameters were used and output given. The two servers use the following tolerances unless otherwise stated: a maximum speed tolerance of $32m{\cdot}s^{-1}$; maximum latency tolerance of $2ms$; and a maximum frame-time tolerance of $15ms$. Each server has a random delay before starting between 0 and the target frame-time in order to prevent the two servers always having synchronous update cycles. Each experiment is repeated 50 times. An experiment was conducted for each aura tolerance value. The aura size for the varying tolerances are plotted and the tolerances used are marked. For each experiment, three tolerances values were chosen. For each tolerance value, the following output graphs were plotted:

- The ratio of late to correct collisions and ratio of misses to total collision.

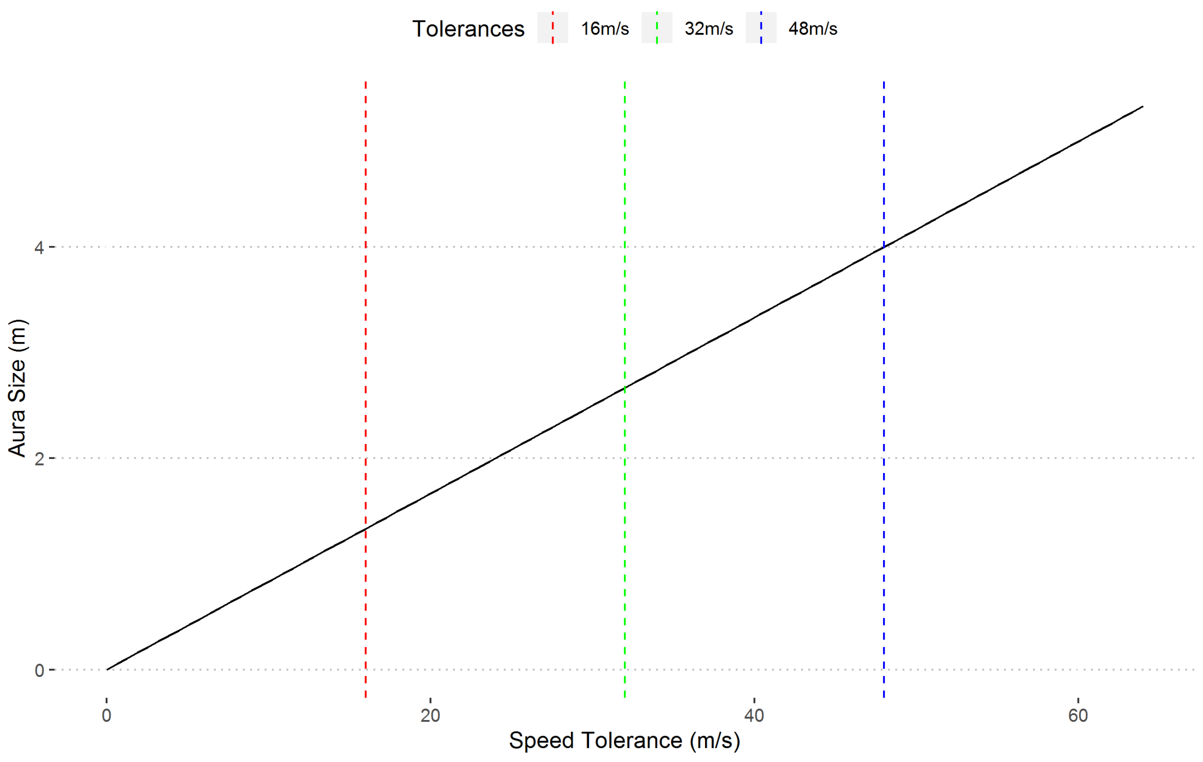


Fig. 5.21 Size of aura vs. speed tolerance. The speed tolerances used are marked in dashed lines

- The mean and $+/-2$ standard deviations of penetration time were plotted against each tolerance factor for each of the three tolerance values chosen.

The first experiment conducted was the varying speed experiment. In this experiment the speed of each of the two objects was varied from $1m{\cdot}s^{-1}$ to double the the following tolerances: $16m{\cdot}s^{-1}$; $32m{\cdot}s^{-1}$ and $48m{\cdot}s^{-1}$. Increments of $1m{\cdot}s^{-1}$ were used and the whole experiment repeated 50 times. In these experiments, it is assumed 1 unit is 1 metre. The speed tolerances chosen represent categories of typical fast-moving objects in games. The categories selected were: slow-moving vehicles ($16m{\cdot}s^{-1}$); fast-moving/motorway-speed vehicles ($32m{\cdot}s^{-1}$); and racing vehicles ($48m{\cdot}s^{-1}$).

Fig. 5.21 shows the size of the aura (not including the bounding sphere of the object) vs increasing speed tolerance, calculated using equation 3.4. The aura size increases continuously with speed, therefore we expect late collisions to begin immediately once the the speed tolerance is exceeded. False-positives are expected due to the frame-time delay method and should increase as speed increases.

The second experiment conducted was the varying latency experiment. In this experiment the latency between the two servers was varied from $0ms$ to double the following tolerances: $2ms$; $8ms$ and $16ms$ in increments of $0.05ms$, $0.25ms$ and $1ms$ respectively. The latency used is the target latency, in reality the latency will never exactly be equal to $0ms$.

Fig. 5.28 shows the size of the aura (not including the bounding sphere of the object) vs increasing latency tolerance, calculated using equation 3.4. The aura size increases discretely with latency tolerance, for example tolerance values from $4ms$ to $20ms$ result in the same aura size. Late collisions are expected to occur only once the actual latency exceeds the maximum

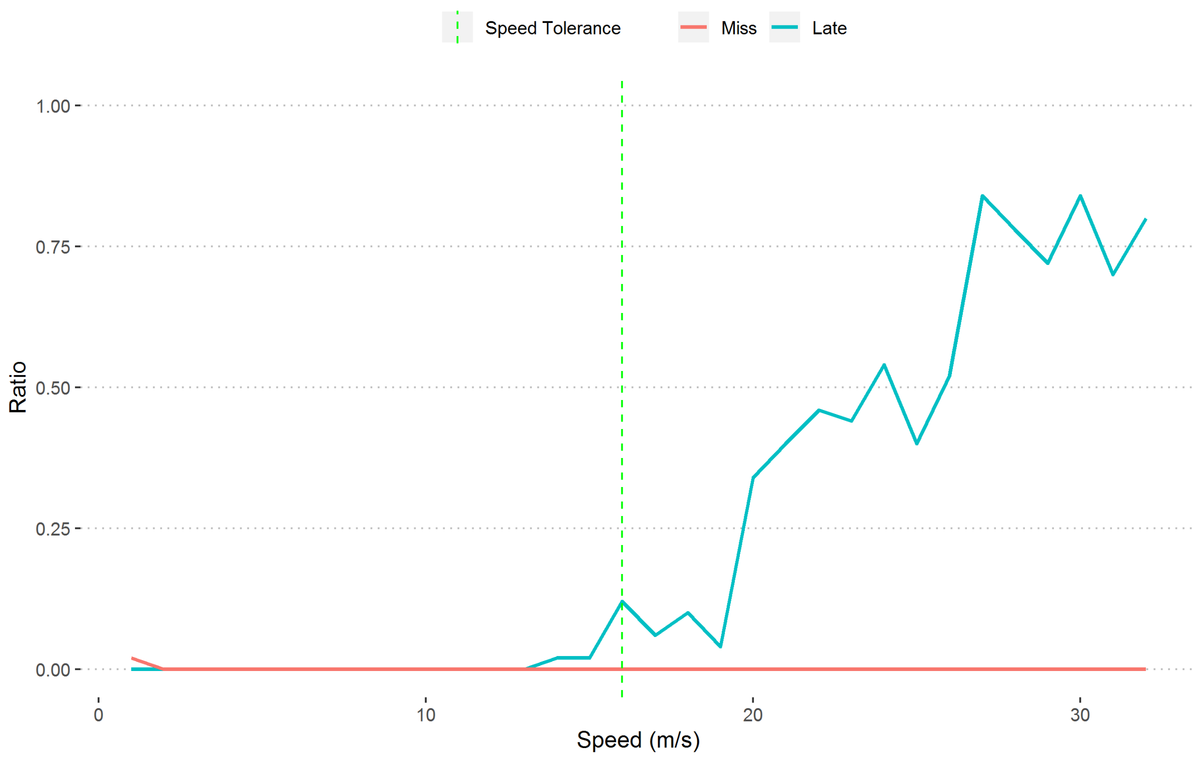


Fig. 5.22 Ratio of Late to Correct Collisions and Misses to Total Collision vs. Speed using a speed tolerance of $16 m{\cdot}s^{-1}$. The speed tolerance is marked in a dashed green line.

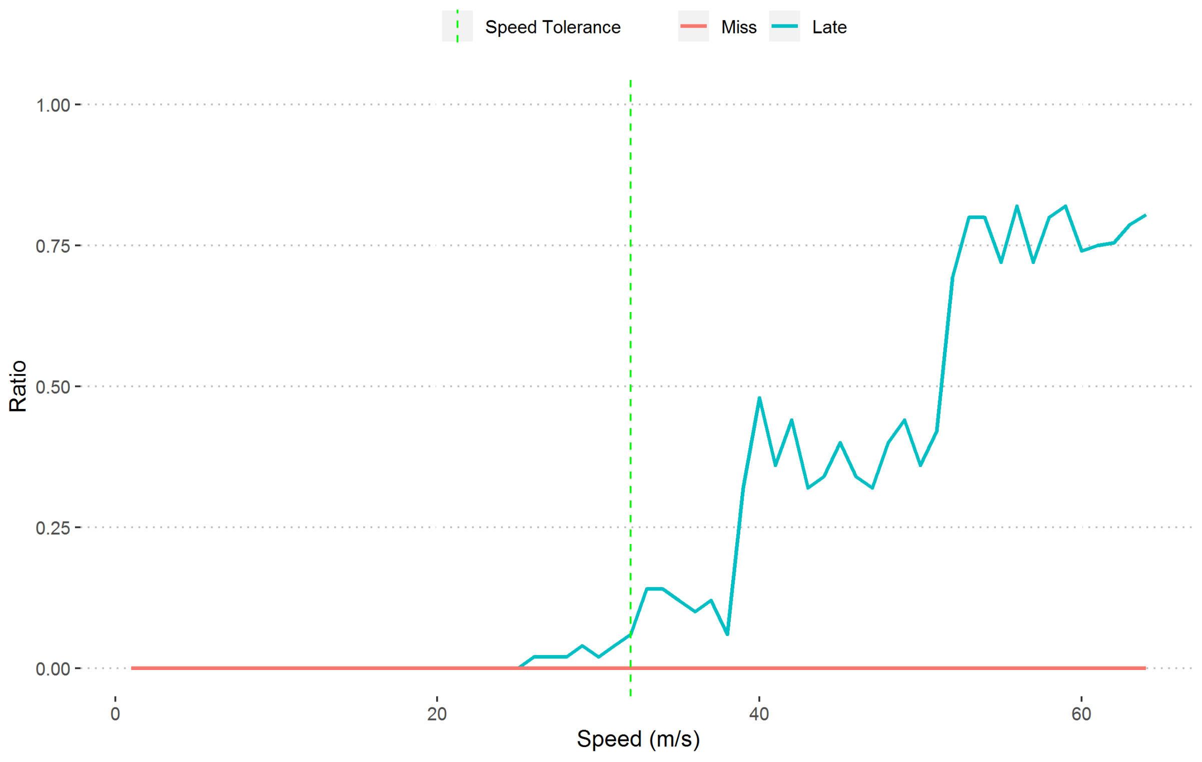


Fig. 5.23 Ratio of Late to Correct Collisions and Misses to Total Collision vs. Speed using a speed tolerance of $32 m{\cdot}s^{-1}$. The speed tolerance is marked in a dashed green line.

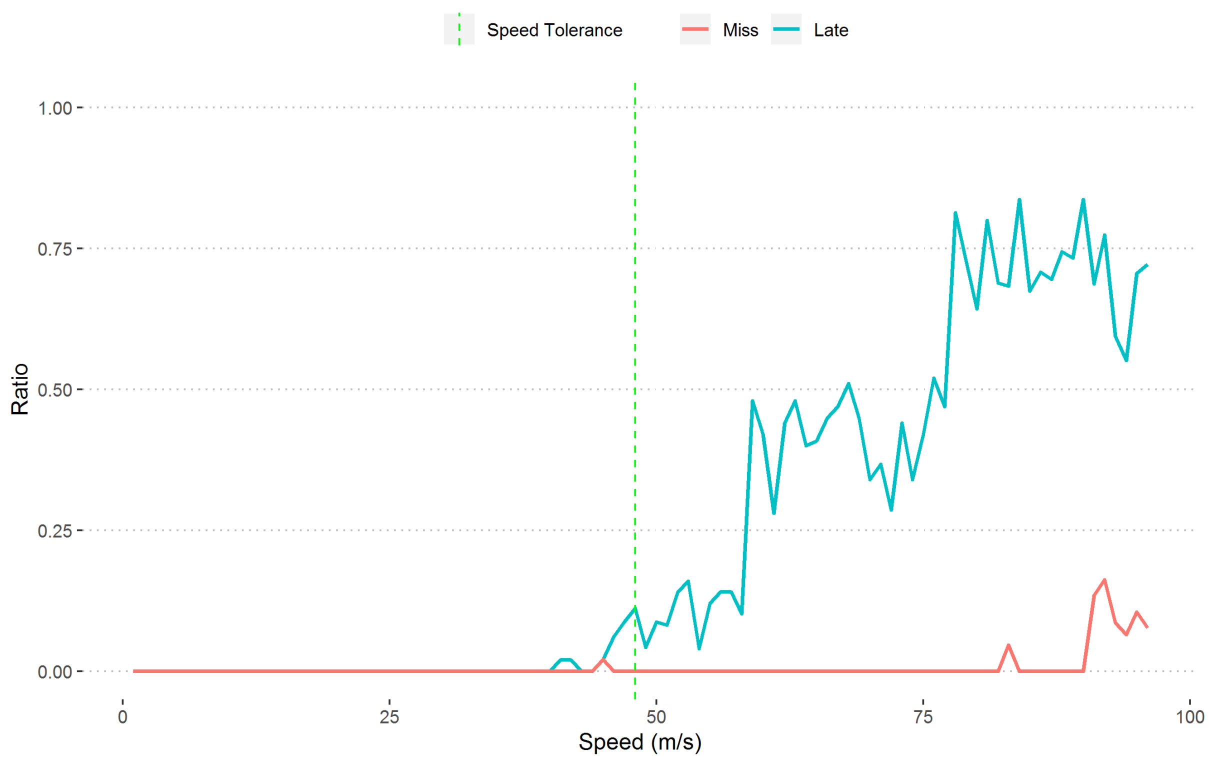


Fig. 5.24 Ratio of Late to Correct Collisions and Misses to Total Collision vs. Speed using a speed tolerance of $48m{\cdot}s^{-1}$. The speed tolerance is marked in a dashed green line.

