VAST: a scalable spatial publish and subscribe system integrated with Minecraft

by

Miguel Smith

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Research) in the Faculty of Engineering at Stellenbosch University

Study leader: Prof. H.A. Engelbrecht
Prof. S.Y. Hu

March 2020

Stellenbosch University https://scholar.sun.ac.za

BYLAE 1

UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Plagiaatverklaring / Plagiarism *Declaration*

1 Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.

2 Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
I agree that plagiarism is a punishable offence because it constitutes theft.

3 Ek verstaan ook dat direkte vertalings plagiaat is.
I also understand that direct translations are plagiarism.

4 Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.

5 Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

Copyright © 2020 Stellenbosch University
All rights reserved

Stellenbosch University https://scholar.sun.ac.za

Abstract

With the world becoming more connected by the day, the need for modern massively multi-user virtual environments (MMVEs) to be capable of hosting an increasing number of players is vital. Unless large sums of money are spent, current system architectures limit how many players can join a single system due to bandwidth and computational strains that arise from interactions between players and the virtual world.

This thesis proposes using VAST, a spatial publish/subscribe system, to handle the update and event dissemination of MMVEs in order to make it more scalable. VAST allows clients within the system to efficiently send and receive messages to each other in a scalable way. It achieves this with a multi-layered system that handles different aspects of the system. The Voronoi-overlay network handles the network connections of different VAST nodes to each other by using a dynamic Voronoi diagram to determine a node's neighbours. It only connects to these neighbours, allowing it to be a part of the system as a whole whilst only interacting with a part of it. This layer is shown to have scalable characteristics, handling up to 1000 nodes with response times of under 25ms. The spatial publish and subscribe system is handled by the Voronoi-spatial overlay. This layer uses a Voronoi diagram to section the virtual world into regions, each one handled by a single VAST node. Clients connect to the world and are assigned to a node depending on which region their joining location falls under. These VAST nodes are dynamic, allowing for load balancing to occur should one node become overloaded. The layer showed latencies of 3ms and a drift distance of under 0.0024 for 350 clients. An entry server handles client logins and packet routing.

Minecraft is used to test the real world applicability of VAST to an MMVE by allowing VAST to handle its update and event dissemination. The Koekepan project provides client and server proxies that allow Minecraft packets to be converted into packets that can be published to VAST. The system is shown to be able to reduce the amount of packets sent in a system containing two Minecraft clients by 33.52% when using a spatial publish/subscribe system to disseminate updates. The system is shown to be capable of handling over 350 simulated clients on a single system without any decrease in performance.

Stellenbosch University https://scholar.sun.ac.za

Samevatting

Met die wêreld se toenemende elektroniese verbindings raak die behoefte al hoe groter vir moderne, massieve multigebruiker virtuele omgewings (MMVEs) om meer gebruikers te kan huisves. Tensy groot bedrae geld spandeer word, beperk die huidige stelselargitektuur die aantal gebruikers wat by ’n enkele stelsel kan aansluit as gevolg van beperkte bandwydte en berekeningsvereistes wat onstaan a.g.v. die interaksies tussen gebruikers en die virtuele omgewing.

Om die virtuele omgewing meer skaleerbaar te maak, word daar in hierdie tesis voorgestel dat VAST, a ruimtelike publikasie/subskripsie stelsel, gebruik word om vir die verspreiding van opdaterings- en gebeure-boodskappe. VAST stel kliënte in die stelsel in staat om hierdie boodskappe effektief aan mekaar te stuur en te ontvang op ’n skaleerbare manier. Dit word bereik deur die gebruik van ’n multivlak stelsel wat verskillende aspekte van die stelsel hanteer. Die Voronoi-netwerklaag hanteer die netwerk konneksies van verskillende VAST nodusse aan hul bure deur ’n dinamiese Voronoi diagram te gebruik om nodusse se bure te bepaal. ’n VAST nodus konnekteer slegs aan sy bure, wat dit toelaat om deel te vorm van totale stelsel terwyl dit slegs kommunikeer met ’n gedeelte van die virtuele omgewing. Ons toon dat die Voronoi-netwerklaag skaleerbaar is, deur tot en met 1000 nodusse te hanteer met responstye van minder as 25ms. Die ruimtelike publiseer/subskripsie stelsel work hanteer deur ’n aparte ruimtelike Voronoi-laag. Die ruitemlike laag gebruik ’n Voronoi-diagram om die virtuele omgewing te verdeel in verskillende streke, waar elke streek beheer word deur ’n enkele VAST nodus. Kliënte konnekteer aan die virtuele omgewing en, afhangende van die streek waarin die klient hom bevind, word dit toegewys aan die VAST nodus wat daardie streek beheer. Hierdie VAST nodusse is dinamies, wat toelaat vir balansering van die las van die VAST nodusse indien ’n nodus oorlaai word. Die ruimtelike laag toon vertragings van 3ms en ’n dryfafstand van onder 0.0024 wanneer 350 kliënte gelyktydig konnekteerd is aan die stelsel. ’n Toegangsbediener word gebruik om aantekening van kliente te hanteer asook roetering van netwerk pakkies.

Om die werklike toepaslikheid van VAST op ’n MMVE te toets word daar van Minecraft gebruik gemaak. VAST is verantwoordelik vir die verspreiding van opdaterings- en gebeure-boodskappe.Die Koekepan-projek bied instaanbedieners vir beide die Minecraft kliënte en bedieners. Hierdie instaanbedieners omskep die Minecraft-pakkies in pakkies wat deur VAST gepubliseer kan word. Daar word gewys dat die VAST stelsel die aantal Minecraft pakkies wat versprei moet word kan verminder met 33.52%, wanneer twee Minecraft-kliënte konnekteer is aan die virtuele omgewing. Daar word ook getoon dat die stelsel meer as 350 gesimuleerde Minecraft kliënte kan hanteer op ’n enkele stelsels sonder om ’n verlies in prestasie.

Stellenbosch University https://scholar.sun.ac.za

Acknowledgements

Firstly, as in all things, glory to my Father in heaven who constantly gives strength and grace to do what needs doing.

I would like to say thank you to my two supervisors, Herman and Shun-Yun. Without the weekly (and for a while daily) meetings and constant advice, ideas and life lessons I would not have finished what I needed to finish. Your wisdom and guidance will be with me for the rest of my life.

Then to my family and friends who provided constant support and encouragement, reminding me that rest is often more productive than work and that we all feel overwhelmed at times, thank you. The prayers at all hours of the day and night carried me when I did not feel like I could make it through.

Finally to my fiancé, Eraine. You saw every day of effort and never stopped giving your love, support and encouragement through it, even in the 4 months that I told you I was almost finished. Thank you for your patience.

Stellenbosch University https://scholar.sun.ac.za

Contents

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

List of Figures

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

List of Tables

Stellenbosch University https://scholar.sun.ac.za

Nomenclature

3D	Three dimensional
AE	Active entities
AI	Artificial intelligence
AoE	Area of effect
AoI	Area of interest
AOIM	Area of interest manager
API	Application protocol interface
B	bytes
BR	Battle royale
C/MS	Client/multi-server
C/S	Client/server
CPU	Central processing unit
DHT	Distributed hash tables
EID	Entity identifier
ES	Entry server
FPS	First person shooter
GB	Gigabyte
GPU	Graphical processing unit
GUI	Graphical user interface
ID	Identifier
IM	Interest management
IP	Internet protocol
JS	JavaScript
KB	Kilobytes
KBps or KB/s	Kilobytes per second
LAN	Local area network
LE	Latent entities
Mbps	Megabits per second
MB/s	Megabytes per second
MMVE	Massively Multi-user virtual environment
ms	Milliseconds
NPC	Non-playable character
NTP	Network time protocol
NVE	Networked virtual environment
P2P	Peer-to-peer
PBT	Predictive binary tree
PCM	Predictive contract mechanisms
PD	Peer density
RAM	Random access memory

Stellenbosch University https://scholar.sun.ac.za

RPG	Role-playing game
RTT	Round-trip time
RTS	Real-time strategy
SPS	Spatial publish/subscribe
TCP	Transmission control protocol
UUID	Universally unique identifier
VE	Virtual environment
VON	Voronoi overlay network
VR	Virtual reality
VSO	Voronoi self-organising overlay
WAN	Wide area network

Stellenbosch University https://scholar.sun.ac.za

Chapter 1

Introduction

The video game industry is growing faster than ever before and the majority of people who are playing are playing together. According to [8], 65% of American adults play video games and of those 65%, 63% play with friends. If this is indicative of world-wide data, then of the world's 2.6 billion gamers [9], 1.638 billion play with others. These people spend an average of 4.8 hours per week playing online with their friends. With such a wide audience of online gamers playing together, video games need to be able to support the players that want to play together with architectures that can allow all players to interact with each other in real-time. Whilst some games are supporting millions of players, only a few hundred of these are playing together in the same world and the same spaces. The reason for this is that whilst research [10] has been done to investigate solutions to scalability problems, the industry has not followed.

1.1 Massively Multi-user Virtual Environments

Massively multi-user virtual environments (MMVE) are systems that have a large simulated world in which users can control an avatar and interact with others whilst completing objectives or actions. One of the most common implementations of MMVEs is in video games. Genres such as role-playing games (RPGs), battle royale (BR) and first-person shooters (FPS) can be classified under the MMVE genre.

Every MMVE game consists of the following components:

- an avatar or viewport (controlled by the client with which the player vicariously interacts with the virtual world around it)
- this simulated world is known as a virtual environment (VE) in which the player's avatar exists
- interactive objects and entities that populate the VE and give the player ways to complete objectives and actions
- a system that both hosts the information that describes the state of everything within the MMVE and a way to disseminate updates that result from events that occur within the VE.

Stellenbosch University https://scholar.sun.ac.za

1.1.1 The client

The client is the software that the user runs to access the MMVE. It is through the client that the user can control the avatar or viewport. The avatar can be any three-dimensional (3D) model that the designer has created for the user to control and interact with the VE. The user will make use of some form of external hardware (keyboard, mouse, controller, virtual reality (VR) headset) to control the avatar's movements and therefore the user's perspective of the VE. The viewport is the name given to the window that the user sees the VE through. It is what appears on the screen when the client is run. In some games the user may not have an avatar and is instead represented by an omnipotent, omnipresent force that still interacts with the world but is not bound to a single avatar (such as in Black and White [11] or Spore [12]).

1.1.2 Virtual environments

Virtual environments are the representation of the world within which the avatar exists. They can either be static and unchangeable to the user, pre-determined by the designers of the MMVE, or dynamic and volatile with the ability to completely change over time and different for every instance of the game (such as Minecraft [13]). This world is stored on the host of the MMVE and defines the bounds of where the avatars and entities within the VE are permitted to go. These VEs can be a real-life representation of places in our world or entirely fabricated construction of the designer's imagination. These worlds can be designed to lead the user along a certain path, supporting the narrative of the game (such as dungeons and raids in World of Warcraft), or open-world which allows the user to freely explore the world and interact with it in any order and manner of their choosing.