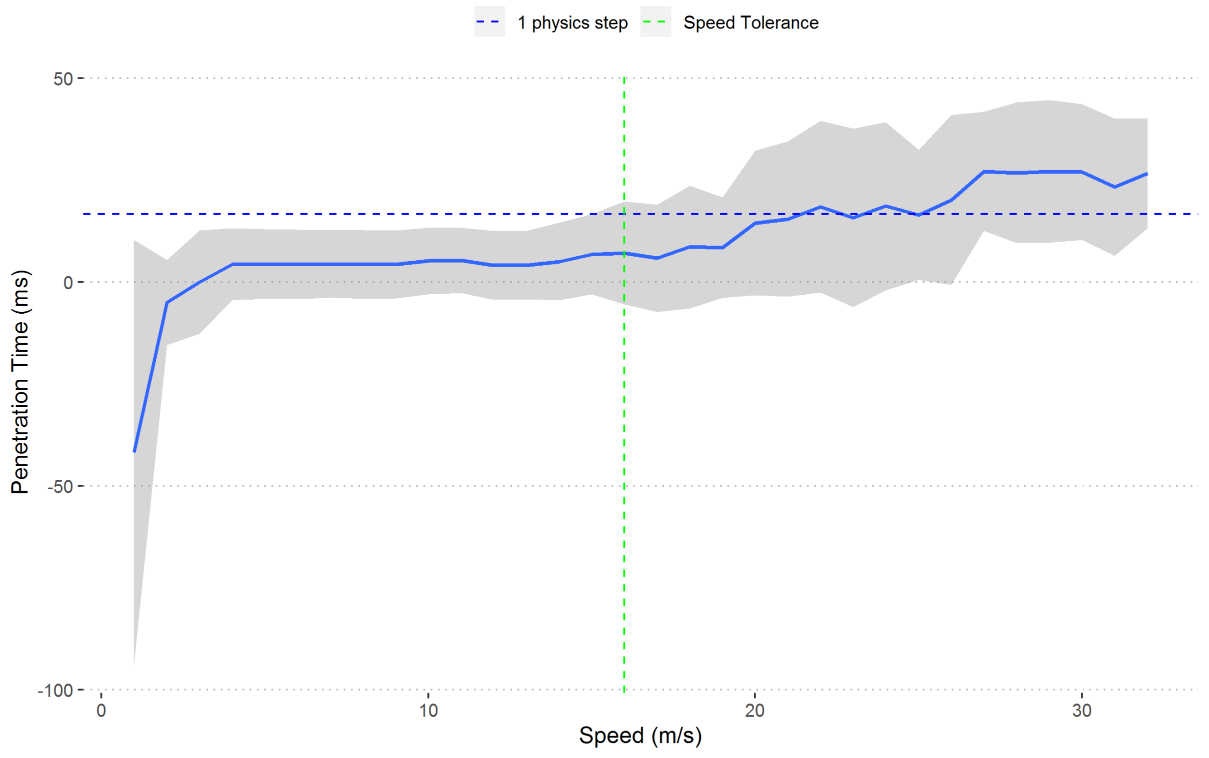


Fig. 5.25 Mean penetration time of objects with $+/-2$ standard deviations with varying speed using a speed tolerance of $16m{\cdot}s^{-1}$. The maximum expected penetration time of 1 physics step is marked with a dashed blue line. The speed tolerance is marked in a dashed green line.

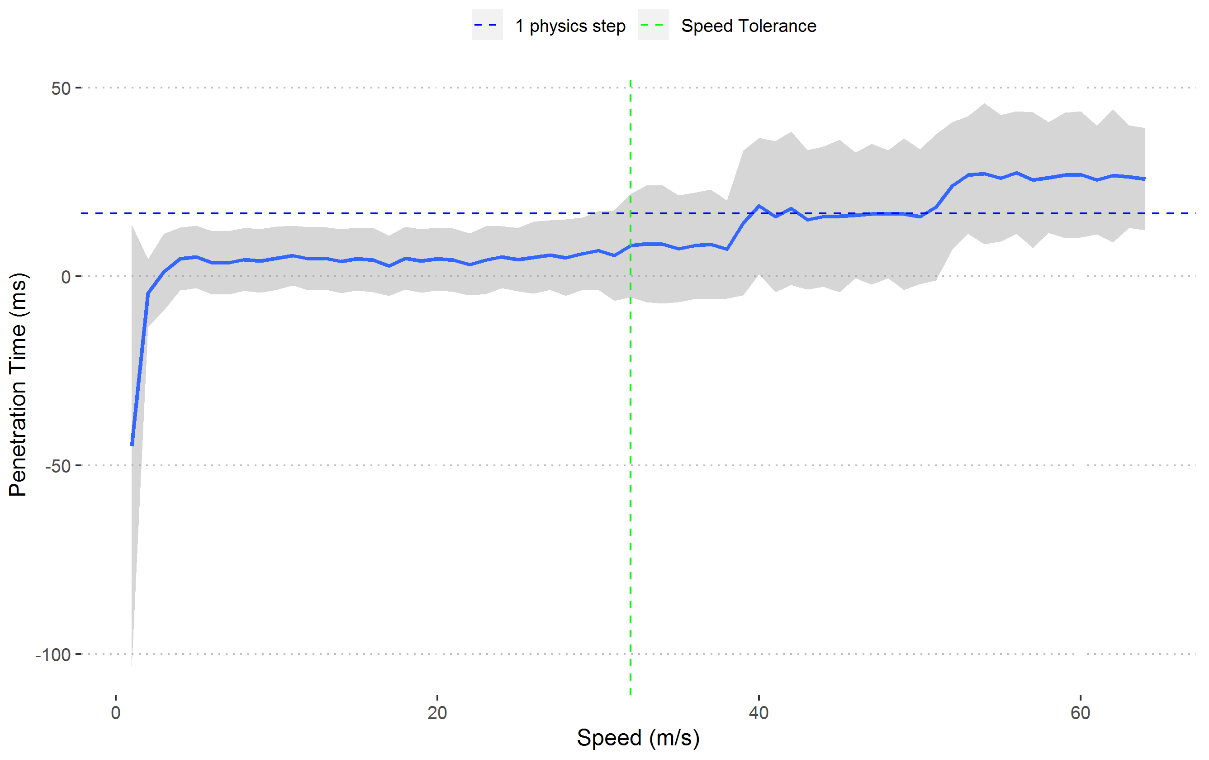


Fig. 5.26 Mean penetration time of objects with $+/-2$ standard deviations with varying speed using a speed tolerance of $32m{\cdot}s^{-1}$. The maximum expected penetration time of 1 physics step is marked with a dashed blue line. The speed tolerance is marked in a dashed green line.

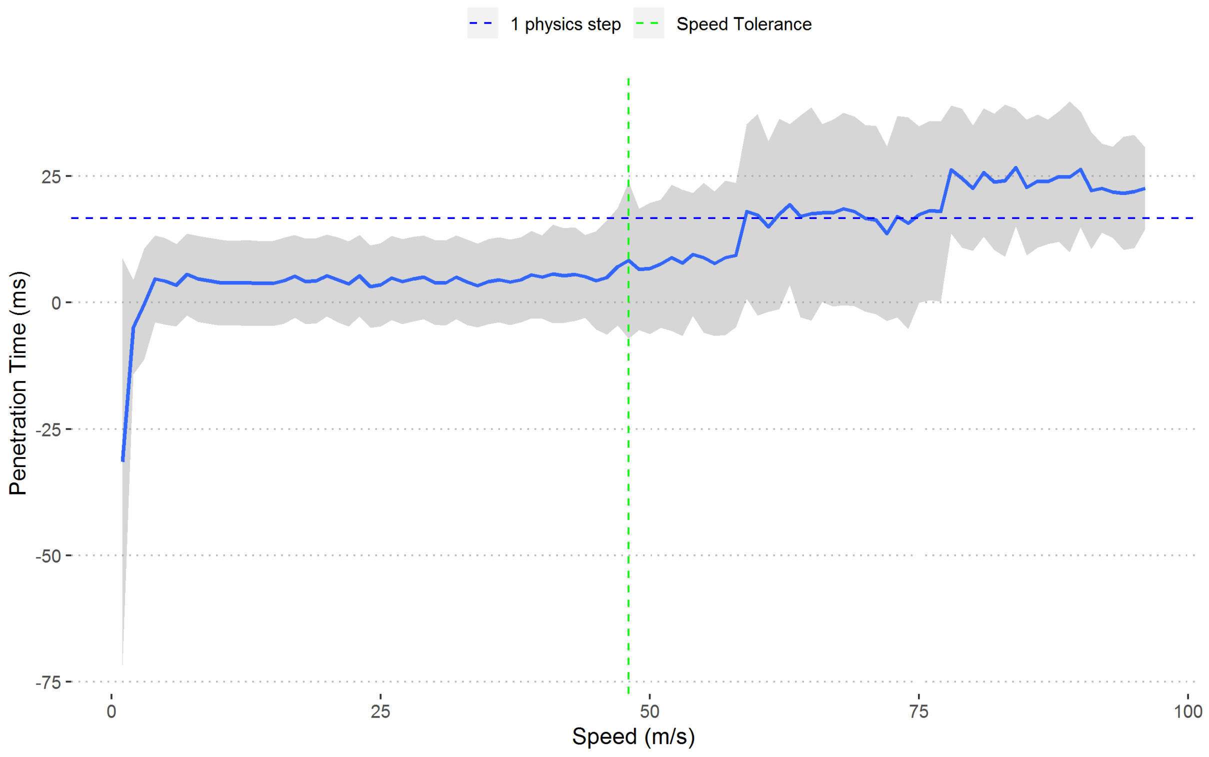


Fig. 5.27 Mean penetration time of objects with $+/-2$ standard deviations with varying speed using a speed tolerance of $48m{\cdot}s^{-1}$. The maximum expected penetration time of 1 physics step is marked with a dashed blue line. The speed tolerance is marked in a dashed green line.

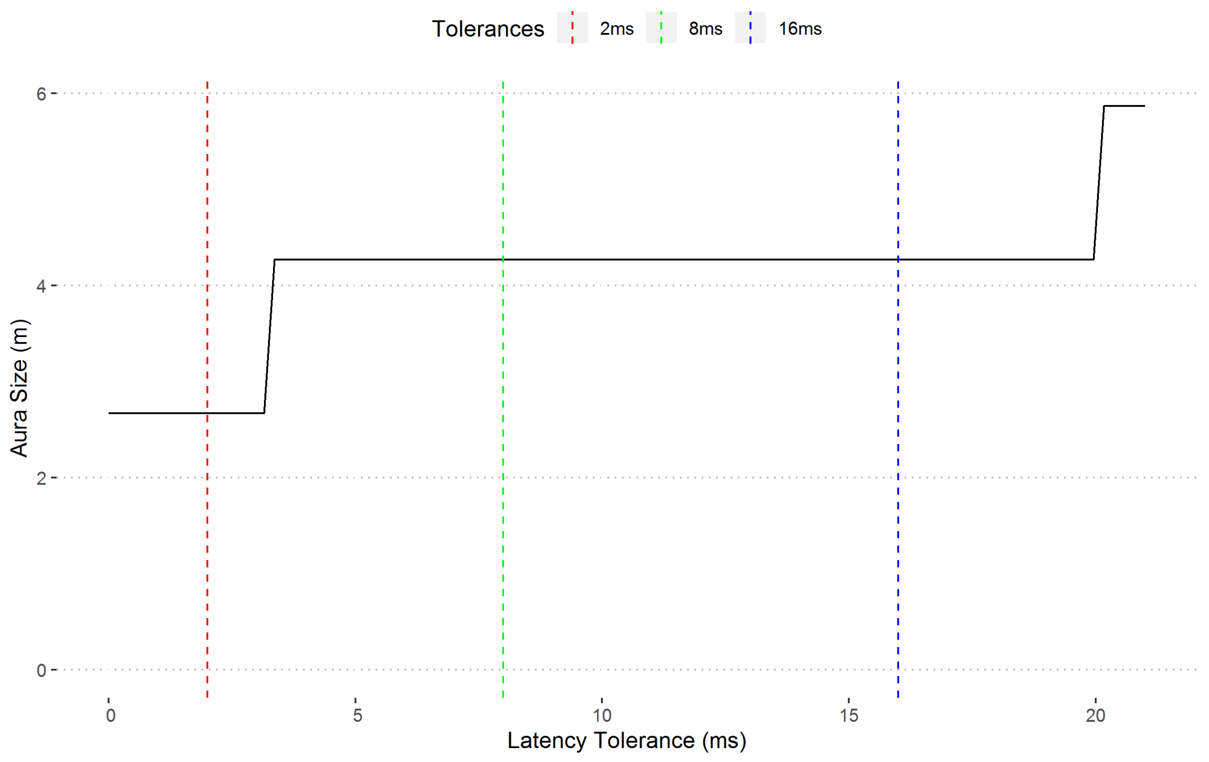


Fig. 5.28 Size of aura vs. latency tolerance. The latency tolerances used are marked in dashed lines

tolerance for the current aura size. In the following experiments, this means that in the $2ms$ tolerance experiments, late collisions are not expected to occur until the actual latency exceeds $3ms$. Likewise, in the $8ms$ and $16ms$ tolerance experiments, $20ms$ actual latency is expected to be reached before late collisions begin to occur. Note that late collisions are not expected for the $8ms$ experiment, as $20ms$ latency is not reached. False-positives are expected due to the frame-time delay method and should increase as actual latency increases.

The third experiment conducted was the varying frame-time experiment. In this experiment the frame-time was varied from $0ms$ to double the frame-time tolerance for the following frame-times: $8.33ms$ $(120Hz)$; $15ms$ $(66.67Hz)$ and $33.33ms$ $(30Hz)$. Increments of $1ms$, $1ms$ and $2ms$ were used respectively. The frame-time used is target frame time, in reality frame-time cannot be equal to exactly $0ms$.

Fig. 5.35 shows the size of the aura (not including the bounding sphere of the object) vs increasing frame-time tolerance, calculated using equation 3.4. The aura size increases discretely with frame-time tolerance, therefore we expect late collisions to begin only once the the actual frame-time is higher than the maximum frame-time the current aura size tolerates. In the varying frame-time experiments, this will be when frame-time reaches $16ms$ for the $8ms$ and $15ms$ tolerance experiments and $32ms$ for the $32ms$ tolerance experiment. Note that late collisions are not expected for the $8ms$ experiment, as $16ms$ frame-time is not exceeded. False-positives are expected due to the frame-time delay method and should increase as frame-time increases.

In addition to recording the number of collisions that resulted in thrashing, the number of times an object thrashed between servers was also measured. The number of times thrashed is

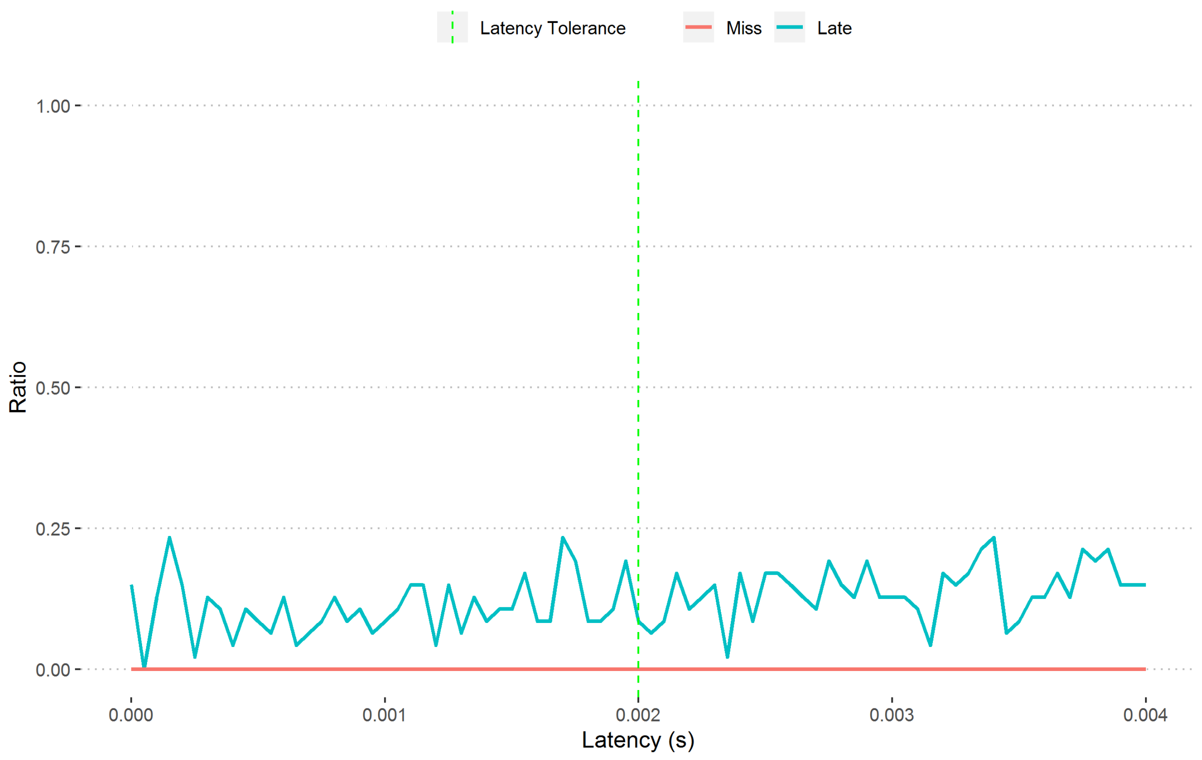


Fig. 5.29 Ratio of Late to Correct Collisions and Misses to Total Collision vs. Latency using a latency tolerance of $2ms$. The latency tolerance is marked in a dashed green line.

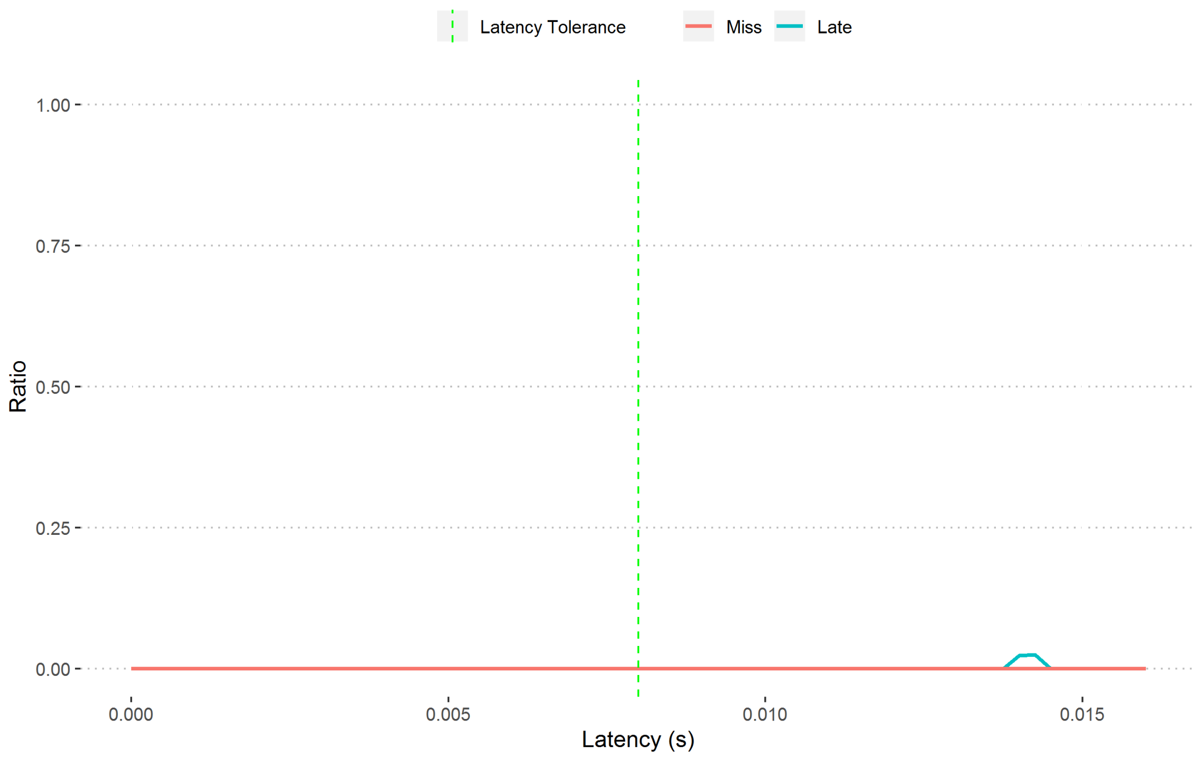


Fig. 5.30 Ratio of Late to Correct Collisions and Misses to Total Collision vs. Latency using a latency tolerance of $8ms$. The latency tolerance is marked in a dashed green line.

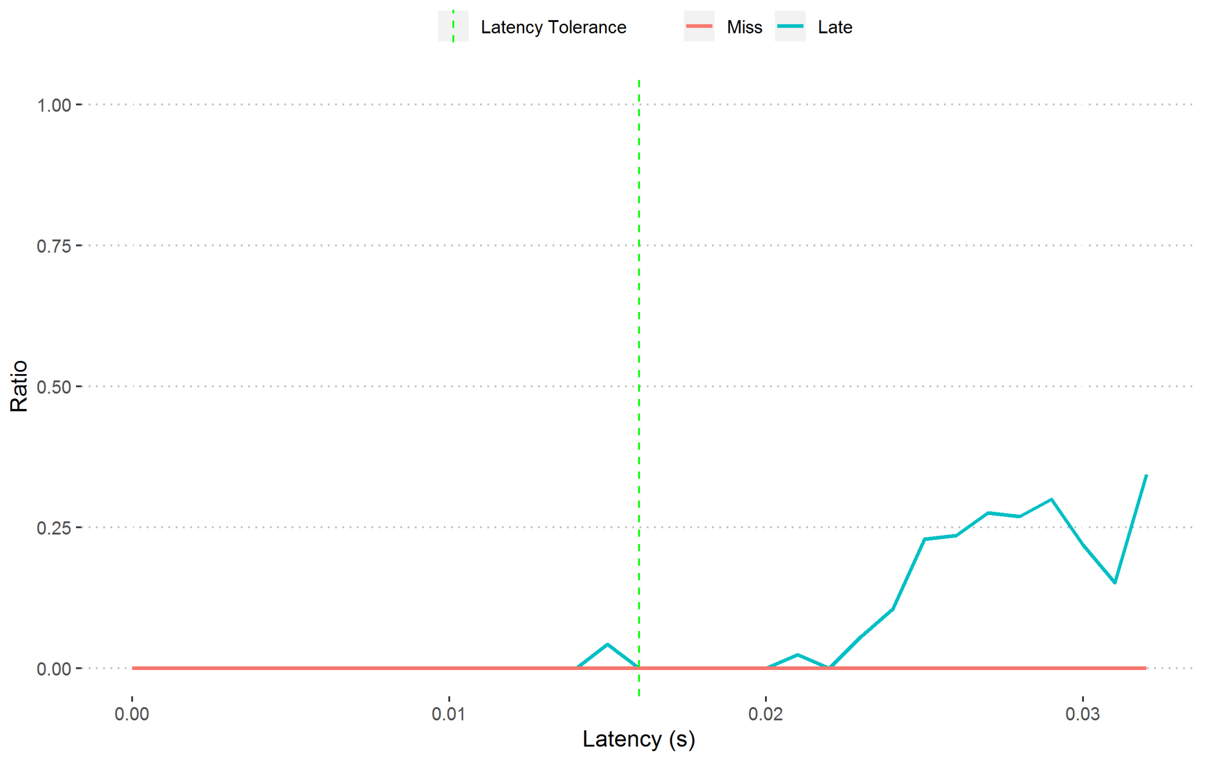


Fig. 5.31 Ratio of Late to Correct Collisions and Misses to Total Collision vs. Latency using a latency tolerance of $16ms$. The latency tolerance is marked in a dashed green line.

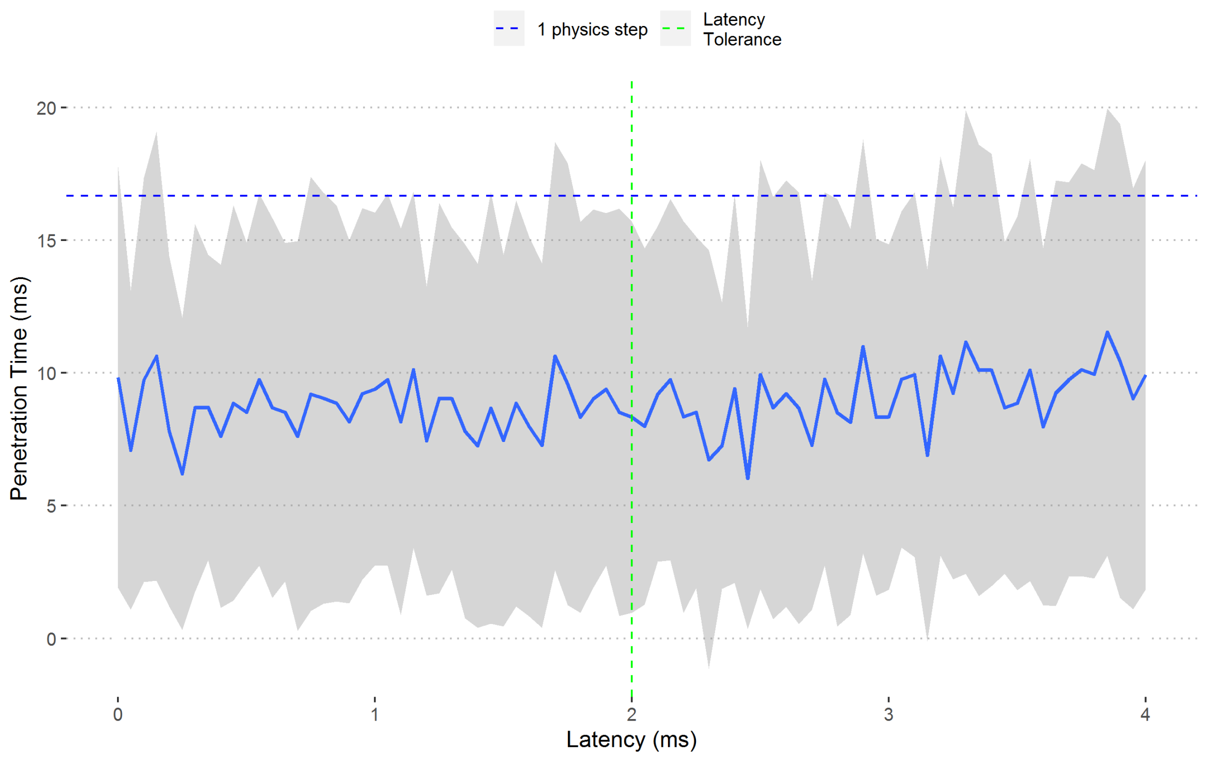


Fig. 5.32 Mean penetration time of objects with $+/-2$ standard deviations with varying latency using a latency tolerance of $2ms$. The maximum expected penetration time of 1 physics step is marked with a dashed blue line. The latency tolerance is marked in a dashed green line.

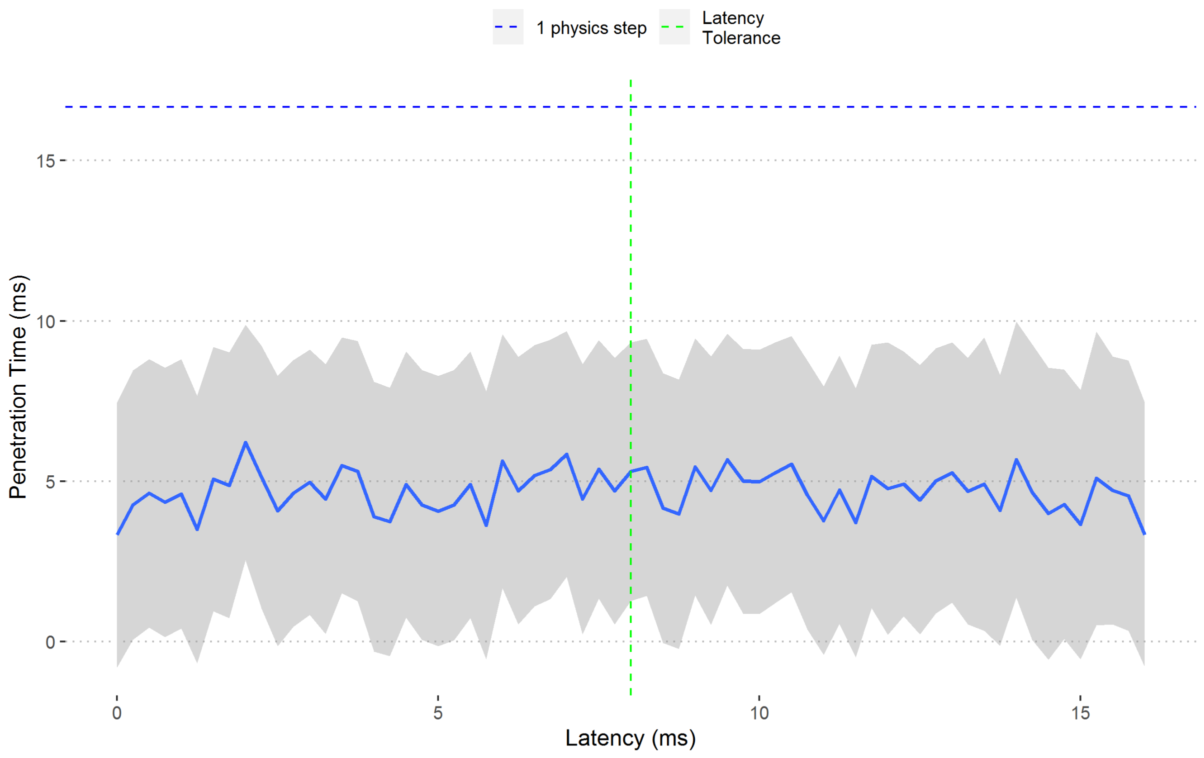


Fig. 5.33 Mean penetration time of objects with $+/-2$ standard deviations with varying latency using a latency tolerance of $8ms$. The maximum expected penetration time of 1 physics step is marked with a dashed blue line. The latency tolerance is marked in a dashed green line.