1.1.3 Objects and entities

MMVEs would be monotonous and unexciting if it were just the VE and player avatars; objects add another element of complexity. Objects are interactive pieces within the VE that the user can utilize to perform certain actions, complete objectives or simply to make the static VE feel more like a real, living world. These interactive objects can be items that the player uses and keeps in their inventory, environmental props that have animations such as trees swaying in the wind or bushes rustling.

Entities are avatars that are controlled by algorithms and artificial intelligence (AI) rather than a user. These entities are also known as non-playable characters (NPCs) or *bots* (which is short for robots). These entities fill the world and create an atmosphere more akin to a real-world full of life rather than an empty shell of a world with merely a few user-controlled avatars within.

1.1.4 System architectures

There are a few ways that the MMVE can be hosted; the most widely used and easiest to set up is the client/server (C/S) architecture. This is where the world is hosted on a single server and the users make use of clients to connect to the server and receive information about the world. All actions performed by the user are sent to the server where these

Stellenbosch University https://scholar.sun.ac.za

changes are logged and sent out to the other users to inform them of the change. This is called the event-update cycle and is discussed in more detail below.

Another well researched but less commonly used architecture is that of peer to peer (P2P). In P2P architectures, there is no server. The clients have a dual responsibility of being both a user in the system, as well as a server for others. This theoretically allows the system to have as many users as desired, but provides design and user-experience challenges.

Finally, there is hybrid P2P and client/multi-server (C/MS). Hybrid P2P attempts to use both P2P and C/S in the design to get the best of both architectures. It stores the world on the servers whilst having clients inform each other of changes that they enact. This has benefits in terms of the stability and scalability of the MMVE, however, the potential for issues with security is still present. C/MS systems aim to keep more security and control over the system by hosting and controlling all of the information regarding objects, entities, avatars, and the VE whilst improving on C/S scalability. It does this by using multiple high-power servers instead of just one, and integrating them so that they can communicate with each other and share resources.

1.1.5 Event-update cycle

Whether there is a central authority that controls all of the states within a VE, or the states are distributed among the client machines, every entity, object and player has a global copy that describes the state that it is in at any given moment. All entities that are interested in a particular entity get their information from the global copy of such. This information is stored as a replica of the global copy and is known as the local copy. If the global copy and a local copy of a state become desynchronised for any reason, the local copy will defer to the global copy for an accurate view of the entity. Whenever an entity within the VE performs an action that affects the state of something around it, it is described as an event. This change in an object's state is communicated to entities that have a local copy of the entity's state via an update. This event-update cycle is integral to the functioning of an MMVE.

1.1.6 Requirements of an MMVE

For MMVEs to handle the millions of potential players on their system whilst still maintaining a high-quality user experience, they have to meet the requirements of five different aspects that make for a functioning and entertaining MMVE. These have been deduced from [14] and [15] and are as follows:

1. Consistency
2. Persistence
3. Interactivity and fairness
4. Scalability
5. Security

Stellenbosch University https://scholar.sun.ac.za

Consistency

Objects, entities and the world are stored on the host as a state. This is a collection of attributes and characteristics of an object that define how it appears in the world, as well as how it is interacted with by other entities. Consistency is when a global copy of an object's state and the copies of this state contain information that is different or contradictory. This desynchronisation can be caused either by latency or packet loss (the loss of a collection of information that describes an update). If a player interacts with a VE, object or entity that is not as it appears to be, the result can be a highly unsatisfactory experience, as the expected outcomes of an action will not be realised.

Persistence

Persistence is both the ability of the MMVE to be left for an indefinite amount of time and to remain as it was at such time, as well as for all the clients to be disconnected from it and then reconnect to it in the same state that the MMVE was left in. When video games were first created, there was no option to save the world state and then return to where you left off. If you started on world 1-1 in the original Mario Bros, played for a while and then turned off the system and returned later, you would have to start from world 1-1 again. This, however, has changed as MMVEs are now required to remain the same no matter how long the user is gone for, and their progress should be kept even if the player only returns after an extended period. Persistence is also the ability of the MMVE to be fault-tolerant. If a host containing global states fails or crashes, the system should be able to either recover the information or reconstruct the information from state copies.

Interactivity and fairness

Interactivity is described as the interactions between different entities within the world and the quality thereof. Within an MMVE, the interactions of players and how quickly these actions register as permanent affects the potential for the game to be immersive. This responsiveness is affected by how long it takes for events and updates to be sent and received, and varies for every genre of online game.

Fairness is the idea that every user that plays the game experiences it in the same way regardless of their machine's processing power or how quickly their messages reach the host of the MMVE. It is directly linked to interactivity; the better the interactivity of the client, the quicker it can affect the world and everything around it and the quicker they can receive the updates informing it of the preceding actions. Fairness must be a priority in MMVEs because alternatively the player experience is significantly reduced by perceived or real disadvantages relative to their peers.

Scalability

Scalability is the most difficult requirement to satisfy because when it is prioritised every other requirement has to sacrifice to be successful. Scalability is the ability of the MMVE to operate in the same way whether there is one user or millions of users. To be scalable means to be able to handle as many users as there is a demand for and still have a functioning game or system. It is not beneficial if there are millions of players but no interactivity within the game because of the high latency or interactions with the game

Stellenbosch University https://scholar.sun.ac.za

not registering due to the player's local state constantly being inconsistent and requiring updating.

It is for this reason that solving the problem of scalability is such a compelling yet elusive task. Whilst the other requirements can be solved while sacrificing merely one or two other requirements, scalability has not been achieved with the same success. Research has already been undertaken on scalability; this will be the focus of the subsequent section.

1.1.7 Scalability research

There are two ways to improve scalability: reduce the consumption of resources or increase the amount of resources within the system. A system is scalable when clients can be indefinitely added with minimal to no impact on resource consumption. Therefore having a genuinely scalable system, unless each client that joins induces a net increase in resources, requires a design that reduces the consumption of the system.

NPSNET [16] uses the idea of network multicast groups to reduce network bandwidth. This was an early introduction to the idea of sending messages to specific client groups instead of broadcasting them or sending them to individual clients. This was instrumental in further research.

SimMud [17] uses the concept of users being most interested in that which surrounds them to form self-organising groups based on their virtual locations to which the system can multi-cast messages. It successfully demonstrated this in a test of 4000 players, showing scalability whilst still maintaining consistency and ease of state distribution. It does not take into account variable latency nor security of information within the architecture.

Donnybrook [18] estimates what players will be interested in and tailors the frequency of updates sent to the client accordingly. It also uses a multicast system that is latency-sensitive and can support a high rate of change of group members. Its bandwidth requirements for the server scale quadratically for every client added to the area of interest (AoI) however, so it does not deal well with clustering/flocking.

RING [19] takes into account the visible lines of sight of clients within a "densely occluded" VE - such as a world with high buildings or mountains. It withholds entity state updates from clients who cannot perceive the changes. Experiments displayed a decrease of 40 times the number of messages processed by clients in a 1024-client test. This approach only works for worlds where there are line-of-sight blockages but decreases the bandwidth needed and therefore improves scalability in conjunction with interactivity.

Kiwano [20] uses a cloud-based solution to handle avatar movement within a virtual world separate from the computing of static objects and entities within the world. It uses Delaunay triangulation indexing to efficiently assign neighbours to Kiwano nodes. By using a cloud-based solution one alleviates the problem of over-provisioning of resources that plagues C/MS systems by allowing the ready removal and introduction of resources from the cloud. Issues arise when the number of nodes grows as the chance for messages to have multiple hops and therefore greater latency increases.

Stellenbosch University https://scholar.sun.ac.za

Chord [21] scales logarithmically ($O(log(n))$ with the number of nodes by using a distributed lookup protocol to solve the problem of scalability in a P2P structure whilst still being able to reliably retrieve the global copies of states. Pastry similarly does this by using a 128-bit identifier to efficiently look up peers within the network. Each neighbour is aware of the nodes whose identifier key is numerically closest to its key. These two approaches look to improve interactivity by making it easy to find and contact neighbours, with Pastry [22] successfully testing a system with 100,000 clients.

VAST [2] is a P2P overlay that uses spatial publish/subscribe to efficiently disseminate updates. It is described in more detail in the section below.

1.2 VAST

VAST is a hybrid P2P system that uses a spatial publish/subscribe (SPS) system to provide a scalable architecture for MMVEs. VAST is composed of three parts:

1. The Voronoi overlay network, VON, which handles all the network requirements and network consistency requirements of the system.
2. The Voronoi self-organising overlay (VSO) which uses nodes called *matchers* to handle all of the update and event dissemination and load balancing.
3. The entry server (ES) which handles all login procedures and message routing.

This can be seen in figure 1.2.1.

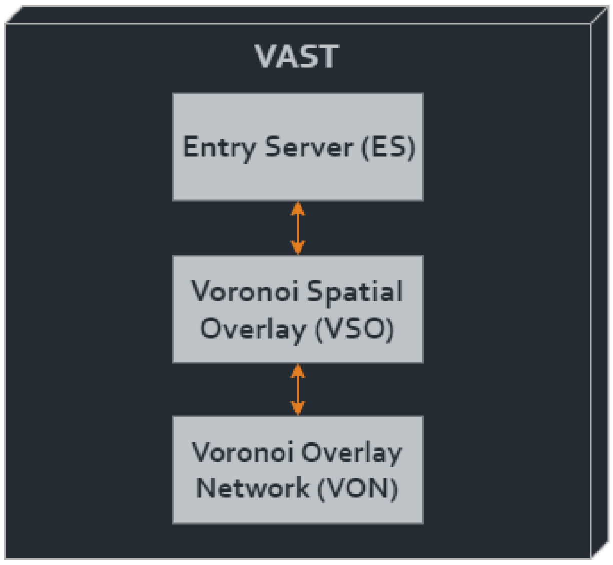
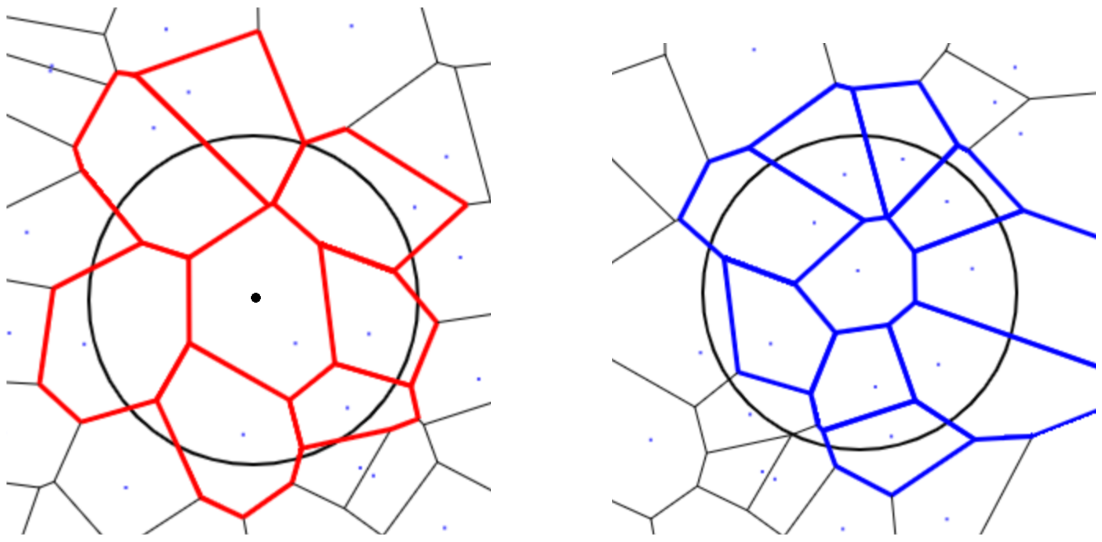