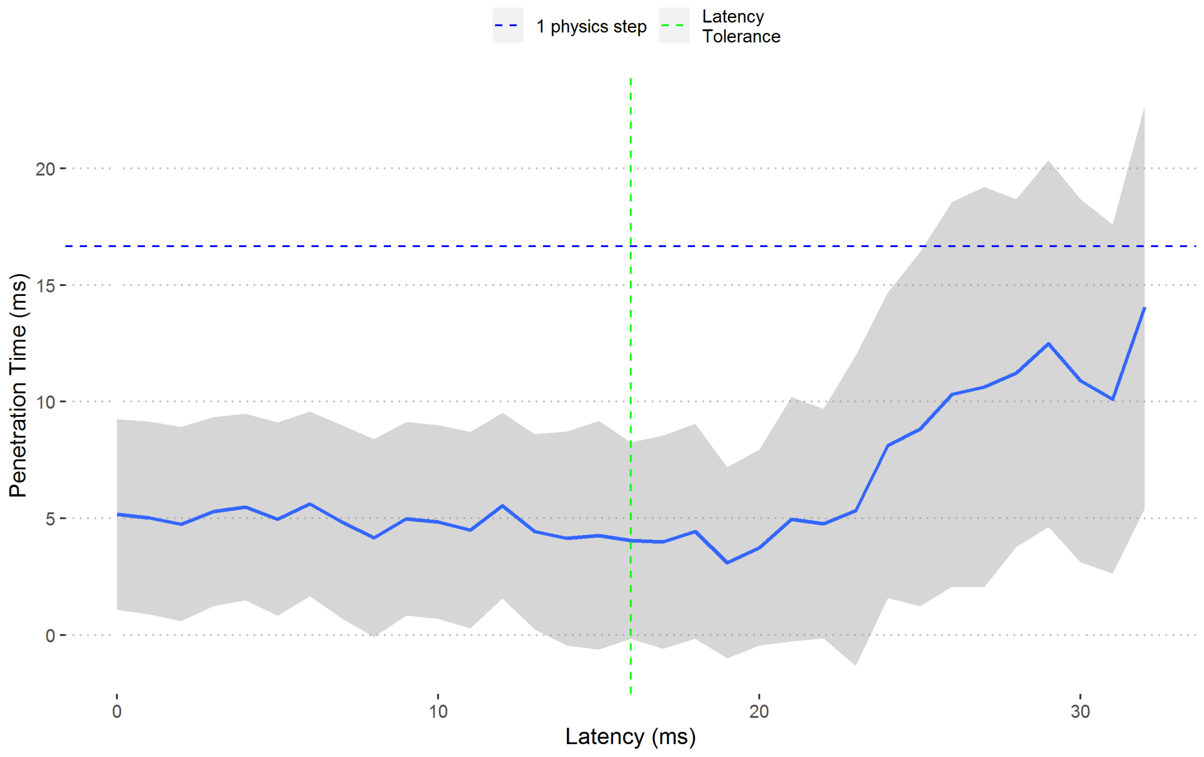


Fig. 5.34 Mean penetration time of objects with $+/-2$ standard deviations with varying latency using a latency tolerance of $16ms$. The maximum expected penetration time of 1 physics step is marked with a dashed blue line. The latency tolerance is marked in a dashed green line.

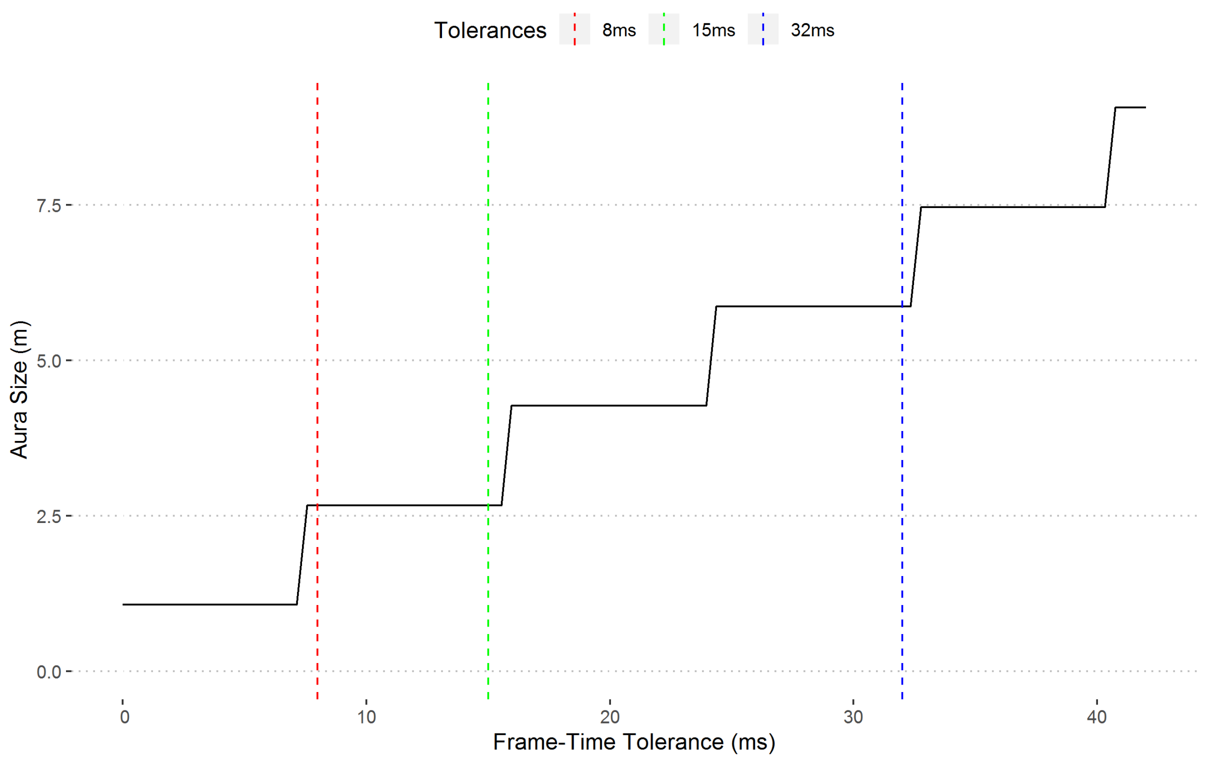


Fig. 5.35 Size of aura vs. frame-time tolerance. The frame-time tolerances used are marked in dashed lines

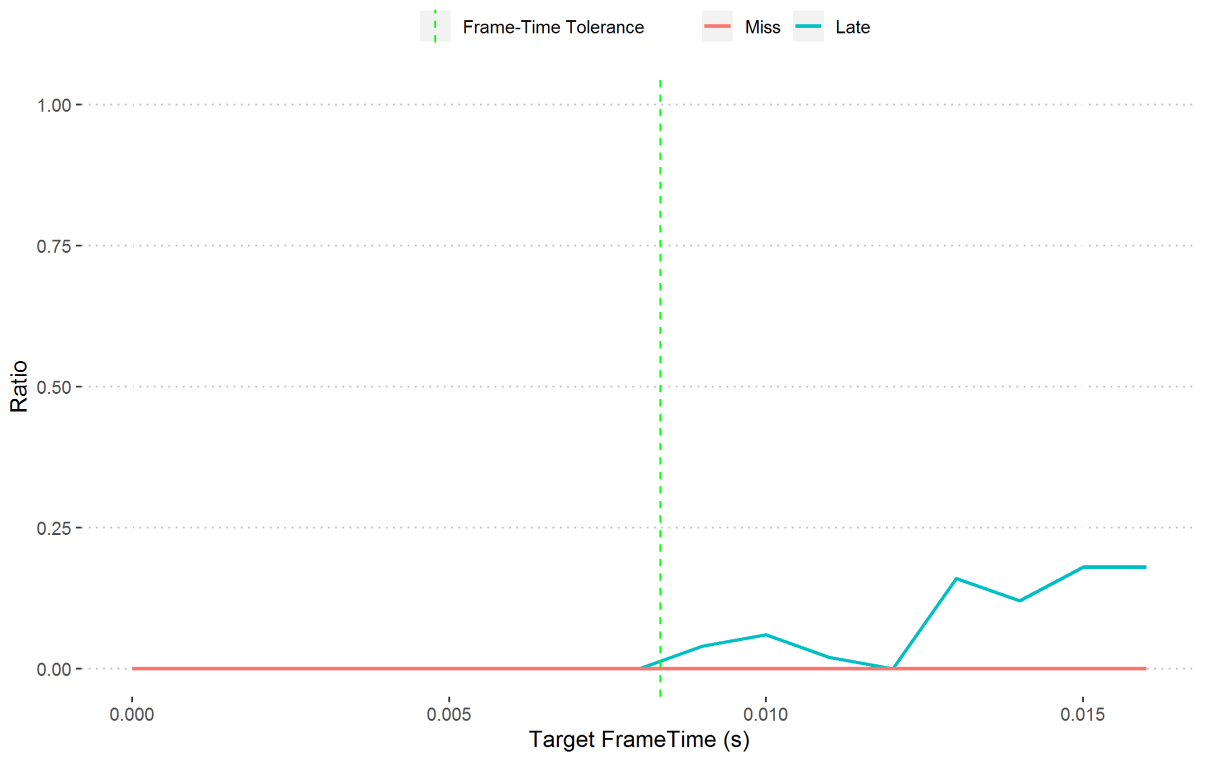


Fig. 5.36 Ratio of Late to Correct Collisions and Misses to Total Collision vs. frame-time using a frame-time tolerance of 8*ms*. The frame-time tolerance is marked in a dashed green line.

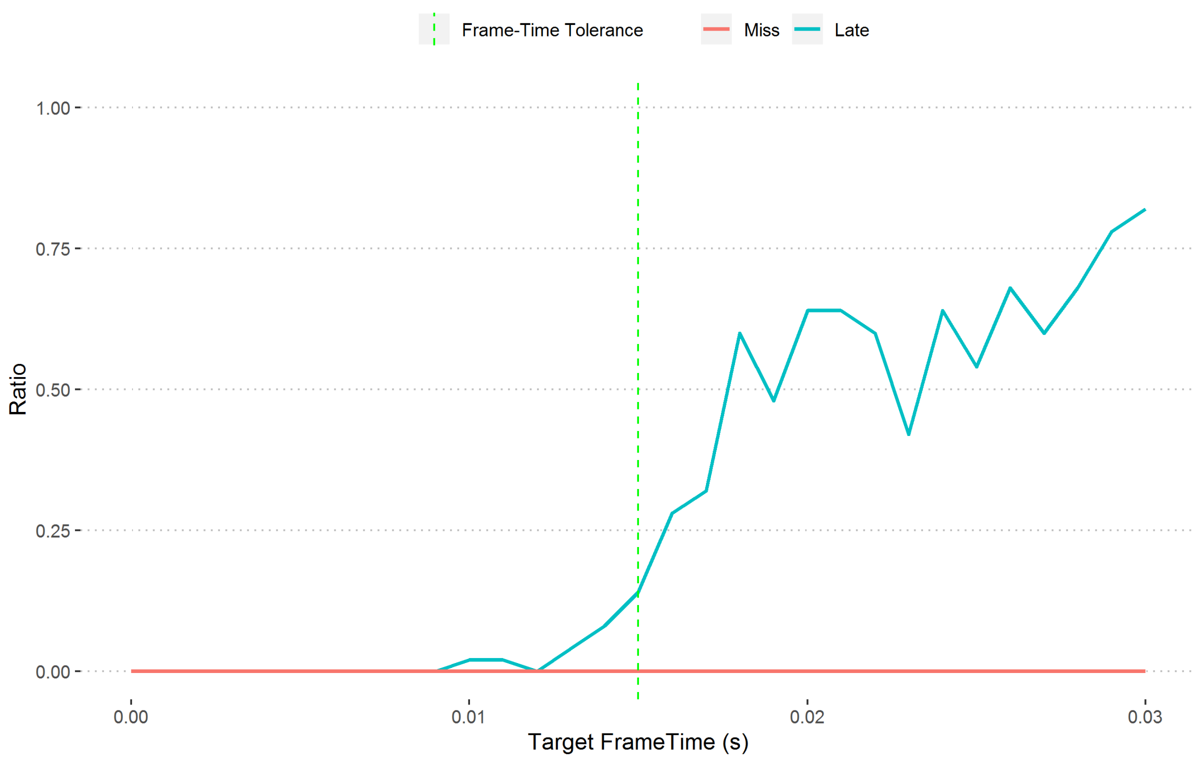


Fig. 5.37 Ratio of Late to Correct Collisions and Misses to Total Collision vs. frame-time using a frame-time tolerance of 15*ms*. The frame-time tolerance is marked in a dashed green line.

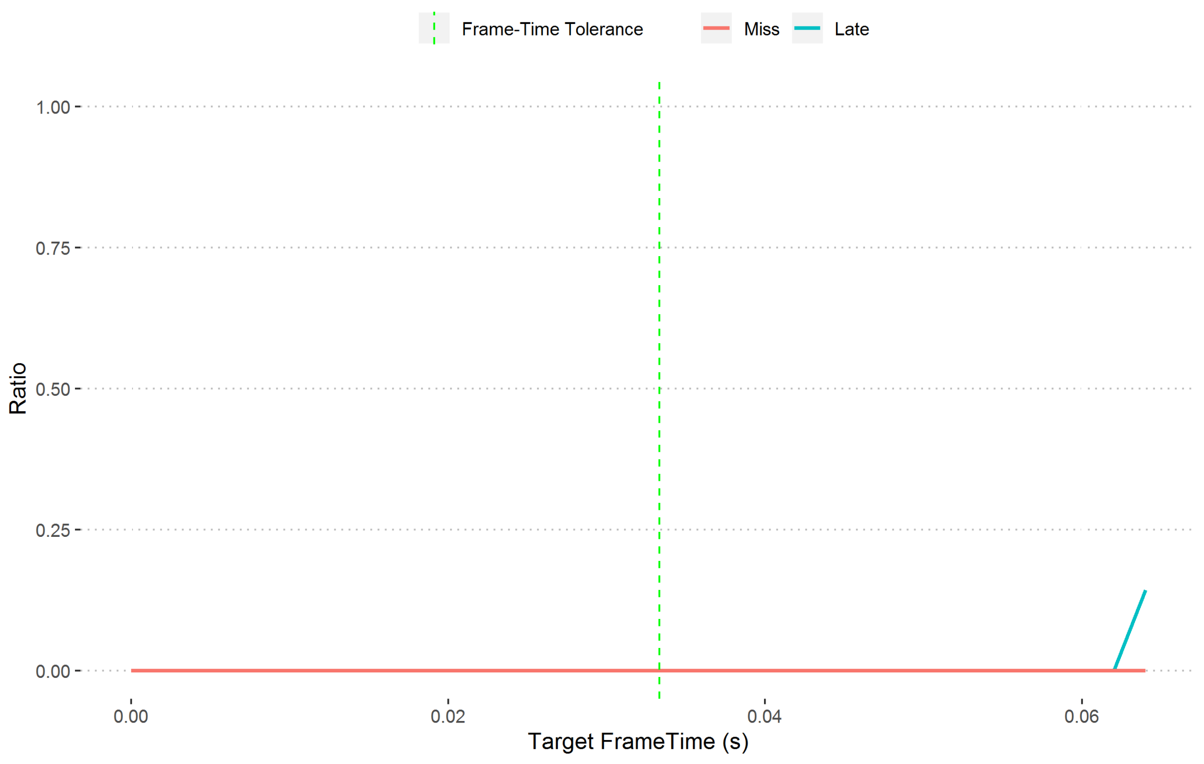


Fig. 5.38 Ratio of Late to Correct Collisions and Misses to Total Collision vs. frame-time using a frame-time tolerance of 32*ms*. The frame-time tolerance is marked in a dashed green line.

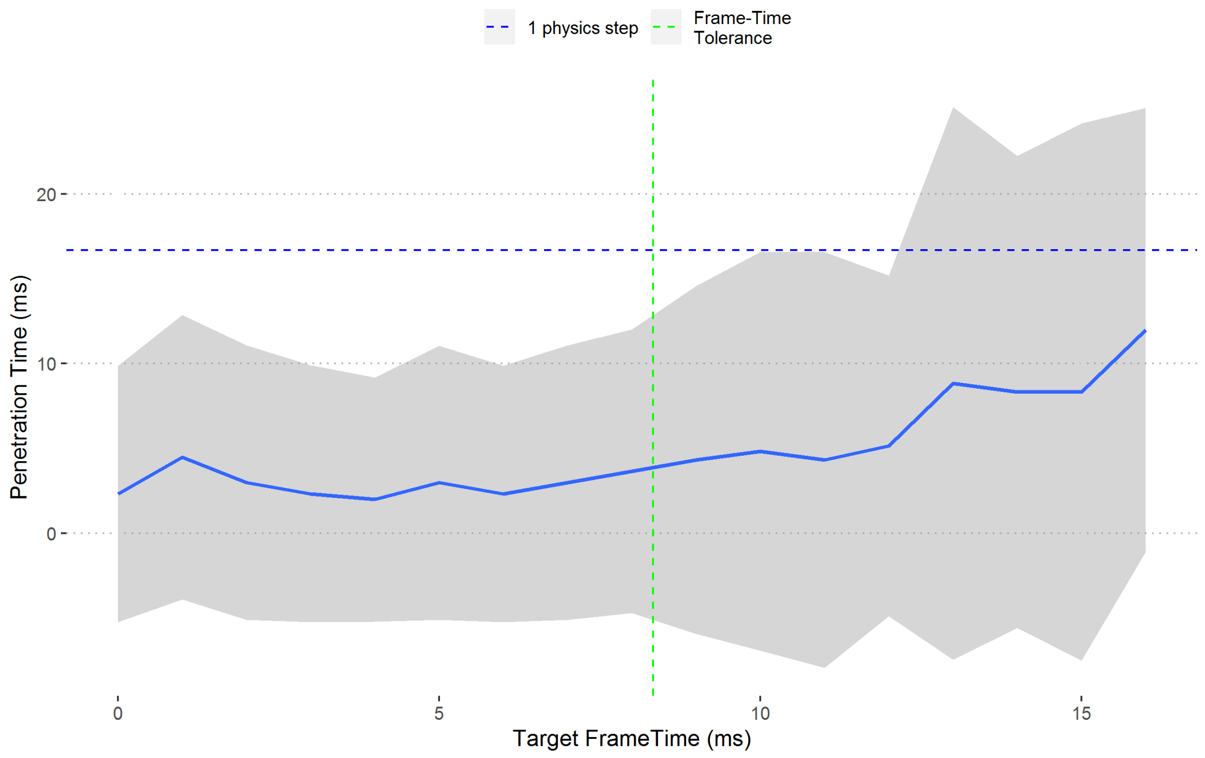


Fig. 5.39 Mean penetration time of objects with $+/-2$ standard deviations with varying frame-time using a frame-time tolerance of $8ms$. The maximum expected penetration time of 1 physics step is marked with a dashed blue line. The frame-time tolerance is marked in a dashed green line.

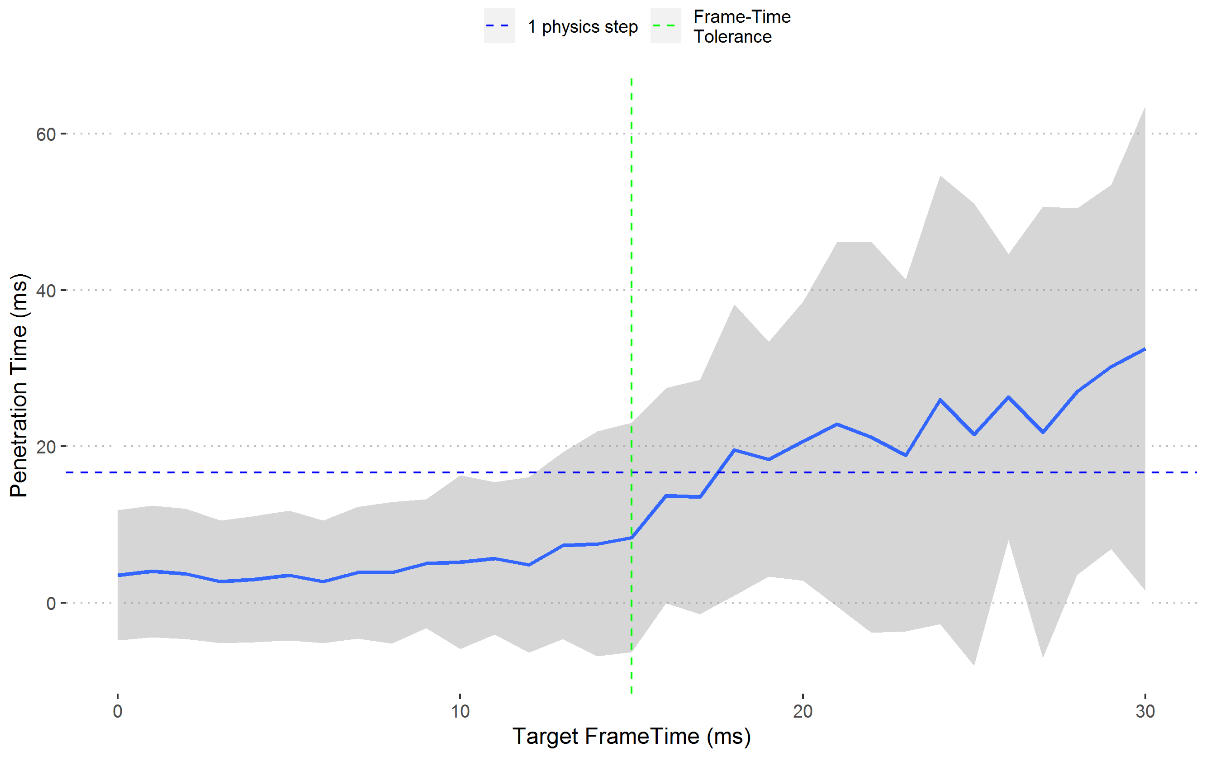


Fig. 5.40 Mean penetration time of objects with $+/-2$ standard deviations with varying frame-time using a frame-time tolerance of $15ms$. The maximum expected penetration time of 1 physics step is marked with a dashed blue line. The frame-time tolerance is marked in a dashed green line.

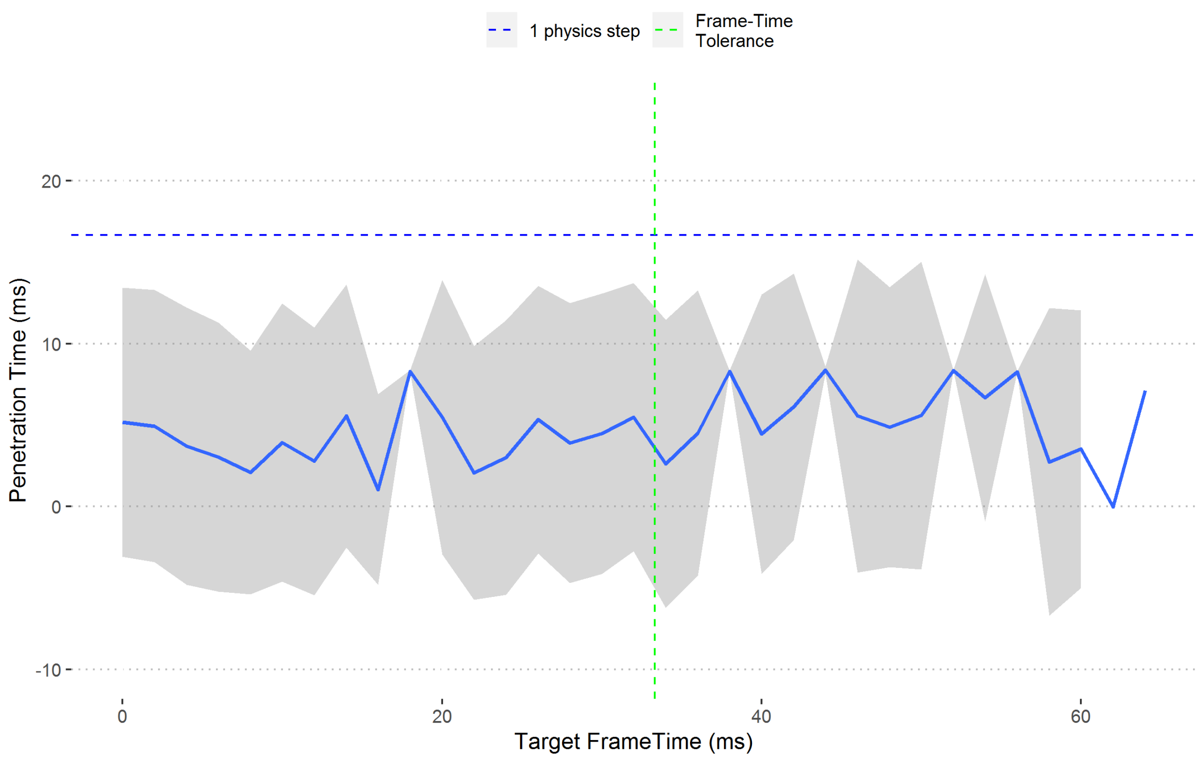


Fig. 5.41 Mean penetration time of objects with $+/-2$ standard deviations with varying frame-time using a frame-time tolerance of $32ms$. The maximum expected penetration time of 1 physics step is marked with a dashed blue line. The frame-time tolerance is marked in a dashed green line.

Factor	Tolerance	N	Late %	Miss %	Thrashing
Speed	16m/s	800	1	0.13	0
Speed	32m/s	1600	0.688	0.0422	0
Speed	48m/s	2372	0.632	0	28
Latency	2ms	1920	10.7	0	7
Latency	8ms	1520	0	0	30
Latency	16ms	797	2.50	0	53
Frame-Time	8.33ms	450	0	0	0
Frame-Time	15ms	800	1.88	0	0
Frame-Time	33.33ms	251	0	0	523

Table 5.1 Results summary of collision correctness experiments before the tolerance was exceeded

Factor	Tolerance	N	Late %	Miss %	Thrashing
Speed	16m/s	800	49.9	0	0
Speed	32m/s	1589	48.6	0	11
Speed	48m/s	2233	45.10	1.12	167
Latency	2ms	1880	13.99	0	0
Latency	8ms	1377	0.15	0	121
Latency	16ms	574	12.20	0	226
Frame-Time	8.33ms	400	9.50	0	0
Frame-Time	15ms	750	58.13	0	0
Frame-Time	33.33ms	106	1.89	0	693

Table 5.2 Results summary of collision correctness experiments after the tolerance was exceeded

determined by measuring the number of migrations away from the server that the collisions takes place on, before the collision takes place. A histogram of the number of occurrences against the number of times thrashed is shown in Fig. 5.42. For this histogram, data from the Frame-Time 33.33ms experiment is used, as this provides the largest sample of thrashing. Fig. 5.42 demonstrates that the most frequent number of times thrashed is 1, by a large margin followed by a downward trend.

The more times thrashing occurs, the later the collision may be detected as objects continue to be updated and move without a collision occurring. It should be noted that thrashing occurring does not guarantee that a late collision will take place. The more times thrashing takes place, the more physics updates occur and the further the objects may be moved towards (and eventually overlapping) each other. Therefore, a greater number of times thrashed, means a late and incorrect collision response is more likely, but not guaranteed.

5.2.4 Experiment- Varying Factors Evaluation

As the speed increases, the proportion of late collisions to correct collisions increases, as shown in Figs. 5.22, 5.23 and 5.24. When speed is below the speed tolerance, late collisions do not exceed 10%. This 10% is assumed to be caused by false-positives. Once above the speed tolerance, the proportion of late collisions increases immediately, which is predicted by the aura calculation. Missed collisions begin to occur when speed exceed 80ms^{-1} in the 48ms^{-1} experiment, this is due to objects travelling so quickly, the objects completely pass one another, missing the collision entirely.