Figure 1.2.1: A figure showing the different layers to VAST and their communication channels [2]

VAST aims to improve the scalability of the system whilst maintaining consistency and interactivity by replacing an MMVE's network system with its own. It does this by distributing the VE across multiple servers in a P2P overlay. This allows the system to be dynamic in its resource usage while also being able to handle fluctuating loads. It limits

Stellenbosch University https://scholar.sun.ac.za


(a) A figure showing the enclosing neighbours (regions outlined in red) of a peer with its AoI

(b) A figure showing the boundary neighbours (regions outlined in blue) of a peer with its AoI

Figure 1.2.2: Two types of neighbours in a Voronoi diagram [2]

the number of neighbours a single server has through the use of Voronoi diagrams. Each server is assigned a Voronoi region and neighbours are determined using this as well as how the other regions lie around it. There are three types of neighbours: *enclosing neighbours*, *boundary neighbours* and *AoI neighbours*. *Enclosing neighbours* are neighbours that share an edge with a specific site's Voronoi region (seen in red in figure 1.2.2a). *Boundary neighbours* are defined as sites that fall within a site's AoI and form a boundary around the site's Voronoi region (seen in blue in figure 1.2.2b). *AoI neighbours* are simply the neighbours whose position falls within a peer's AoI.

Clients connect to this P2P overlay through an ES that handles login procedures and message routing. From the client's perspective, VAST looks like a single server. However, when a client joins it is assigned a starting server based on the position of its spawning point and which Voronoi region this lies within. When the client crosses these regional boundaries a handover sequence occurs that routes the packets that the client sends and receives through the new region's server. If a single server has too many clients connected a load balancing procedure happens that prevents a single server from being overloaded. If the entire system starts becoming overloaded more servers can be added to increase the capacity of the system.

Update and event dissemination is handled by the SPS system. A client states its interest within the VE by subscribing to an area in which it wants to receive updates (usually consistent with its AoI). Events and updates are then published to a specific position or area and any subscriptions that intersect with these publications receive them. This is a form of multi-casting as a single message can be sent with multiple receivers defined by an interest set. This helps to reduce the number of packets that the server needs to send as it can publish to the intended clients with a single message instead of sending individually to each one. This synchronises well with MMVEs as almost all events and

Stellenbosch University https://scholar.sun.ac.za

updates occur at a specific location within the VE with clients also being limited in what they interact with to a specific area.

Minecraft is one such MMVE. The following section will introduce the game and explain what it is in essence and how it works.

1.3 Minecraft

Minecraft started as a sandbox survival game created by Swedish game developer Markus Persson [13]. It was published on the 18th of November 2011 by Mojang. It has since become a massively popular game with the current player count at over 112 million active monthly players [23].

1.3.1 The world

The Minecraft world is made up of cubes 1 meter in length and of varying types. Players within the game control an avatar that can break and collect these different blocks to place them in creative ways and craft items. These varying blocks are placed in 16x16x256 columns called chunks. When a player's AoI intersects with any part of a chunk, the entire chunk is loaded into the player's memory. Similarly, when a player's AoI stops intersecting a chunk, it gets unloaded from the client's memory.

1.3.2 Objects, entities and the event/update cycle

The world is filled with NPCs such as animals and zombies that the player can interact with, as well as other players when playing in a multiplayer fashion. These entities and players create events when they interact with each other and with other objects.

Users connect to a Minecraft server that hosts the world within which all of the objects and entities exist. Players can interact with other objects and entities and this is sent to the server which then processes the event using the internal game logic. The server then updates the global copy of the affected states and subsequently generates updates that get disseminated to other players in the system. These updates renew the player's local copies of the states of the affected objects/entities.

Stellenbosch University https://scholar.sun.ac.za

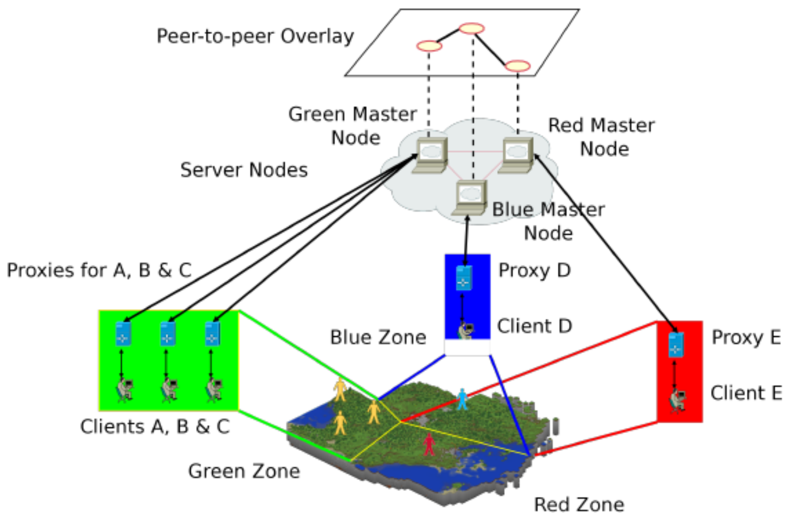


Figure 1.3.1: The Koekepan architecture [3]

1.4 Koekepan

Koekepan [3] is a system that aims to turn Minecraft into a generic research tool for use in MMVE, scalability and other such areas of research. It uses a distributed architecture that consists of a Minecraft server clone (such as Bukkit), an unmodified Minecraft client, a client proxy and Augeo, a server plugin for Bukkit. This architecture is shown in figure 1.3.1. Koekepan uses Minecraft's inherent networking system, as opposed to VAST which instead replaces it. The server and Augeo are run together and therefore make up the server node. It is created in this modular fashion so that the researcher can control different aspects of Minecraft and the architecture such as the networking between the server and clients, the AoI management and the partitioning of the VE. It can be run in a C/MS or P2P architecture due to these abstractions.

The VE is sectioned into different regions using zoning [24], also known as spatial partitioning, with a server node as the authority within that region. The different server nodes are organised in a P2P overlay to determine with whom they can communicate. Being the authority of a zone means that the global states of entities, objects, and players within that zone are controlled by the server node, as well as the event and update dissemination within that zone. These dynamic zones allow for both load balancing and dynamic server addition and removal. When clients migrate across zones, a connection handover sequence is performed, wherein the client connection is transferred between server nodes. It does this by briefly creating a connection to the destination server so that the client is connected to both servers. This is to ensure that the connection is not dropped.

Stellenbosch University https://scholar.sun.ac.za

1.5 Research goal

This thesis aims to complete the implementation of the JavaScript version of VAST and prove its scalable properties. It aims to use Koekepan's proxies to interface between VAST and Minecraft to show that Minecraft can successfully be operated through an SPS structure. Using the scalability of VAST and the viability of Minecraft through an SPS, this thesis aims to prove the scalability of MMVEs using VAST.

1.6 Thesis Approach

The approach taken by this thesis is as follows:

1. Firstly, this thesis undertakes to do an exposé of research on VAST to explain how it works and how to properly implement it with all of its features. The system will also be analysed to see how well it can be integrated with MMVEs.
2. This thesis will subsequently convert the current implementation of VAST in C++ to JavaScript, starting with the networking layer, VON.
3. VON will be implemented to handle neighbour discovery procedures as well as the addition and removal of other VON peers. VON will be rigorously stress-tested for any faults as well as to test its scalability.
4. This thesis will then implement the VSO layer. This layer will be designed to handle SPS functions, which consist of subscription and publication management. It will work in conjunction with VON for its neighbour discovery and communication requirements. It will be designed to accept connections from the ES and for each matcher to be in charge of its load monitoring and request for balancing.
5. The ES will then be designed and implemented. It will provide an API for clients to use to interface with VAST and will route all client information to the matchers and back to the clients. It will handle the positioning of matchers in load-balancing procedures.
6. The VSO layer and ES will then be tested to see if it provides the same scalability properties that the VON layer showed.
7. Lastly, VAST will be integrated with Minecraft using the developed interface to translate between them. It will be shown that Minecraft can be used with an SPS structure, and therefore VAST can be used to make the Minecraft server scalable.

1.7 Thesis Objectives

The thesis aims to:

1. Successfully convert VAST and its features from C++ to JavaScript
2. Show that VAST has scalable properties as a standalone system
3. Show that SPS is a viable form of event/update dissemination in an MMVE environment and assists in scalability

Stellenbosch University https://scholar.sun.ac.za

4. Create an interface for Minecraft to integrate with VAST.
5. Show that a real MMVE such as Minecraft functions when the event/update dissemination is replaced with an SPS system such as VAST.

1.8 Thesis Overview

This section gives an overview of the layout of the thesis and what will be discussed.

In Chapter 2, the background is given on MMVEs and how they work. The chapter starts with an in-depth discussion of MMVE requirements and the different aspects of it. Following this is a discussion of MMVE architectures and their advantages and disadvantages, ending with an observation of the solutions to these architectural flaws that have previously been researched.

VAST is discussed in more detail in chapter 3. Background on the system is given followed by a discussion on the system's three layers and the different aspects that function together to power the system. The procedures of VON are discussed, followed by a more in-depth analysis of SPS. Finally, the VAST API is outlined and challenges and considerations are noted.

In Chapter 4 a background on Minecraft is given with an emphasis on the inner workings of each aspect of the system. The server, world state, player entity, NPCs, networking and Minecraft protocol, and interest management are all discussed in more detail. Event and update dissemination in Minecraft is examined as well as MMVE requirements in relation to Minecraft. Finally, previous research in Minecraft is summarised.

Chapter 5 describes the integration of VAST and Minecraft and how Koekepan interprets between them. The client and server proxy for the Minecraft system are analysed followed by the system's interactions in the Minecraft login procedure. The challenges in the implementation complete the discussion of the design of the system. A theoretical and practical analysis of Minecraft's SPS potential is then explored.

Chapter 6 evaluates the performance of both VAST and its integration with Minecraft. Firstly, the metrics used to test the systems are outlined and explained. Then a test of VON's scalability potential is presented. This contains two tests: one that varies the peer density by keeping a constant AoI whilst increasing the peer count in the system, and another that keeps peer density constant by varying the AoI whilst increasing peer count. The VSO and ES layers are then tested to show their scalable potential as more matchers are added to the system. Finally, a test showing Minecraft working through an SPS system is presented.

A conclusion on the system and its performance is presented in Chapter 7. It discusses how the thesis meets the defined objectives and describes further work that can be done to improve the system as well as how to add more features and capabilities to the system.