It should also be noted that object speed is application specific and objects need to be travelling directly towards the centre of each other and one of the objects has to be travelling above the tolerance speed, in order for errors to occur.

As the speed increases, the mean penetration time increases, as shown in Figs. 5.25, 5.26 and 5.27. With the exception of the starting speed, the standard deviation remains constant below the tolerance line. Once speed increases above the tolerance line, the standard deviation initially increases, but appears to remain constant after the initial increase.

As the latency increases, the proportion of late collisions to correct collisions increases, as shown in Fig. 5.31. However, Figs. 5.29 and 5.29 do not show this relationship. In the

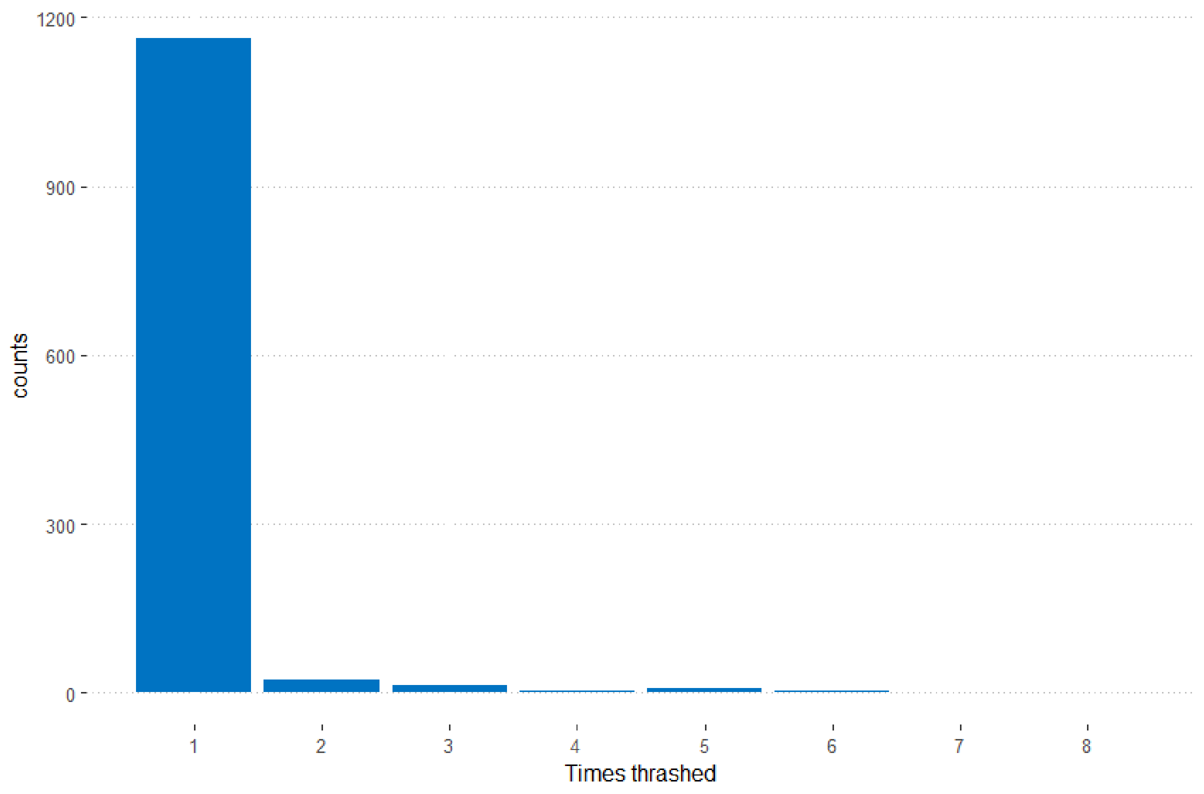


Fig. 5.42 Thrashing Histogram. The number of occurrences vs the number of times thrashed.

2*ms* latency tolerance experiment, late collisions do not exceed 20%, which is assumed to be accounted for by false-positives. The aura calculation predicts that late collisions should start occurring once 3*ms* has been exceeded, however, this is not observed, as the proportion does not increase above what would be expected for false-positives. In the 8*ms* latency tolerance experiment, late collisions do not exceed 2%, which is assumed to be accounted for by false-positives. This remains the case even when the tolerance is exceeded. The aura calculation predicts that late collisions will not begin to occur until 20*ms* latency is reached, which this experiment does not reach, therefore no late collisions outside of false-positives are expected. In the 16*ms* latency tolerance experiment, late collisions do not exceed 3% below the tolerance, which is assumed to be accounted for by false-positives. The aura calculation predicts that late collisions will not begin to occur until 20*ms* latency is reached. In Fig. 5.31 we see the proportion of errors increase from 20*ms* and above, however, it does not exceed the expected false-positive proportion until 25*ms* latency is reached.

As latency increases standard deviation remains constant below the tolerance value, as shown in Figs. 5.32, 5.33 and 5.34. With only a small number of exceptions (in 2*ms* experiment), the mean +2 standard deviations does not exceed the maximum expected penetration when below the latency tolerance. Above the latency tolerance value, the standard deviation increases for the 16*ms* tolerance experiment, but remains constant for the first two latency experiments. In addition, the first two experiments do not show a clear increase in mean penetration, this is due to the aura calculation's discrete nature and as 15*ms* has been used for the frame-time tolerance, small increases in latency of 2 and 8*ms* will not show the increase in error to the next discrete step. However, in the final experiment using 16*ms*, the relationship of increasing penetration as latency increases is demonstrated.

The 2*ms* latency tolerance experiment exhibits more false-positives than the other experiments, including across the entire range of values used. This may be a result of small aura sizes producing more false-positive, the sample size being to small or another unknown effect which has not been accounted for. In addition, although late collisions increase as actual latency exceeds 20*ms* in the 15*ms* tolerance experiment, the mean +2 standard deviations does not exceed the maximum expected penetration line until 26*ms* actual latency is reached. Both of these unexpected results should be investigated further in the future.

Outside of cloud-server to cloud-server communication latency is prone to "jitter", meaning tolerances are likely to be sometime exceeded, however, "jitter" is for short durations and would need to take place at the same time as an object migration. However, cloud-server to cloud-server communication has negligible "jitter" [83]. If collision correctness is required even in cases of "jitter", the latency tolerance would need to be higher than the peaks in latency.

As frame-time increases, the proportion of late collisions to correct collisions increases, as shown in Figs. 5.29 and 5.30, however in 5.31 the sample size is small above the tolerance line due to the high occurrences of thrashing and so the relationship in this experiment can not be deduced. For the 8*ms* and 15*ms* experiments the late collisions do not exceed 10% and are assumed to be accounted for by false-positives. In the 8*ms* frame-time tolerance experiment, late collisions are not expected to occur until 16*ms* frame-time, based on the aura size. Late collisions do not exceed 20% and is assumed to be accounted for by false-positives. In the 15*ms* frame-time tolerance experiment, late collisions exceed 20% once 16*ms* latency is exceeded as predicted by the aura size and increase to over 75% at 30*ms* latency. In the final frame-time tolerance experiment using 32*ms* frame-time tolerance, late collisions only occur for frame-times of 64*ms*. The aura calculation predicts that late collisions should begin to occur as soon as 32*ms* frame-time is exceeded, however, due to the low sample size, as a result of the high occurrences of thrashing, the plot does not accurately represent any relationship between frame-time and proportion of late collisions.

For the first two frame-time experiments (8.33*ms* and 15*ms* tolerance values), as frame-time increases, the mean penetration time increases and standard deviation also increases, shown in Figs. 5.39 and 5.40. Standard deviation increases as the delay between frame-time update and physics update can always vary between 0 and the frame-time, meaning even in high frame-time situations, the delay between frame-time update and physics update can be 0. Exceeding the frame-time tolerance very quickly leads to high max error. This is due to the delay in messages being exchanged being created by the long frame-times on both servers (this is reflected in the aura calculation by the frame-time tolerance being multiplied by 2). However, as the target frame-time increases over the tolerance, variation in error increases and maximum errors become less likely. For the final frame-time experiment, (33.33*ms* tolerance value), thrashing occurs the majority of the time (as demonstrated in 5.1 and 5.2), both above and below the tolerance, due to this the results shown in 5.41 have a very small sample size and thus will not accurately represent the relationship between frame-time and penetration.

In summary, the varying factors experiments demonstrate that the aura calculation is conclusively correct for speed and frame-time aura tolerance factors and at minimum approximate for latency. For each varied factor, for all three tolerance values chosen, below the tolerance value, the mean + two standard deviations lies under the maximum penetration time expected on a centralised simulation, with only a small number of exceptions discussed above. These experiments also demonstrate the relationship between increasing the tolerance values and amount of thrashing expected to occur. There are no missed collisions, with the exception of the speed experiments and these have significantly fewer than 1% misses below the tolerance and 1.12% misses above the tolerance. Once factors increase above their respective tolerance values, late collisions begin to occur and the more the factors are exceeded, the number and magnitude of errors increase. This demonstrates the correctness of the aura calculation in AP.

5.2.5 Experiment - Varying Packet-Loss

In addition to varying each aura tolerance factor, an experiment was undertaken to determine the effects of packet-loss on the correctness of collisions between objects. In this experiment the packet-loss was varied from 0% to 20% in increments of 1%. The aura calculation does not account for packet-loss and and therefore there is no aura tolerance. Packet-loss is controlled using the Traffic Control tool; it is simulated on each server for all communication received from the other server.

RakNet, the library used for network communication in this project, includes a reliability layer, which uses best effort. This means if packets are lost and an acknowledgement message is not received, sending will be re-attempted, but message delivery cannot be guaranteed. However, this takes time and therefore causes a delay in messages being received. Delays in messages will lead to erroneous collisions, therefore as packet-loss increases, erroneous collisions should increase in number and magnitude. If messages are delayed enough, the collision will be missed as the two objects pass each other on different servers without ever interacting with each other.

For the varying packet-loss experiments, the following parameters were used and output given. The two servers use the following tolerances: a maximum speed tolerance of $32m{\cdot}s^{-1}$; maximum latency tolerance of $2ms$; and a maximum frame-time tolerance of $15ms$. Each server has a random delay before starting between 0 and the target frame-time in order to prevent the two servers always having synchronous update cycles. Each experiment is repeated 50 times. The mean and $+/-2$ standard deviations of penetration time were plotted against each tolerance factor for each of the three tolerance values chosen. In addition, the ratio of errors to correct collisions and misses to total collisions were plotted against packet-loss.

As packet-loss increases, late and erroneous collisions increase and late collisions increase in magnitude. This is due to messages being delayed as packets are lost and need to be re-sent by RakNet's reliability layer. Missed collisions begin to occur as soon as packet-loss exceeds 1%, but increase at a lower rate relative to late collisions. (Note, late collisions begin to occur from 0%, this is expected as all factors are set to be equal to their tolerances). These results demonstrate that AP is able to prevent 50% of late collisions even in cases of packet-loss as

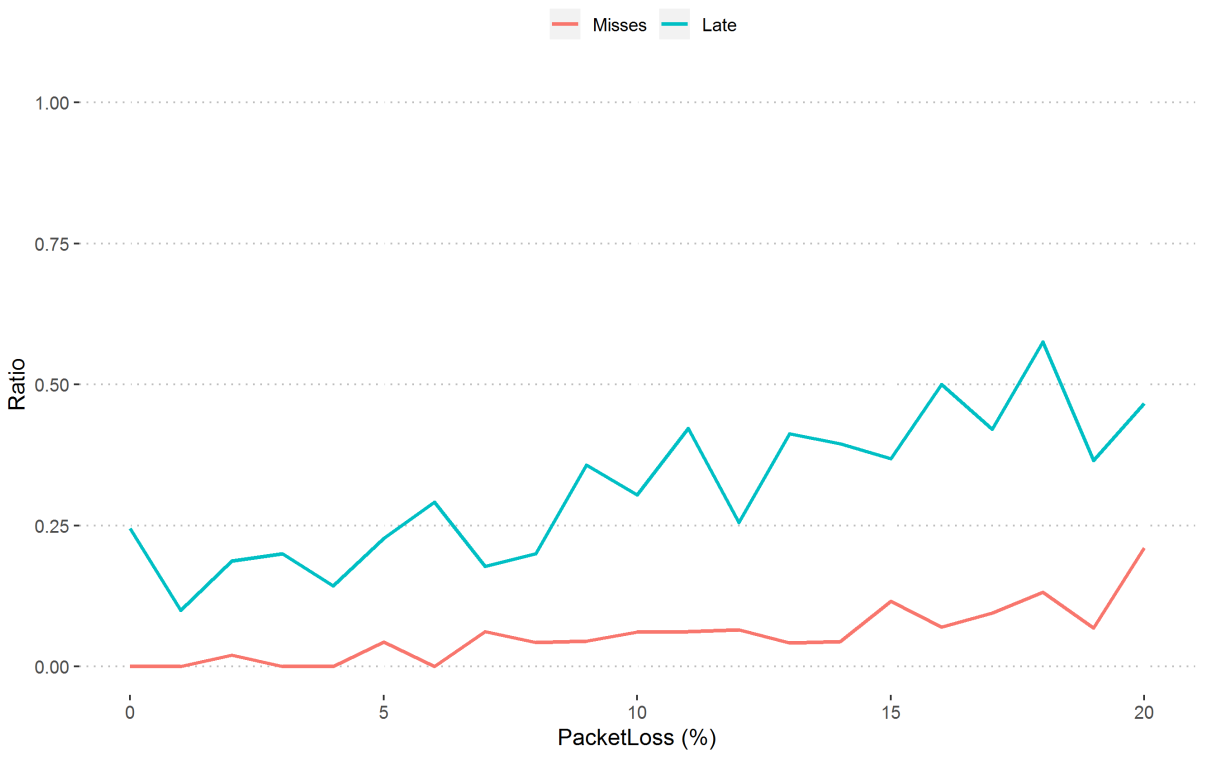


Fig. 5.43 The ratios of erroneous collisions to correct collisions and missed collisions to total collisions with varying packet loss.

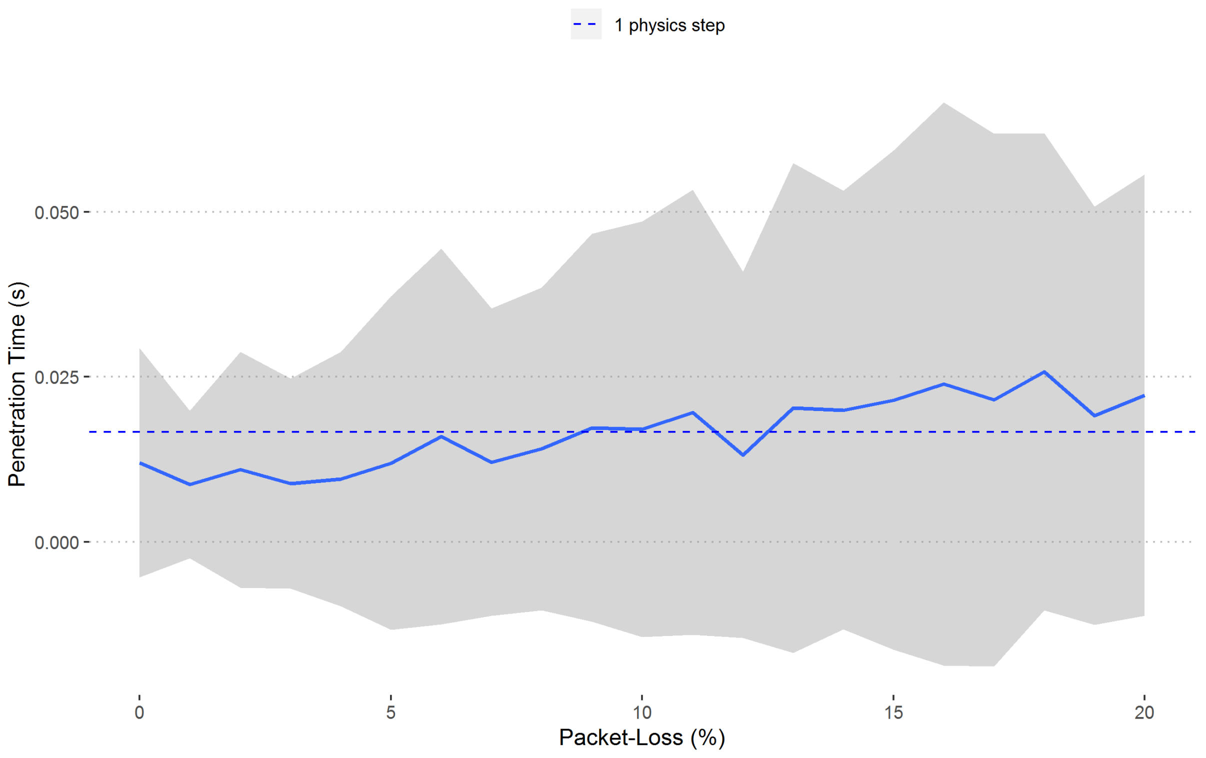


Fig. 5.44 Mean penetration time of objects with $+/-2$ standard deviations with varying packet-loss. The maximum expected penetration time of 1 physics steps is marked with a dashed blue line.

high as (15%) and prevent all missed collisions with up to 1% packet-loss, which is much greater packet-loss than would be expected in practice (0.01%) [83].

5.3 Results Summary

In this chapter, the design for the experiments for testing both the scalability and correctness of AP were presented and justified. The results of these experiments were analysed and evaluated.

The results from the experiments demonstrate the contribution of this study:

- AP is scalable, proving real-time physics simulations can be scaled.
- AP provides at minimum, an approximate solution to maintaining correctness of real-time physics when scaled.

The results from the scalability experiments demonstrate that AP is scalable, as the addition of more servers improves the overall performance of the system (measured using the maximum frame-time of all servers). This holds true even in the corner cases, including when the communication requirements between servers are at the highest, when using 9 servers in a grid layout.

The results from the collision correctness experiments demonstrate that AP provides at minimum, an approximate solution to ensuring the correctness of collisions across server boundaries when AP is employed. Collisions were shown to be correct in the majority of cases. False-positives were expected in the results due to the limitations of the experiments. The correctness was measured using the 'penetration time' of two objects colliding, one of which was intersecting the server-region boundary. The three factors that influence the aura calculation (speed, latency & frame-time) were independently tested using different values for each (based on values used by commercial games). Thrashing results, which made up a small portion of the results (except in the final frame-time experiment) were excluded and it assumed these errors will be fixed in the future.

Additionally, increasing packet-loss was also tested, which also demonstrated that collisions remain correct in the majority of cases even when exceeding values that would be expected in practice.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

An approach to networked real-time physics simulations that is scalable and alleviates the processing limitation of a single server has been presented. Only open-source software has been used in this study's approach and the algorithms used in AP have been developed in a way that is agnostic to any specific application technology.

In Chapter 3 the challenges facing distributed real-time physics were defined, specifically why maintaining consistency is necessary in real-time physics and the difficulties of maintaining consistency when latency is present. Possible solutions were addressed and this study's proposed solution, AP, was presented and justified. AP provides eventual consistency for the states of objects through the use of auras and strong consistency for collisions between objects, by ensuring any two objects that are possibly colliding are simulated on the same server. The origination of the aura calculation used for the auras in AP was also presented. The implementation of AP was discussed in Chapter 4, including potential topologies that could be used by a system using AP and the breakdown of the components of each server. Chapter 4 also gave an overview of the visualiser used by the system, which is responsible for rendering the state of the simulation through the use of replicas. Its features and their implementation details were discussed.

In Chapter 5, the design and results of the experiments were presented for addressing the questions set out by this study:

- "Can scalable real-time physics simulations be achieved?"
- "Can real-time physics simulations remain correct when scaled?"

The experiments carried out in this study establish that the approach is scalable, as demonstrated by the addition of servers improving the performance of the system when simulating an increasingly large number of objects. This study has demonstrated that a standard real-time physics engine (in this case, PhysX) may be incorporated into this study's scalable real-time physics system and achieve performance that is acceptable for real-time distributed simulations such as networked games.

The overhead of boundary objects graph (Fig. 5.13) contains a lot of noise and the results are close to the error margin of the fitted line (0.5ms for the number of objects involved). However, it appears as though there is a positive trend between increasing nodes and increasing overhead, which converges at 8 servers. It is also not clear what the exact relationship is between boundary objects and frame-time as the data for 3 and more servers are indicative of a linear relationship. However, the data for 2 servers suggests an exponential relationship based only on the single data point at 2500 objects and this may be a result of noisy data. More data is required to draw a more concrete conclusion, such as a greater number of iterations or experiments that have greater numbers of objects per server when more servers are used.

The optimal scale of the system was explored, and it can be concluded in the case of these performance experiments, going up to 10,000 objects, using 8 servers is the optimal amount. 8 servers results in a performance increase by a factor of 4.22 (see Fig. 5.17). Increasing servers beyond 8 results in minimal return and the coefficients suggest performance starts to reduce slightly, which may be a result of both overheads and reduced object efficiency (see Fig. 5.14). However, further work should be carried out to determine how the system performs with more than 10,000 objects and with greater numbers of servers.

The correctness of collisions in AP have been measured and evaluated. The experiments conducted in this study establish that AP provides at minimum, an approximate solution to maintaining correct collisions when the simulation is scaled, even when objects interact while intersecting the boundary. This has been shown to be true while increasing each of the following factors: speed; latency and frame-time. Collisions remain correct in the majority of cases up to the tolerance values for each factor and begin to show errors as soon as the maximum tolerance provided by the given aura size is exceeded, demonstrating the accuracy of the aura calculation used in AP. The holds true if thrashing collisions are excluded from the results. The number of thrashing collisions and amount of times thrashed were both measured, potential solutions to the thrashing problem were proposed in 3.7 and it can be assumed that these errors will be fixed in future work.

In addition to the aura factors being tested, the effects of packet loss on the system have also been explored and the limits of AP with increasing packet loss have been shown. This research demonstrates that collisions of rigid-bodies can be handled correctly in a scalable, distributed real-time physics system, even when those collisions take place on or near a server-region boundary.

6.2 Future Work

Work on AP will continue to be carried out, including experiments to determine the effects that changing each of the aura tolerances has on the performance and scalability of AP. Future work will enable each tolerance value to be dynamic and the aura size could adapt with the changing conditions of the system. This would improve performance when factors are low and maintain correctness when factors are high.

Acceleration could also be factored into the aura calculation. The velocity between time steps is often limited and using an acceleration tolerance would allow for smaller auras for slow moving objects. Smaller auras lead to better performance as it leads to fewer aura broadcasts and object migrations. However, the acceleration tolerance could only apply to the displacement calculation of the local object, as the velocity of a potential remote object is unknown.

An alternative to just the aura position and radius being sent over the network would be to send more details of the object projecting the aura. This could include bounding sphere and velocity and would allow the receiving server to determine the size of the aura. Under changing conditions, such as changes in latency the receiving server could dynamically change the size of the auras without the need for the sending server to send the new sizes of the auras.

The implementation of AP could be improved through the separation of the main server update rate and the network tick rate (network update frequency) as discussed in 2.3.8. These two update rates are often decoupled in video games, as it enables the server to maintain an accurate simulation while allowing for a lower tick rate to reduce the network requirements i.e. bandwidth. A lower frequency tick rate would mean a larger aura is required as there would be a longer delay between aura creation/update/collisions events and messages being sent and a longer delay between a message being received and processed. Larger auras would likely lead to a drop in performance. However, a lower frequency tick rate would mean lower processing overhead for sending, receiving and processing network messages. This trade-off could be investigated to determine how it affects performance of the overall system.

Scalability could be improved through the use of load-balancing and run-time elastic resources. Server regions could use dynamic boundaries to balance the workload of the simulation between servers. This could make use of existing research into graph-partitioning, where objects are treated as edges and overlapping auras are undirected edges. Boundaries could be moved to reduce the number of boundary-edge intersections. In addition, statistical modelling and analysis could be employed to predict future states of the simulation in order to reduce boundary-edge intersections that will occur. Through the use of elastic resources, it would also be possible to 'spin-up' and 'wind-down' nodes as required by the simulation. Combined with dynamic boundaries between servers, the simulation workload could be moved to newly created servers, by dividing existing server regions when computational requirements are high. When they are low, the regions from multiple servers could be consolidated into a single server, reducing the number of servers required, resulting in lower financial cost.

The use of containers, such as Docker, for the deployment of AP could also be investigated and compared with the current implementation that uses virtual machine cloud instances. Containers would allow for efficient provisioning and management of compute resources within the system as they enable more precise control over resources allocated. In addition, containers provide faster on-demanding provisioning of resources. Using containers instead of cloud instances may require additional overhead in terms of performance, which could be investigated and the trade-off between cloud instances and the benefits of containers explored.

Future cloud deployment architectures could also be investigated. The use of 'overseer' nodes to enable scalability in terms of clients connected could be developed and measured, and their ability to provide a level of fault-tolerance in the event of node failure could be tested. Overseer nodes could also be located at the edge, providing lower latencies for connecting clients, but at the cost of a higher latency between overseer nodes and the cloud-based simulation servers and the other overseer nodes.

AP works on the principle that any two bodies that may be interacting are always being simulated on the same server. An alternative technique to this would be Cross-Boundary Interaction, as discussed in 3.3, allowing for bodies to interact across region boundaries i.e. objects that are being simulated on two different servers interacting over the network. There are likely various advantages and disadvantages of Cross-Boundary Interaction compared to Aura Projection, which could be explored and evaluated. The two solutions may have better performance in different situations (such as a large cluster on either side of a boundary) and a solution which uses both techniques under different circumstances could be developed.

An important feature of a physics engine is the ability to query the state of the simulation. A typical example of this would be a ray-cast, in which a ray is cast between two points in the simulation space to find intersecting bodies. Distributed physics adds an extra level of complexity to this as the query may concern spaces and/or objects being simulated on different servers. In the case of spatially-partitioned solutions, the query could overlap the region boundary and in order for the query to be completed, a round-trip of messages between the two servers would be required. If the query is blocking/synchronous (execution cannot continue until the query is complete), this would introduce a large amount of wait time (at least double the network latency). Another option would be to use asynchronous queries, the results of which are returned using a function callback as soon as the results have been received. However this is more complex than a query on a centralised simulation which can return the results synchronously without any network delay.

Experiments in this study have focused on performance and correctness of AP. Further experiments could be carried out to measure bandwidth requirements of AP in different scenarios as well as the effect of reduced bandwidth on both performance and correctness.

More scenarios could be tested for performance. This study has tested the performance of AP with a 0.5 ratio of objects near and far from the boundary. Performance is expected to be best when all objects are far from the boundary as almost no message exchange is required between servers, likewise performance is expected to be worst when all objects are near to the boundary and are both 'awake' and moving, as this requires the most amount of message exchange and subsequently the highest demand of both network resources and computational overhead from the algorithms used in AP. Investigating different ratios of objects near and far from the boundary, different numbers of moving and 'sleeping' objects, or simply an increasing number of objects near the boundary, would determine at what point and under what circumstances AP becomes no longer scalable.

AP is currently limited to the migration of single rigid-bodies between servers. Support for other physical entities such as groups of attached rigid-bodies, soft-bodies and constraints within AP could be added and investigated.

References

[1] Aggarwal, S., Banavar, H., Khandelwal, A., Mukherjee, S., and Rangarajan, S. (2004). Accuracy in dead-reckoning based distributed multi-player games. In *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '04, page 161–165, New York, NY, USA. Association for Computing Machinery.

[2] Aggarwal, S., Banavar, H., Mukherjee, S., and Rangarajan, S. (2005). Fairness in dead-reckoning based distributed multi-player games. In *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '05, page 1–10, New York, NY, USA. Association for Computing Machinery.

[3] Allard, J. and Raffin, B. (2006). Distributed physical based simulations for large vr applications. In *Virtual Reality Conference, 2006*, pages 89–96. IEEE.

[4] Beigbeder, T., Coughlan, R., Lusher, C., Plunkett, J., Agu, E., and 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, page 144–151, New York, NY, USA. Association for Computing Machinery.

[5] Bernier, Y. W. (2001). Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*.

[6] Bezerra, C. E., Cecin, F. R., and Geyer, C. F. R. (2008). A3: A novel interest management algorithm for distributed simulations of mmogs. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT '08, pages 35–42, Washington, DC, USA. IEEE Computer Society.

[7] Bharambe, A. R., Padmanabhan, V. N., and Seshan, S. (2004). Supporting spectators in online multiplayer games. In *Proceedings of ACM SIGCOMM Workshop on Hot Topic in Networks*.

[8] Bharambe, A. R., Pang, J., and Seshan, S. (2006). Colyseus: A distributed architecture for online multiplayer games. In *NSDI*, volume 6, pages 12–12.

[9] Blizzard (2019). World of warcraft. https://worldofwarcraft.com/en-us/. (Accessed on 22/02/2019).

[10] Boeing, A. and Bräunl, T. (2007). Evaluation of real-time physics simulation systems. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, GRAPHITE '07, page 281–288, New York, NY, USA. Association for Computing Machinery.