Stellenbosch University https://scholar.sun.ac.za

Chapter 2

Background of MMVEs

This chapter serves to introduce the requirements of the MMVE genre. It will first discuss the different requirements of MMVEs. Following this will be a discussion on MMVE architectures and their respective advantages and disadvantages and will end off by looking at the solutions to some of the problems with these architectures and the requirements they fail to adequately fulfil, finally focusing on the problem that this thesis aims to solve.

2.1 MMVE requirements

The requirements of an MMVE are needs that must be fulfilled to some extent to have a game that is both enjoyable to play and performs to the level that experience is not negatively impacted when using the game in a reasonable way (being able to play with the number of people that the game is designed to be played with) [25]. The user experience is determined by how well these requirements are met. These requirements include:

- Interactivity and fairness between players and in-game objects. This is how fluid the game feels for the player as well as how equal the experience is for every player within the game.

- Consistency in the view that a player has of the world around them. When the player's view is consistent with the actual state of the world, it gives the player trust in the system and makes the game enjoyable to play. When it is inconsistent, the game is no longer fully immersive as the player distrusts the authenticity of the experience.

- Persistence of the world after an indefinite time. When the game is not persistent, the player cannot build upon what they did before and results in a lack of progression in the game, making the game frustrating and less compelling to play.

- Scalability of the architecture as more players join the system. When players cannot play with the people that they desire to due to the performance of the game not allowing it, user experience is negatively impacted.

These requirements, introduced in section 1.1.6, are discussed in more detail below.

Stellenbosch University https://scholar.sun.ac.za

2.1.1 Interactivity and fairness

This section will talk about interactivity and the different factors that pertain to it. It will also discuss fairness and how interactivity and fairness are linked as well as ways to improve them.

Interactivity is how responsive the game feels to the user when interacting with objects and entities within the VE. The type of game and how tolerant it is to latency are factors that affect interactivity.

Game types and latency tolerance

Latency is defined as the time it takes an update to reach its recipient once it has been generated. Ordinarily it is measured in milliseconds (ms). Latency within MMVEs is important because the longer it takes for the updates to reach the intended destination, the less the player will be responding to what is happening in real-time and so the less accurate the player's responding updates will be. This can create frustration and distrust of the game as the user's actions will not result in the expected response.

Claypool et al.[26] speak about different latency thresholds for different games based off of their perspective and the genre of game. [26] states that if the player looks from the perspective of the avatar (first person) then the latency requirements are a lot stricter because the player's view is most similar to how the player experiences the real world and so any 'jittering' appears more pronounced. This includes game genres such as FPS and racing games. The latency threshold in these genres was found to be 100ms before the gameplay was unpleasant. If the player is looking from an over-the-shoulder or isometric view (third person) the view is like looking at someone else and so is less strict but still close enough that any lagging that occurs is noticeable but more manageable than first-person. This includes the sport and role-playing game (RPG) genres and the threshold is 500ms. For both of the aforementioned cases, the player controls a specific avatar, but there are genres of games where the player is omnipresent, meaning that the viewport is not specifically tied to a single avatar or object. In this omnipresent view, the player has a very high view of what is happening, smaller inconsistencies are less noticeable and is, therefore, the least strict of all three cases with a threshold of 1000ms, or 1s. This includes the genres of real-time strategy (RTS) and simulations such as SimCity or The Sims. For an MMVE the viewport is tied to the avatar and can be either first- or third-person. This means that the maximum latency threshold sits between 100-500ms before it starts affecting the gameplay.

When the latency exceeds these thresholds, the performance of the game becomes unacceptable and makes the game uncomfortable, undesirable or frustrating to play. This means that games have to meet the minimum latency requirements of the genre for them to be feasible.

Fairness talks about how differing latency between the client and server across users can put into effect natural advantages for those with low latencies [15]. This is due to their interactions being registered more quickly and updates to be received more quickly which

Stellenbosch University https://scholar.sun.ac.za

allows the player to make decisions on how to respond faster. To counteract this, measures are taken to improve the interactivity between players and therefore improve how fair the game is.

Bandwidth

Bandwidth requirements describe the amount of bandwidth needed to interact with an MMVE completely without losing access to critical information. This affects both interactivity and fairness as having less bandwidth than what is required means that information that describes interactions is lost, resulting in lower interactivity. This by definition creates a difference in the gaming environment of the player and therefore reduces fairness.

2.1.2 Consistency

The requirement of consistency is vital to an MMVE. To better understand what consistency is, one needs to understand object types, different player interactions with objects and finally object replication and consistency control.

Object Types

Object types refer to the different categories that all in-game entities are grouped into. These four categories are:

- immutable objects
- mutable objects
- characters, or avatars
- Non-player characters

Immutable objects are objects within the VE that cannot be altered or changed. They are instantiated when the world is created and continue to exist for the rest of the lifetime of the VE. This usually consists of the world that the rest of the objects are in, such as buildings and other structures. These objects are installed on the client-side and are not altered after that.

Mutable objects are assets within the VE that can be altered and changed. They usually have a base form that is instantiated either when the VE is created, or when an in-game event spawns them, after which they can be modified, used or changed by events or updates initiated by the state manager, avatars or NPCs within the VE. This can include weapons, tools, and food and are temporary within the VE.

Player characters, also known as avatars, are sprites that the player controls to enact changes in the world. The player installs a client on their machine and this client is a portal through which the player controls the avatar. It takes their input and translates it to movement and actions performed by the avatar in the VE. These actions are grouped into three different categories: player updates, player-object interactions, and player-player interactions [17][18]. These change the player's state, other objects' states or other players' states respectively. They are discussed below in section 2.1.2.

Stellenbosch University https://scholar.sun.ac.za

Finally there are NPCs or *bots*. These are avatar-like entities within the VE that are controlled by algorithms that make them seem to have a mind of their own. They can enact changes to mutable objects as well as other NPCs and player characters.

Player interactions

As mentioned above, there are three categories in player interactions. These are player events, player-object interaction and player-player interactions.

Player events refer to interactions wherein the action only affects the player, such as inventory management, movement or player statistics changes (health, mana, etc.). Most updates in systems that are simple or unoptimised are movement updates [17].

Player-object interactions are where the player's actions change the state of a mutable object. This involves any changes to the world or other objects within the world, such as picking up a mutable object and adding it to the player's inventory, or using a tool to affect an object in the environment.

Player-player interactions are where the actions of one player's character affect another player's character. These actions affect the state of another player's character, such as a change of health or a transfer of items between inventories. Since NPCs are avatars controlled by AI, player interactions between players and NPC can either be considered as player-player interactions or player-object interactions depending on how the system is defined.

Object replication

Mutable objects within the VE are stored as states. These states are stored on the server hosting the VE in C/S architectures. When a client connects to the server, it receives a copy of the state of the relevant mutable and immutable objects which are called secondary copies or replicas. This copy is not the authoritative copy (called the primary or global copy) of the object. In essence, this means that if a client makes a change locally to the state of an object but does not authenticate the change with the server, which has the authoritative state, then the change will be reverted the next time the client receives an updated version of the object state.

In distributed server or P2P architectures, multiple machines can host mutable and immutable objects. By definition, since there is no centralised authority within the system, there is no central machine that owns all of the objects within the system. Therefore, every object has a specific server or peer that owns the object. In this case, the copy of the object state can be given not only to clients that connect but also to other servers who require the information. The changes applied by holders of replicas are sent via an update to the owner of the global copy and then distributed to all of the other replica holders who consequently update their replicas with the change. Publish-subscribe systems use this idea as the core concept of the system. Every replica of an object state subscribes to the publications emitted by the global copy. This concept is explained in more detail in section 2.3.1.

Stellenbosch University https://scholar.sun.ac.za

With clients using replicas of an object instead of the global copy, there is a chance that the information can become desynchronised when delays are introduced into the update dissemination process. This poses a challenge which is known as consistency control, which is discussed in section 2.1.2 below.

Consistency control

When there is a loss of synchronisation between the global and local, this is called inconsistency. This can happen when there is a delay between an update to an object being generated and it being sent to the owner of the global copy, or when the global is disseminating the update to an object to other clients and there is a delay. This can have varying levels of consequence: from insignificant, to game-breaking. An insignificant inconsistency could, for example, be when an object whose sole purpose is decoration is in a slightly different position to the position recorded in the global copy. This will not result in the game being unplayable but could be a minor annoyance if not fixed quickly. An example of a game-breaking inconsistency is if a player's position in an FPS game is shown to be in another player's line of fire but they are actually in a position that is not in this line of fire. This kind of interaction creates frustration for the player and has the potential to ruin the experience for the player if it happens frequently.

Another source of inconsistency can come from parallel or conflicting updates happening to the same object. If two updates happen nearly at the same time, the order in which the updates happen can be very important. Take the scenario where two low-health players are running for a single health pack. They both have replicas of each other as well as of the health pack. Both players arrive at almost the same time, after which an explosion happens that would be fatal for the player who does not receive the extra health from the pack. Which player picks up the health pack and therefore does not die in the game? It is unfair for the player who does not receive it, as if there is any delay in updating the local state with what happened, what will appear to happen is that they will pick up the health pack which is no longer there, incorrectly appear to gain the health and then die due to what, from their perspective, should not have had the ability to kill them due to their actual health level being passed to them from the owner of the global copy of their state. This would be aggravating for any player of such game and is therefore why consistency control is so important.

The final possible cause of inconsistency is if the update gets lost due to an unreliable connection to the host of the global copy of an object or a poorly designed system. If an update gets lost, then the replica is desynchronised from the global copy and only another update can rectify it if there is no inherent consistency control.

Consistency is measured by comparing the replica with the global copy at the same instance in time. The most important metric in this project's context is precision and will be explained more in section 6.1.1. This is how closely the replica matches the global copy's data at any point in time.

Consistency is, therefore, a time-sensitive problem. One could have very strong consistency by insisting that every disseminated update be treated as a transaction with transactional properties such as isolation or atomicity. Isolated transactions are where the owner of the global copy of an object ensures that concurrent updates are individually

Stellenbosch University https://scholar.sun.ac.za

processed. Each update should not work with intermittent data produced by the processing of another update, but rather the result of the preceding update. Atomic transactions either apply all updates to an object, or none at all. This means that if a single update cannot be processed for an object then none of the subsequent updates will be processed. Both of these transactional properties ensure that consistency is maintained by taking more time to ensure consistency is held. An MMVE need not treat updates as transactions though as MMVEs do not require strong consistency due to the slower nature of the genre.

As discussed in 2.1.1, [26], [27] describes the threshold for the maximum latency a player can experience for different game models before the player experience is unreasonably affected. MMVEs fall into either the first person or third person category depending on the game, where the maximum threshold is 100ms or 500ms. From this, we can see that the order of magnitude of time is in the milliseconds.

The other option for consistency is eventual or weak consistency [28]. This means that objects are allowed to be temporarily inconsistent but will eventually become consistent if given enough time even if updates ceased to be received for a long time. Brewer's conjecture, expanded on in [29], states that a web service cannot guarantee more than two of the following three properties: *consistency*, *availability* and *partition-tolerance*. Since availability and partition tolerance are highly favoured in MMVEs, consistency is usually not ensured, but rather inconsistency resolution is favoured and at most eventual consistency. One option that is appealing is varying the levels of consistency according to the importance of the object, for example, ensuring consistency for health packs but only having weak consistency for positional updates.

There are two categories of techniques for consistency control - Predictive Contract Mechanisms (PCM) [30] and multiresolution simulation [31] - but only one is relevant here and that is a form of PCM called *dead reckoning*. Dead reckoning predicts the movement of an object/avatar in motion when the expected does not arrive on time. This is done by taking the last received positional update and projecting where the entity will be in the next frame (snapshot at the time between ticks, explained in section 2.3.2) using the motion vectors of the object. This is effective if there are sparse occurrences of positional update loss, but fails to varying degrees when there is a change of direction contained within an update.

Bandwidth requirements

The bandwidth requirement in MMVEs refers to the upload and download speeds required of the client's connection to the network of the MMVE in order to operate normally within it. This means that it needs to be able to send and receive updates at the appropriate rate so that it does not result in inconsistency for the client. It can be calculated based on the average message size, update rate and the number of recipients. In systems with millions of players or high update rates, the bandwidth requirements are high. The client also needs to be able to handle bursts of network traffic due to in-game events that can generate a lot of updates all at once or attract many players into a small area, which would have the same effect.

Stellenbosch University https://scholar.sun.ac.za

2.1.3 Persistence

Persistence is the ability of the MMVE to retain the state of states over an indefinite period of time. A client who leaves their profile and virtual items untouched for an extended period of time should be able to come back to it and them in the same state that they were left in. This requires the MMVE to be able to store the states of all of the entities and objects within the game somehow and for those states to be retrievable at any stage. This is easy for architectures that have a centralised authority that the states can be stored on, but more difficult for distributed architectures where this centralised point is non-existent or difficult to define. The architecture should be able to handle any or all players leaving the system only to come back to it in the same state that they left it in (barring natural, intentional changes caused by the normal running of the VE).

Another problem with persistence in distributed systems is the referencing of objects across storage hosts. If two server nodes exchange information about states of an object, they could use the same identifying code yet be referencing two different objects [32]. In this case, a system where one identifier takes precedence solves this problem.

2.1.4 Scalability

The problem with scalability in MMVEs is that there are many potential bottlenecks that can cause the MMVE to not be scalable. These bottlenecks are dependant on the computational power, latency and bandwidth requirements of the client and host systems. This can be controlled with something called *interest management* (IM). However, before discussing IM, the three bottlenecks are dissected.

Modern MMVEs can have millions of users playing them at once. To handle this many players, a sophisticated hosting architecture is required to meet scalability requirements and the computational and network limitations of having all of these players in the VE simultaneously. These requirements are categorised by [33] as follows:

- Bandwidth
- Network latency
- Computational power

When designing for scalability, graph 2.1.1 shows the basic relationship that each resource must hold to as the number of clients increases in the system. Each one of these limitations has to be addressed when designing an MMVE for thousands of players and more.
The following sections look at these three bottlenecks and an aspect of scalability called interest management.

Stellenbosch University https://scholar.sun.ac.za

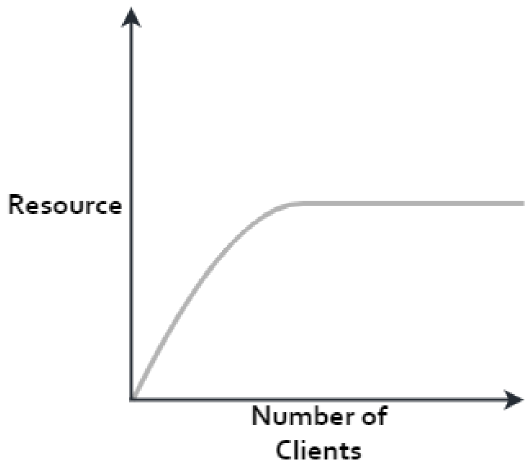


Figure 2.1.1: The ideal relationship between resource usage and the number of clients in a scalable system [4]

Bandwidth

Bandwidth is the amount of network data travelling through a communication channel over a certain period of time. Every packet of information that is sent between nodes in a network has a certain size, usually measured in bytes (B) or kilobytes (KB). This information, measured over a period of 1 second, creates bandwidth measured in kilobytes per second (KBps or KB/s) or if enough data is passing through, megabytes per second (MB/s).

Every action in a VE creates information that needs to be communicated to other interested parties, which requires a certain amount of bandwidth. If enough of these actions happen, the required bandwidth can be higher than the server or connected clients have access to, creating a bottleneck. When this bottleneck occurs, either the clients have to wait to send the information through when possible, the packets will get corrupted as bits of the packet do not make it through the bottleneck, or the packet will fail to send altogether, which is called *packet loss*. All of this can cause erroneous behaviour in a system that is not robust enough to deal with it. If the packet loss is high enough the system can even fail completely making the game unplayable.

There are a few techniques that, in conjunction with IM, affect how much bandwidth a system uses when it sends out updates and events. Early network implementation would send every message to each client, which is called broadcasting. This is very inefficient as not every user needs every packet and so redundant information is sent. Uni-casting sends a single message to a single client. This is the most efficient option when the packet is unique to the client, but not when the same packet needs to be sent to multiple clients (which is usually the case) as the sender will have to send the same packet multiple times. Multicasting is, therefore, the most efficient method as a single packet can be sent to multiple clients that have joined a multicast group. This can be seen in figure 2.1.2.

This allows clients to send a single message (like unicast) to a receiver that can then multicast (as in broadcasting) to a multicast group. It incorporates the best aspects of the previous two approaches which is why it is used more than any other technique. There

Stellenbosch University https://scholar.sun.ac.za

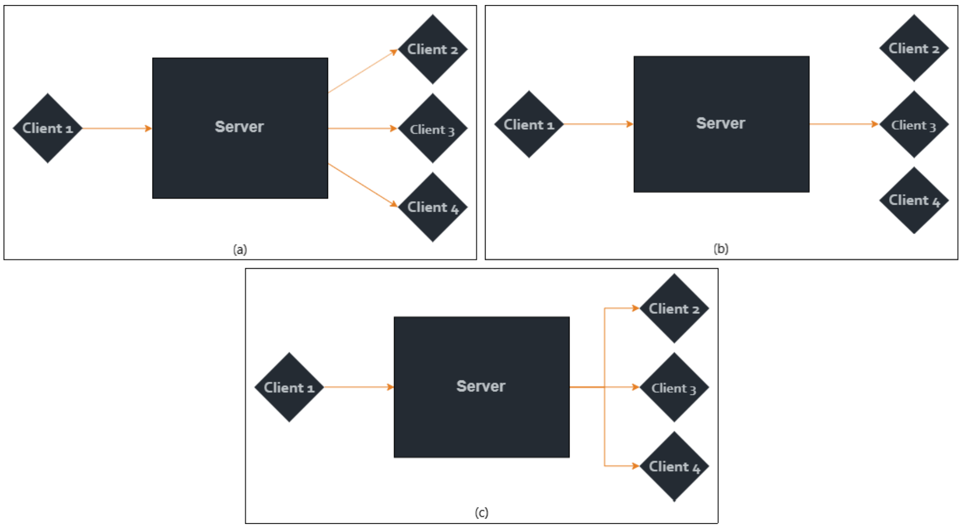


Figure 2.1.2: In a) the sender broadcasts a packet received from a client as multiple packets to every other client. In b) the received packet is sent to a single client at a time. In c) a single packet received is sent to multiple clients.

are numerous examples of multi-casting being used in systems such as Scribe [34] and techniques such as predictive binary tree (PBT) [35]. A useful technique that is used in this thesis is spatial publish/subscribe (SPS) systems, explained in section 2.3.1.

Latency

Latency, described above in 2.1.1, describes how long a message takes to reach the designated destination. It is important to note that because the communication mediums (whether fibre optic or electrical wiring) are not ideal, latency can never be completely eliminated by using these channels. As such, the latency limits discussed before need to be low enough to handle these extra latencies as well. If the latency goes above the threshold of what has been measured as acceptable for extended periods of time, then it becomes the bottleneck in the system and any further scaling would be futile as it would make it worse and would still be unplayable.

Computational Power

Computational power refers to how much data the host machine or user machine can process without slowing down to the level where game-play is adversely affected. These can refer to the computational processing unit (CPU) and graphical processing unit (GPU) processing time, memory usage and accessing as well as retrieval and writing of information to and from storage devices where the bulk of a game's assets are stored.

Every update and event that occurs within a game requires the processing of millions of calculations on low-level computational levels. These calculations are done by the CPU, which has multiple cores that allow it to do many calculations at once. Since a large portion of these calculations are focused around producing an image that the user interacts with, a specialised CPU focused around computing graphics-related calculations is utilised, called the GPU. Whilst the intensity of the computations can be adjusted for

Stellenbosch University https://scholar.sun.ac.za

graphics by changing in-game settings, the intensity of calculations done by the CPU is directly influenced by how the game is designed and it is, therefore, important to design the game in a way that reduces the CPU usage as much as possible. So in the realm of scalability, the more information coming in about clients connected to the system, the harder the CPU has to work until the amount of information to be processed is coming in faster than it can be processed, producing a delay in the processing of events and updates. This leads to reduced performance of the system in a similar way to how latency would but can be even more detrimental as it affects the local performance of the game and not just what is perceived from others.

Similarly, information that comes in needs to be readily available for processing. This requires space in memory (known as random access memory - RAM) as RAM is much faster than longer-term storage such as HDDs and SSDs. This memory is smaller as a result and thus is more valuable when it comes to what we decided to keep in memory. This means that not all information can be received and kept in RAM when we are working with scalable systems as even if each client added only contributes a small percentage to the total memory usage, if one scales up then the memory will quickly become a bottleneck. The ideal design results in behaviour where the system reaches a limit where more clients no longer increase the memory usage, as in figure 2.1.1 above.

Finally, if information needs to be accessed from a slower form of storage in HDDs and SSDs, this needs to be done in a way where the system can do it as efficiently as possible, as this accessing is relatively slow when compared to how everything else in the system works. For example, if a player needs to load a dungeon that the rest of their party has gone in to, but the loading of resources from the hard drive is slow, then they could be stuck in the loading screen for longer than necessary, impacting player enjoyment and possibly game playability.

In the final section, this thesis will look at what has been done already to try and address the scalability issues in MMVEs and other scalable networks that face the same problems.

2.2 MMVE architectures

There are four different architectures that MMVEs run on:

- Client/server
- Client/multi-server
- Peer-to-peer
- Hybrid

Each one has its advantages and disadvantages when it comes to the requirements stated above. These are discussed below.

Stellenbosch University https://scholar.sun.ac.za

2.2.1 Client/server

The CS architecture is the most basic of all of the architectures. It is widely used in simple systems as it has much control over the system due to the single centralised authority in the server [15]. There can be multiple clients that connect to a single system. The immediate problem that becomes apparent is the bottleneck since all of the events and updates have to be routed through the server before going to the intended clients. This has a huge impact on scalability and is, therefore, one of the main reasons that other architectures get researched.