[11] Bossa Studios (2019). Worlds adrift. https://www.worldsadrift.com/. (Accessed on 22/02/2019).

[12] Brown, A., Ushaw, G., and Morgan, G. (2019). Aura projection for scalable real-time physics. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, page 1. ACM.

[13] Cai, W., Shea, R., Huang, C., Chen, K., Liu, J., Leung, V. C. M., and Hsu, C. (2016). A survey on cloud gaming: Future of computer games. *IEEE Access*, 4:7605–7620.

[14] Che, X. and Ip, B. (2012). Packet-level traffic analysis of online games from the genre characteristics perspective. *Journal of Network and Computer Applications*, 35(1):240 – 252. Collaborative Computing and Applications.

[15] Chen, K.-T., Chang, Y.-C., Hsu, H.-J., Chen, D.-Y., Huang, C.-Y., and Hsu, C.-H. (2013). On the quality of service of cloud gaming systems. *IEEE Transactions on Multimedia*, 16(2):480–495.

[16] Chen, K.-T., Chang, Y.-C., Tseng, P.-H., Huang, C.-Y., and Lei, C.-L. (2011a). Measuring the latency of cloud gaming systems. In *Proceedings of the 19th ACM International Conference on Multimedia*, MM '11, page 1269–1272, New York, NY, USA. Association for Computing Machinery.

[17] Chen, K.-T., Chang, Y.-C., Tseng, P.-H., Huang, C.-Y., and Lei, C.-L. (2011b). Measuring the latency of cloud gaming systems. In *Proceedings of the 19th ACM international conference on Multimedia*, pages 1269–1272.

[18] Choy, S., Wong, B., Simon, G., and Rosenberg, C. (2012). The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6.

[19] Claypool, M. and Claypool, K. (2006). Latency and player actions in online games. *Commun. ACM*, 49(11):40–45.

[20] Claypool, M. and Claypool, K. (2010). Latency can kill: precision and deadline in online games. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 215–222.

[21] Coulouris, G. F., Dollimore, J., and Kindberg, T. (2005). *Distributed systems: concepts and design*. pearson education.

[22] D'Amora, B., Nanda, A., Magerlein, K., Binstock, A., and Yee, B. (2006). High-performance server systems and the next generation of online games. *IBM Systems Journal*, 45(1):103–118.

[23] de Senna Carneiro, T. G. and Arabe, J. N. C. (1998). Load balancing for distributed virtual reality systems. In *Computer Graphics, Image Processing, and Vision, 1998. Proceedings. SIBGRAPI '98. International Symposium on*, pages 158–165.

[24] Dick, M., Wellnitz, O., and Wolf, L. (2005). Analysis of factors affecting players' performance and perception in multiplayer games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–7.

[25] Dong, L. and Yue-Long, Z. (2013). An overlapping architecture for roia in cloud. In *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*, pages 61–65.

[26] D'Angelo, G., Ferretti, S., and Marzolla, M. (2015). Cloud for gaming. *Encyclopedia of Computer Graphics and Games*, page 1–6.

[27] Epic Games (2020). Unreal engine. https://www.unrealengine.com/en-US/. (Accessed on 29/02/2020).

[28] Ferretti, S. and D'Angelo, G. (2018). *Client/Server Gaming Architectures*, pages 1–2. Springer International Publishing, Cham.

[29] Foster, I., Zhao, Y., Raicu, I., and Lu, S. (2008). Cloud computing and grid computing 360-degree compared. In *2008 Grid Computing Environments Workshop*, pages 1–10.

[30] Fujimoto, R. M. (2000). *Parallel and distributed simulation systems*, chapter 7, pages 195–221. Citeseer.

[31] GameNow (2020). Gamenow. https://www.ugamenow.com/. (Accessed on 01/03/2020).

[32] García-Valls, M., Cucinotta, T., and Lu, C. (2014). Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9):726 – 740.

[33] Gautier, L. and Diot, C. (1998). Design and evaluation of mimaze a multi-player game on the internet. In *Proceedings. IEEE International Conference on Multimedia Computing and Systems (Cat. No.98TB100241)*, pages 233–236.

[34] Google (2020). Stadia. https://store.google.com/product/stadia_learn/. (Accessed on 04/02/2020).

[35] Greenhalgh, C. and Benford, S. (1995). Massive: a distributed virtual reality system incorporating spatial trading. In *Proceedings of 15th International Conference on Distributed Computing Systems*, pages 27–34.

[36] Gupta, N., Demers, A., Gehrke, J., Unterbrunner, P., and White, W. (2009). Scalability for virtual worlds. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1311–1314.

[37] Hadean (2020). Features. https://www.hadean.com/aether-engine/features/. (Accessed on 14/01/2020).

[38] Hori, M., Iseri, T., Fujikawa, K., Shimojo, S., and Miyahara, H. (2001). Scalability issues of dynamic space management for multiple-server networked virtual environments. In *2001 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (IEEE Cat. No.01CH37233)*, volume 1, pages 200–203 vol.1.

[39] Hu, S.-Y., Chen, J.-F., and Chen, T.-H. (2006). Von: a scalable peer-to-peer network for virtual environments. *IEEE Network*, 20(4):22–31.

[40] Improbable (2018). Introducing the upgraded runtime: the new heart of spatialos – improbable. https://improbable.io/blog/new-spatialos-runtime-v2. (Accessed on 08/15/2020).

[41] Improbable (2019). SpatialOS. https://spatialos.improbable.io/. (Accessed on 22/02/2019).

[42] Jarschel, M., Schlosser, D., Scheuring, S., and Hoßfeld, T. (2011). An evaluation of qoe in cloud gaming based on subjective tests. In *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 330–335. IEEE.

[43] Korhonen, J. and Wang, Y. (2005). Effect of packet size on loss rate and delay in wireless links. In *IEEE Wireless Communications and Networking Conference, 2005*, volume 3, pages 1608–1613 Vol. 3.

[44] Koszela, J. and Szymczyk, M. (2018). Distributed processing in virtual simulation using cloud computing environment. In *MATEC Web of Conferences*, volume 210, page 04018. EDP Sciences.

[45] Lee, W.-K. and Chang, R. K. (2015). Evaluation of lag-related configurations in first-person shooter games. In *2015 International Workshop on Network and Systems Support for Games (NetGames)*, pages 1–3. IEEE.

[46] Lee, Y.-T., Chen, K.-T., Su, H.-I., and Lei, C.-L. (2012). Are all games equally cloud-gaming-friendly? an electromyographic approach. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6. IEEE.

[47] Lu, F., Parkin, S., and Morgan, G. (2006). Load balancing for massively multiplayer online games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, New York, NY, USA. ACM.

[48] Lu, X. and Guan, H. (2017). *Visualization for Earthquake Disaster Simulation of Urban Buildings*, pages 303–326. Springer Singapore, Singapore.

[49] Macedonia, M. R., Zyda, M. J., Pratt, D. R., Barham, P. T., and Zeswitz, S. (1994). Npsnet:a network software architecture for largescale virtual environments. *Presence: Teleoperators and Virtual Environments*, 3(4):265–287.

[50] Mashayekhi, O., Shah, C., Qu, H., Lim, A., and Levis, P. (2018). Automatically distributing eulerian and hybrid fluid simulations in the cloud. *ACM Transactions on Graphics (TOG)*, 37(2):24.

[51] Mauve, M., Vogel, J., Hilt, V., and Effelsberg, W. (2004). Local-lag and timewarp: Providing consistency for replicated continuous applications. *Trans. Multi.*, 6(1):47–57.

[52] Mazlami, G., Cito, J., and Leitner, P. (2017). Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531.

[53] McLoone, S. C., Walsh, P. J., and Ward, T. E. (2012). An enhanced dead reckoning model for physics-aware multiplayer computer games. In *2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*, pages 111–117.

[54] Mehta, M. R., Bhatt, M. Y., Joshi, M. Y., and Vidhani, M. S. (2015). Animated movie making using a game engine. *Int. J. Emerg. Trends Sci. Technol.*, 2:2320–2324.

[55] Metzger, F., Rafetseder, A., and Schwartz, C. (2016). A comprehensive end-to-end lag model for online and cloud video gaming. *5th ISCA/DEGA Work. Percept. Qual. Syst.(PQS 2016)*, pages 15–19.

[56] Microsoft (2020). Project xcloud. https://www.xbox.com/en-GB/xbox-game-streaming/project-xcloud/. (Accessed on 01/03/2020).

[57] Millington, I. (2007). *Game physics engine development*. CRC Press.

[58] Min, D., Choi, E., Lee, D., and Park, B. (1999). A load balancing algorithm for a distributed multimedia game server architecture. In *Proceedings IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 882–886 vol.2.

[59] Moghe, U., Lakkadwala, P., and Mishra, D. K. (2012). Cloud computing: Survey of different utilization techniques. In *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, pages 1–4.

[60] Moll, P., Lux, M., Theuermann, S., and Hellwagner, H. (2018). A network traffic and player movement model to improve networking for competitive online games. In *2018 16th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6.

[61] Morgan, G., Lu, F., and Storey, K. (2005). Interest management middleware for networked games. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 57–64, New York, NY, USA. ACM.

[62] Morgan, G. and Storey, K. (2005). Scalable collision detection for massively multiplayer online games. In *19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers)*, volume 1, pages 873–878 vol.1.

[63] Negrãlo, A. P., Adaixo, M., Veiga, L., and Ferreira, P. (2014). On-demand resource allocation middleware for massively multiplayer online games. In *2014 IEEE 13th International Symposium on Network Computing and Applications*, pages 71–74.

[64] NINPO (2019). Vanishing stars: Colony wars by ninpo. http://www.vanishingstars.com/. (Accessed on 22/02/2019).

[65] Nvidia (2019a). Cloud gaming - gaming as a service (gaas) | grid|nvidia. http://www.nvidia.com/object/cloud-gaming.html. (Accessed on 22/02/2019).

[66] Nvidia (2019b). Physx knowledge base/faq | Nvidia UK. https://www.nvidia.com/object/physx_knowledge_base.html. (Accessed on 22/02/2019).

[67] Nvidia (2020a). Geforce now. https://www.ugamenow.com/. (Accessed on 01/03/2020).

[68] Nvidia (2020b). Gpu rigid bodies — nvidia physx sdk 3.4.0 documentation. https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/GPURigidBodies.html. (Accessed on 20/03/2020).

[69] Nvidia (2020c). Nvidia holodeck. https://www.nvidia.com/en-gb/design-visualization/technologies/holodeck/. (Accessed on 28/02/2020).

[70] OnLive (2020). Onlive. http://onlive.com/. (Accessed on 01/03/2020).

[71] Pantel, L. and Wolf, L. C. (2002). On the suitability of dead reckoning schemes for games. In *Proceedings of the 1st Workshop on Network and System Support for Games*, NetGames '02, page 79–84, New York, NY, USA. Association for Computing Machinery.

[72] Perkins, C., Hodson, O., and Hardman, V. (1998). A survey of packet loss recovery techniques for streaming audio. *IEEE Network*, 12(5):40–48.

[73] Pisan, Y. (2004). Challenges for network computer games. In *ICWI*, pages 589–595.

[74] RakNet (2020a). Reliability types. http://www.jenkinssoftware.com/raknet/manual/reliabilitytypes.html. (Accessed on 24/02/2020).

[75] RakNet (2020b). Replica manager. http://www.raknet.net/raknet/manual/replicamanager3.html. (Accessed on 25/02/2020).

[76] Savery, C. (2014). *Consistency maintenance in networked games*. PhD thesis, Queen's University (Kingston, Ont.).

[77] Shah, S., Dey, D., Lovett, C., and Kapoor, A. (2018). Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In Hutter, M. and Siegwart, R., editors, *Field and Service Robotics: Results of the 11th International Conference*, pages 621–635, Cham. Springer International Publishing.

[78] Sharkey, P. M., Ryan, M. D., and Roberts, D. J. (1998). A local perception filter for distributed virtual environments. In *Proceedings. IEEE 1998 Virtual Reality Annual International Symposium (Cat. No.98CB36180)*, pages 242–249.

[79] Shea, R., Liu, J., Ngai, E. C. ., and Cui, Y. (2013). Cloud gaming: architecture and performance. *IEEE Network*, 27(4):16–21.

[80] Sony (2019). Playstation now. https://www.playstation.com/en-gb/explore/playstation-now/. (Accessed on 22/02/2019).

[81] Storey, K., Lu, F., and Morgan, G. (2004). Determining collisions between moving spheres for distributed virtual environments. In *Computer Graphics International, 2004. Proceedings*, pages 140–147. IEEE.

[82] Tasora, A., Serban, R., Mazhar, H., Pazouki, A., Melanz, D., Fleischmann, J., Taylor, M., Sugiyama, H., and Negrut, D. (2016). Chrono: An open source multi-physics dynamics engine. In Kozubek, T., Blaheta, R., Šístek, J., Rozložník, M., and Čermák, M., editors, *High Performance Computing in Science and Engineering*, pages 19–49, Cham. Springer International Publishing.

[83] ThousandEyes (2018). 2018 public cloud performance benchmark report. https://www.thousandeyes.com/resources/2018-public-cloud-performance-benchmark-report/. (Accessed on 11/12/2019).

[84] ThousandEyes (2019). Cloud performance benchmark 2019-2020 edition. https://www.thousandeyes.com/resources/cloud-performance-benchmark-report-november-2019/. (Accessed on 11/12/2019).

[85] Turchini, G., Monnet, S., and Marin, O. (2015). Scalability and availability for massively multiplayer online games. In Gashi, I. and Busnel, Y., editors, *11th European Dependable Computing Conference (EDCC 2015)*, Proceedings of Fast Abstract - EDCC 2015, Paris, France.

[86] Unity Technologies (2020). Unity engine. https://unity.com/. (Accessed on 29/02/2020).

[87] Vähä, M. (2017). *Applying microservice architecture pattern to a design of an MMORPG backend*. PhD thesis, University of Oulu.

[88] Valve (2020). Source multiplayer networking - valve developer community. https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking. (Accessed on 20/02/2020).

[89] Wu, D., Xue, Z., and He, J. (2014). icloudaccess: Cost-effective streaming of video games from the cloud with low latency. *IEEE Transactions on Circuits and Systems for Video Technology*, 24(8):1405–1416.

[90] Xu, J., Tang, Z., Wei, X., Nie, Y., Yuan, X., Ma, Z., and Zhang, J. J. (2017). *A VR-Based Crane Training System for Railway Accident Rescues*, pages 207–219. Springer International Publishing, Cham.

[91] Yahyavi, A. and Kemme, B. (2013). Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Comput. Surv.*, 46(1):9:1–9:51.

[92] Yates, R. D., Tavan, M., Hu, Y., and Raychaudhuri, D. (2017). Timely cloud gaming. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9.