The other requirements affected by the scalability issue is interactivity and availability. If there are many clients in the system, all of the updates that get sent to the server can easily overload the computing power and bandwidth limitations resulting in higher latency in processing events and sending updates out to the clients. This impacts how quickly a client action results in a visible change that holds and therefore impacts the user experience. Over and above this, since everything is stored on the server, if the server is off or if it crashes then no one can retrieve information and the VE will not be available to the players. These servers are expensive to install and maintain as they need to be very powerful in order to manage the loads that they carry. Whilst C/S is the worst of the four architectures when focusing on scalable systems, these are the only disadvantages it has.

The advantages of this architecture come in the consistency, persistence, and fairness of the system. Since the server acts as a centralised authority, the global copies of states are all stored in one place, giving clients a single place to authorise their local copies. This allows the clients to stay strongly consistent assuming the scalability aspect has been respected and the number of clients is low enough for the system to be running normally. The persistence is easy to guarantee since all of the state information can be stored on the server and retrieved from there. Finally, since all of the clients have to authorise their events and receive updates from the single server, there is no unfairness between clients with better machines as the server does not give any preference to who is sending information and treats it all equally. However the problem of players having different latencies and faster or slower connections is still present and has to be addressed, but this is not a result of the architecture.

2.2.2 Client/multi-server

C/MS works similarly to C/S except that it has many servers that the clients can connect to and play through instead of the single one. Since the MMVE architect has control over the communication mediums and channels between servers, they can be designed in a way that accounts for minimal latency, high bandwidth capabilities, and robustness. Servers share information between themselves and work together to host a world that is capable of handling vastly more clients than a single server could. The challenge then comes in making sure that every server can access any information, even if it is stored on a different server.

Stellenbosch University https://scholar.sun.ac.za

Cloud computing

An implementation of this that has become popular in recent times is cloud computing. Cloud computing takes advantage of the high-speed data lines between servers in the cloud as well as their enormous computational capacity to host MMVEs for many more clients than regular servers could. It can also divide different modules of the MMVE such as login management and authentication onto different machines so that these services are not hindered by overloading on different parts of the architecture [36].

The disadvantage of this architecture is the cost of running such a system. In [37], an example is given where a system (comprising of an entry server, several map servers, and a database server) using 27 TB (terabytes) per 12 hours upstream bandwidth and 4.7TB downstream bandwidth usage at the prices of using Amazon EC2 ($0.08 per gigabyte (GB)) would result in a monthly fee of $130,000 just for bandwidth usage. As one can see, this cost is not affordable for any but the largest of game companies.

The advantage of cloud computing specifically is the on-demand nature of it. Since the server capacity is rented only for what is needed, this allows flexibility in how much computational power is needed from the servers and allows the MMVE to easily get more computational power when needed [38]. This is enticing as there is no excess spent on servers sitting on standby. The generic advantages of C/MS is the benefit of the centralised authority of C/S (persistence, consistency, and fairness) with the added power of having more than one server to share the computational and bandwidth requirements (improved scalability and interactivity).

For these reasons, C/MS is used by most of the industry for the trade-offs of cost versus benefits. The MMVE still has to be designed to make use of the interconnected nature of the servers.

2.2.3 Peer-to-peer

P2P is an architecture where there is no centralised authority within the system. Instead, all participants in the system fulfil the same role, that of being an active user in the system as well as a provider of computational and resource power to ensuring that the system functions. Each user has a responsibility to manage some part of the greater system.

One can immediately see that there is an issue of fairness in P2P. If one user has a very powerful machine and the other user a weak machine, if they are given the same load then the user with the more powerful machine would be more able to handle the load without adverse effects on interactivity, whereas the user with the less powerful machine might not be able to handle the load and maintain the same level of interactivity with the game.

P2P is a contentious architecture that has been well researched [14][39][40][15] because of how appealing the scalable aspects of the architecture are. Whilst there are issues in almost all of the requirements stated in section 2.1, the promise it brings in handling the scalability issues of C/S and C/MS results in it being worth investigating. Since each client brings its own computational power, the system, in theory, can be self-sufficient. No additional computational resources need to be added no matter how many clients join the

Stellenbosch University https://scholar.sun.ac.za

system. There is no idle computational power as it is always immersed within the system. All of this in an extremely cost-effective way which is in direct contrast to C/S and C/MS.

The issues appear when considering the other requirements. Since there is no centralised authority where global copies of states can be stored, the issue arises as to how every entity that interacts with another entity knows where to find the global copy of its state. And even if it knows which client the global copy of the state is stored on, if that client is offline, the global copy may not be there or may be entirely unavailable. So there is the issue of persistence in the system. Interactivity is also a big problem for P2P systems. If all of the clients within a system try and communicate with each other, the bandwidth requirements and resulting computational load would quickly become a bottleneck for the clients. Finally, since there is no central authority in the system, the safety of the information such as global copies of states can be in jeopardy from ill-intentioned peers in the system.

2.2.4 Hybrid P2P

Hybrid P2P architectures attempt to take the best of both C/S and P2P architectures and meld them into a single mega architecture. They take the promise of resource addition via client contribution from P2P and connect it with the centralised authority benefits of C/S and C/MS. It uses the concept of a super-peer, a peer in a P2P distribution that has powerful computational and bandwidth capabilities and elevated permissions and therefore has an authoritative role within the P2P network. The super-peer needs to have both the bandwidth and computational resources to handle the extra load and network traffic that results from ordinary peers checking for consistency. It also needs resources for other functions that the super-peer may have such as storage of sensitive information and player progress and state.

The disadvantage of the hybrid architecture is that if a super-peer is chosen incorrectly and they do not have the capabilities to be one, then the entire system can be affected. If powerful servers are chosen as super peers then the cost of getting such servers is higher than in a P2P system but lower than in a C/S or C/MS system. If the super-peer is a peer, then it is also not fair for them to extend extra resources to maintaining the system over and above their ordinary requirements in being a peer. This is why super peers are usually chosen to be servers so that there is fairness between all peers in the system.

The advantages are that you are getting the benefits of consistency management, persistency and interactivity that comes from having a semi-centralised authority in the super peers whilst also having the scalability of P2P architectures.

Table 2.2.1 shows the relative advantages and disadvantages of each architecture. No one architecture is better than another, but rather which architecture is better suited to the requirements. The hybrid solution is an attractive architecture however as it combines the advantages of the previous three architectures whilst minimising disadvantages.

Many solutions have been researched to try and fix these problems with each architecture. They will be discussed in the next section.

Stellenbosch University https://scholar.sun.ac.za

Table 2.2.1: Comparison of the advantages and disadvantages of different MMVE architectures

Architecture	Advantages	Disadvantages
C/S	- Consistent - High Security - Persistent - Fair - Simple to set up and manage	- Not Scalable - Servers are expensive - Not fault tolerant
C/MS	- Consistent - High Security - Persistent - Fair - Good fault tolerance	- More scalable than C/S but not completely - More expensive than C/S - Higher complexity than C/S
P2P	- Highly scalable - Low cost - Fault tolerant	- Bad consistency management - Persistency difficult to maintain without external help - Fairness - Difficult to develop - Security
Hybrid P2P	- Good scalability - Medium cost - Consistent - Persistent	- Super peers require more expensive hardware - Can be unfair if super peers are peers as well

2.3 Researched solutions to architectural problems

Many different solutions have been proposed to combat the requirement problems discussed above. These include both C/S, C/MS as well as P2P solutions and use two different approaches: adding more resources to the system to meet the game's requirements or decrease the consumption to match the current available resources' limits.

2.3.1 Generic solutions

The following solutions can be applied to C/S, C/MS, P2P, and hybrid P2P architectures. These include dead reckoning, interest management, and message aggregation and compression.

Stellenbosch University https://scholar.sun.ac.za

Dead reckoning

Dead reckoning is where a client predicts the movement or actions of an entity in instances where the updates are not received quickly enough or where the updates are lost [33]. This prediction can be based on the movement vectors that the entity displayed at the last update or can be controlled by an artificially intelligent agent (known as a bot) that uses a model such as a neural network to determine the next most likely movement [41]. It is a method to improve interactivity within the MMVE as it prevents entities from randomly stopping or moving erratically and smooths out inconsistencies in the receiving of updates due to lag. Whilst dead reckoning can provide consistency in the continuous domain of the client's view, it does so at the cost of true consistency and a correct state. This is seen as an acceptable trade-off in games that have moderate consistency requirements but is not good enough for other strict consistency replicated continuous applications [42] such as FPS games.

Interest Management

A player is said to be interested in an object if it requires the information about the object to function in the VE. IM is the determining of what the client would want to be aware of and what they could be focusing on and then only sending information pertaining to this. It is a large factor in the scalability of MMVEs. It is, therefore, an important aspect in the design of an MMVE. If the player were to receive updates and events of the entire VE at every point in time, the bandwidth usage would increase to an unacceptable level for every player that interacts with an object or another player. [43] shows how even a simulation of 1000 peers can have a bandwidth requirement of 3.75 megabits per second (Mbps) per peer and when reaching 100 000 peers it can use up to 375 Mbps per peer. The amount of information that the player would be receiving would be significantly more than the player could be aware of or even what they could feasibly interact with, making the updates and events unnecessary [18]. Therefore, the interest of the player is limited to only what the player is spatially close to or what is in vision and how this is achieved can have a drastic effect on the playability of the game.

IM is about choosing what information the player receives relative to its surroundings and therefore becomes a spatial problem. This spatial management of interest follows an *aura-nimbus* model [44] where the *aura* is defined as the boundaries of the world around the player's avatar and the *nimbus* is described as the area around the client in which all the objects it is interested in lie. The nimbus is often referred to as the AoI. A player can ordinarily only interact with those objects which lie within its AoI and therefore these objects are the only replicas that the player needs to receive updates about. This significantly reduces the bandwidth and computational requirements needed when compared to keeping replicas of every object within the VE.

This method of IM is subsidiary of a technique called *zoning* [24]. The simplest form of zoning is where the world is sectioned into completely separate zones and the AoI of the player is completely bounded by this zone. The only way for a player to interact with an object in another zone is to migrate to that zone. More complex forms of zoning involve sectioning a world into continuous zones where a player's AoI can reach over zone boundaries to interact with objects in that zone. The world can be pre-sectioned

Stellenbosch University https://scholar.sun.ac.za

into structured zone patterns such as grids [37][45][10] or hexagons [46][47][48], or into unstructured patterns such as triangulation [49][50] and Voronoi sections [4][1][51].

Grids are commonly used because of their simplicity in creating the zones and the straightforward nature in determining their size. Hexagonal grids are used because the hexagons are always adjacent to each other, meaning that players will always be moving into and interacting with objects in hexagons that are adjacent and therefore always known by the zone the player is in. As [15] notes, hexagons are also desirable because of their approximation to circles, which is the typical shape of an AoI.

Triangulation and Voronoi sectioning are useful as obstacles within the VE can be occluded as navigable zones when partitioning the world into VEs. This can lead to a reduced AoI which allows for fewer objects to be considered by the interest manager and therefore fewer replicas to be sent to the player which results in fewer updates needing to be sent and therefore a lower bandwidth requirement [49]. One technique of triangulation, Delaunay triangulation [49][50], make it easy to triangulate inside or around polygons that represent objects.

One of the most important aspects of zoning is the size of the zones. This consideration is application-specific and requires careful planning. If the zones are too big, they will contain too many objects and make IM inefficient as the player will only be interested in a small subset of these objects. If the zone is too small, the player's AoI could intersect multiple zones and make IM overly complex. Since objects within the world are not all static, this can make the requirements for zone size shift. This is where dynamic zoning can be useful.

Dynamic zoning is where the size of the zones are not precomputed but are constantly being recalculated to meet the changing demands of IM within the VE. Occasionally within an MMVE, there can be a point of interest or an event that can attract players and objects to a specific location in the world. These are not always predictable. In static zoning, this would overload the IM system as one zone would have a disproportionally larger amount of objects and avatars within it than other zones, rendering the zoning ineffective. Dynamic zoning combats this by constantly changing the size of the zones and their locations to alleviate the load on the zone where all of the players are flocking to. This characteristic of combating the overload of a zone is called *load balancing*. But what happens if neither dynamic nor static zoning works? The current solution is to have discrete and separate servers that can host a certain amount of players. The current record for the most concurrent players on a single server at a time is held by Eve Online's *Tranquility* server, which hit 65,303 concurrent players on the 5th of May in 2013 [52] with the most people in a single zone being held by the same game at 6,142 players in January 2018 [53]. However, this method of splitting up the population between servers can negatively impact the game experience of a player as it is limiting their interaction with other player's of the game.

[18] uses the idea of interest sets to determine which entities the player needs to receive updates about. In a busy VE, they determined that a player can only be aware of a certain amount of entities at a high fidelity at a single point in time. Using estimations,

Stellenbosch University https://scholar.sun.ac.za

they predict which entities these are going to be and use an entity called a *doppelgängers* to represent a low fidelity copy of an entity that can generally model the movement of that entity in the VE.

Message aggregation and compression

Message aggregation reduces the bandwidth of the system by aggregating similar information into a single update to reduce computational time in sending multiple packets as well as the bandwidth of message overhead such as packet headers and serialization information [49]. This can be especially powerful in MMVEs that have a lot of shared updated between players and do not utilise this already. The downside of this is the possibility of introducing network latency when waiting for additional messages to aggregate.

Message compression is something that MMVEs incorporate already due to the ease of implementation. Packets can be compressed to reduce the size of the data and therefore reduce network bandwidth. The disadvantage of the technique is increased computational time to compress the packets.

Spatial Publish-Subscribe (SPS)

SPS systems are based on the idea that each participant in the system has an AoI and an area of effect (AoE). The AoIs are represented by subscriptions, meaning that they have a center position in the VE and a radius representing the area that it covers. This area does not have to be a circle and can be any polygon, as long as it is closed. Any update or event that a player or entity enacts within the system is transmitted via a publication. These publications have a coordinate point representing the origin of the publication and optionally have a region that is affected by the publication. Depending on the addition of this region, these publications are respectively known as *point publications* and *area publications* respectively. If a publication or its region intersects a subscription region in any way then the owner of the subscription is sent the publication.

These subscriptions and publications are handled by an SPS manager. There need not be a centralised system that hosts and manages these subscriptions and publications as this task can be delegated to nodes in an overlay. These nodes can then ensure that the appropriate publications are sent to the right subscriptions.

There are two categories of message dissemination in an SPS: *content-based* and *channel-based* [54]. Content-based SPS filters messages according to their content type and then disseminates them according to whether subscriptions have subscribed to that content. This provides flexibility at the cost of computational time and resource usage. Channel-based SPS has the subscriptions declare the channel that they are subscribing to so that any publication that gets published has to be on the channel that the subscriber subscribed to.

In an MMVE, the client subscribes to the area that coincides with their view distance within the game and the subscription is updated to their position whenever they move. Then any interaction they have within the VE produces a publication at that point or at

Stellenbosch University https://scholar.sun.ac.za

the origin of the affected point and any other player who can see that would receive the update.

The advantage of SPS is that it provides a low resource way to reduce bandwidth costs and computational time by reducing the number of packets that need to be sent. Instead of every update needing to be regenerated for each client that the user is connected to, a single packet could be sent to the SPS manager which can then be disseminated. It also provides an efficient way to handle interest management as the interest expression is inherent in a client's subscriptions.

2.3.2 Client-server and multi-server solutions

A single server does not meet the requirements of MMOGs and MMVEs as the hardware is not good enough to handle the vast number of clients that can be playing a game at one time. However, using multiple servers in a distributed structure has been proven [4] to be an effective solution to the scalability problem prevalent in these architectures in an expensive manner. It uses the approach of adding more resources to the system so that it can handle the loads it is presented with.

Sharding, zoning and instancing

Under the category of reducing consumption are techniques such as sharding, zoning and instancing. Sharding is where an entire copy of the game world exists on a server and a client's entire experience can exist on one of these shards. This means that each shard with its clients acts as a singular C/S system. The disadvantage of this approach is that there is still a limit on the number of players that a single shard can handle, with little to no interconnection between shards.

Zoning is discussed above in section 2.3.1 and is where parts of the world are bounded and a player is completely encapsulated by this zone. When the VE is split into distinct zones, it creates an ideal environment to use zoning. Instancing is a technique that takes a group of avatars that have entered a zone and puts them in a copy of that zone. This zone is completely separate from other instances of the same zone, thereby reducing the load on the clients whilst allowing everyone to access the zone. This is an effective form of IM as it allows the developer to decide how many clients are in a single instance and therefore how many clients a single player will interact with. All of these approaches, however, have significant drawbacks in that they are not scalable, have minimal fault tolerance and have high set-up costs [15].

Bucket synchronisation and frame rate

Bucket synchronisation, or more commonly known as *local lag* [55], is used in most multi-player games as a way to combat latency issues in clients who have higher network latency. When there are multiple clients with varying levels of latency, updates can come through at different times and cause consistency and interactivity issues. To counteract this, the server collates updates across a specific period called a *frame* or *bucket* and only sends the collected updates at a synchronised interval predetermined by the design of the MMVE. This allows time for slow incoming updates to reach the server in the correct interval and for lagging connections to receive updates sent out by the server. This enforces fairness

Stellenbosch University https://scholar.sun.ac.za

between clients with fast and slow connections to be able to play the game in the same way. This also allows updates to be executed in the correct order as they can be received and rearranged according to the chronological order that they were instantiated. These frames are executed a specific number of times per second called the *frame rate*. This can be anywhere from 10-20 times every second [15]. The frame rate is very important to the overall experience a player has in the game as if it is too slow, the game can be inconsistent and possibly impossible to play but if it is too fast it is unfair for the clients with slightly slower connections. A faster update speed also means a higher bandwidth which can have other implications for the system. A balance needs to be found so that the experience is smooth and pleasant for the player, but also take into account the limitations of the bandwidth capabilities of the client connections.

2.3.3 P2P and hybrid P2P solutions

P2P and hybrid P2P architectures are a well-researched field [15][39][40][14]. Since resources are naturally added to the system when more clients connect, the resource-addition technique of scalable networks is inherent in the architecture, making it an appealing prospect because we can add consumption-reducing techniques into the game design to make a very scalable system. According to [39] the different consumption reducing techniques are:

- World partitioning
- Distributed hash tables
- Multicasting
- Fully-connected neighbours
- Neighbour-list exchange
- Mutual notification

Spatial Partitioning

World partitioning is similar to zoning, where the VE is partitioned into zones where players will most likely be interested in what is going on in that zone. These zones can be rectangular, tree-based, hexagonal and polygonal (i.e. Voronoi). A master node, sometimes known as the super-peer, manages the region and the objects within it. The inherent problem with this technique is deciding how large or small the region must be in order for it to most effectively reduce resource consumption and the handling of player crowding in zones, known as load-balancing. To combat this, some solutions dynamically partition the world so that a single zone is never overloaded. SimMud [17] uses this approach by partitioning the clients into regions and having them form an interest set for that region within which they communicate locally.

Distributed Hash Tables (DHTs)

DHTs aims to make the lookup of connections to other peers efficient through the use of hash table-like functions in conjunction with load-balancing of the peers as well as connecting them through a logical overlay. The issues involved are the fair distribution

Stellenbosch University https://scholar.sun.ac.za

of load amongst the peers and latency of the messages travelling through the overlay across multiple hops to reach the intended client. Pastry [22] uses this approach to route messages from one client, identified by a 128-bit identification number (ID), to another numerically closest to it by using a DHT in less than $\log_{2^b} N$ steps where b is the length of the ID in bits. Chord [21] uses a distributed lookup protocol to map a specific key onto a node in order to efficiently locate a node. The issue with this is that if the nodes are placed far apart in the DHT key space, the high number of hops the message would need to take to reach the desired node results in high latency.

Multicasting

The multicast approach involves groups of peers subscribing to an interest set to which any peer can post a message which is then disseminated amongst everyone subscribed to the set. These sets usually span a world zone or are specific for an entity or object. The problem in this is that a high number of peers can be subscribed to an interest set and when coupled with constantly moving avatars can result in a lot of overhead processing. NPSNET uses multicast networks to partition the environment and then an AoI manager (AOIM) uses this to send messages to these multicast network groups [16]. Whilst this is efficient, the approach breaks down when clients congregate into a single area, called clustering or clumping. [56] utilises three tiers of IM to reduce resource consumption. The first tier sections the world into dynamic zones (world partitioning). Using data from the first tier, the second tier tries to create a protocol-independent match between the client's interest and the environment in the section. The final tier adds protocol dependence, or multicast groups, to determine what information the client will receive. The advantage of this is that the client can receive information from multiple groups with the same underlying filter mechanisms.

Fully-connected neighbours

Fully-connected neighbours connect every peer to every other peer in the world, but with message filtering to decide which messages are sent to whom. Players will send to small groups of players frequently whilst sending to everyone infrequently. This is not a good approach due to the overhead of the connections quickly stacking up to take up crucial resources needed for the processing of updates and events. RING is an algorithm that checks whether clients should be able to see each other or not before sending it. Donnybrook [18] uses this approach with two components to it: the first being that it estimates what the client is focusing on, thereby reducing the frequency of the updates it sends to those clients that it is not focusing on (as discussed in 2.3.1) and secondly, it uses multicast systems designed for the specific requirements of the game it is hosting to reduce the cost of message dissemination.

Neighbour-list exchange

Neighbour-list exchange is when connected peers regularly share information about neighbouring peers that they are connected to in order to decide whether or not to make adjustments to whom they are connected to based on their spatial locations. The peers closest to each other make connections, whilst those that are far away from each other do not connect to each other. The problem with this is that global connectivity is not guaranteed as groups of peers could be too far away from each other as well as the constant

Stellenbosch University https://scholar.sun.ac.za

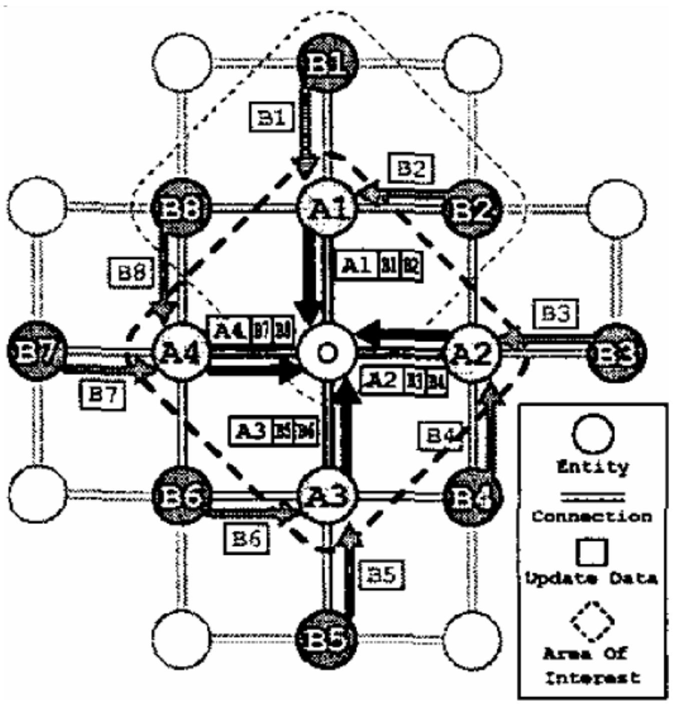


Figure 2.3.1: P2P message exchange where O is the peer, A's are the active entities and B's are the latent entities (source: [5])

neighbour-list exchange resulting in large overhead. P2P message exchange [5] is a system where the peer is connected to a certain number of its closest neighbours, called active entities (AE). Its AE also have neighbours, which are called the peer's latent entities (LE). This can be seen in figure 2.3.1. It regularly updates information about its AE from the information it receives from the AE and uses this to have the whole system be connected. The disadvantage to this is that the system, when uncrowded in certain areas, can cause fragmentation of the overlay.

Mutual notification

Finally, mutual notification is where peers that are near to each other based on a structured geometry (such as a grid). Mutual notification allows for global network connectivity as well as neighbour discovery. This approach has high overhead costs in maintaining the overlay due to high connection change rates that result from continuously moving clients. There are many examples of this approach, such as pSense [57], Solipsis'03 [58], relaxed triangulation [59], Red-Black Delaunay [60] and finally, VON [1] which will be discussed at length in the next section.

2.4 Summary

This chapter described the requirements of an MMVE and the specifics of what they entail within MMVEs. The different architectures that an MMVE can be built on were discussed, giving the advantages and disadvantages of each of them with respect to the requirements discussed beforehand. Finally, solutions to the disadvantages that have

Stellenbosch University https://scholar.sun.ac.za

already been proposed were looked at. The next chapter will discuss the proposed design that this project developed to address the scalability of MMVEs and the respective problems that arise from it.

Stellenbosch University https://scholar.sun.ac.za

Chapter 3

VAST

If the problems of meeting all the requirements of an MMVE feasibly are to be solved, a creative solution that incorporates the most successful aspects of previously-researched approaches needs to be taken. This section will describe the solution that is used to solve the problems presented as well as how possible issues are accounted for. It will discuss VAST and how it works and end off summarising everything discussed.

3.1 Voronoi diagrams

To understand VAST and its components, one needs to know what Voronoi diagrams are. Voronoi diagrams are a well-researched mathematical concept [61]. If there are p peers in a VE (their positions known as a site), the VE will be sectioned into p non-overlapping regions with only a single site within it and where the boundaries of the region outline the area closest to its site, shown in figure 3.1.1. These shapes are simple polygons that can take on any convex shape depending on how the sites are arrayed. The average number of edges per Voronoi cell is less than 6.

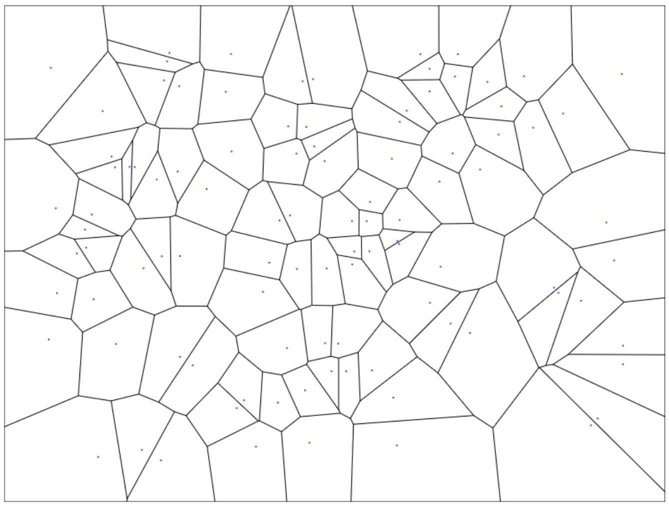


Figure 3.1.1: A Voronoi diagram showing p peers and their corresponding regions

Stellenbosch University https://scholar.sun.ac.za

Using this diagram, *enclosing*, *AoI* and *boundary neighbours* can be calculated. These neighbours are useful for neighbour discovery which is an important aspect of the working of the system. *Enclosing*, *AoI* and *boundary neighbours* are not mutually exclusive so a neighbour can fulfill multiple roles simultaneously. When a site moves or is removed from the Voronoi diagram, the entire diagram needs to be recalculated to get the new regions. When the regions change, the neighbour information can change as well. Since the average number of edges is less than 6 per cell when the number of sites increases, this makes the diagram useful for limiting neighbours.

3.2 VON [1]

VON is the underlying architecture that handles the routing of network traffic and connections to other peers. It is the backbone of the VAST system.

VON is a P2P overlay. This means that no node is more important than any other, but also means that there is no centralized authority to provide a connection point for the other nodes to connect to. Therefore, the first node that joins the system becomes what is known as the *gateway*. It is responsible for providing a port for the other nodes in the system to connect through as well as the initial set up of the system. This set up entails creating a Voronoi diagram with the gateway (itself) inserted into it and the initial server that listens for incoming connections from joining nodes. Every other node uses the gateway port to connect to the system. The gateway, however, does not handle every joining procedure, but rather finds the closest peer to the joining peer's initial location and offloads the join request to that peer. The joining procedure is explained further in section 3.2.2.

The spatial location of the VON peers is purely in relation to the networked virtual environment (NVE) and not the physical location of the peers' hosts. Each node in the system constructs and maintains its own local Voronoi diagram using the spatial coordinates of all of the peers that are within its AoI. These neighbours are classified as boundary, enclosing and/or AoI neighbours.

3.2.1 Neighbour discovery

The purpose of the Voronoi diagram, as explained in section 3.1, is for neighbour discovery. A VON peer is only connected to its enclosing, boundary and AoI neighbours. The reason for this is because a client in a VE for the majority of the time only needs to interact with the clients and objects surrounding it. These objects fall into their AoI by definition and therefore the enclosing and boundary neighbour regions will contain these objects. Its Voronoi diagram will therefore only contain its neighbours who are the VON peers it is connected to. When it needs to communicate with objects that are not near it, the messages can be routed through its neighbours. Figure 3.2.1 shows the global view of the Voronoi diagram (left) and a peers' local view (right) of the same system. We can see how the positions of the VON peers are the same as the global view of the system, but the number of VON peers it sees is dramatically less than how many are in the system. On average, each VON peer has less than 6 enclosing neighbours. This allows for the peer

Stellenbosch University https://scholar.sun.ac.za

to have fewer connections and therefore decrease the amount of resources it needs to have a full view of its surroundings.

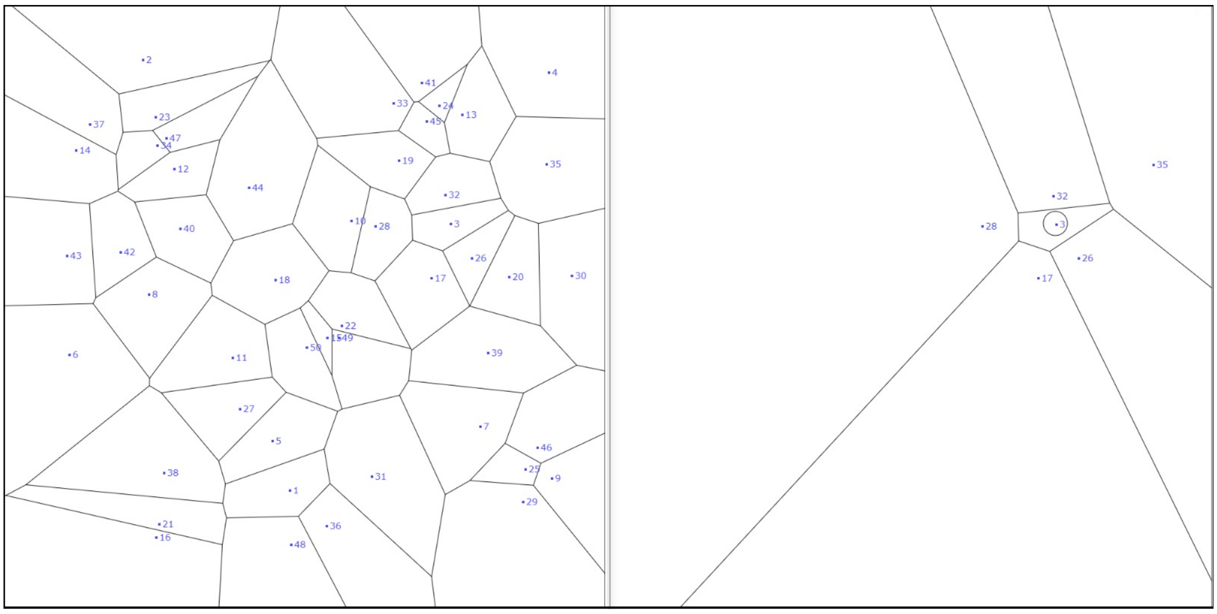
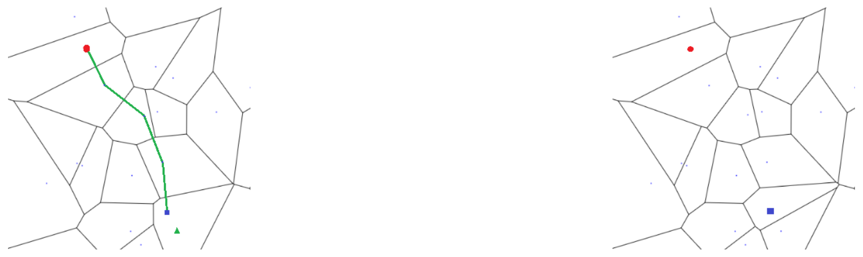


Figure 3.2.1: Figure showing the global view of the system versus what an individual peer sees

VON peers use *mutual notification* to keep track of who their neighbours are and therefore whom they stay connected to. This means that whenever a VON peer moves, the Voronoi diagram is recalculated and neighbour discovery is recalculated. This movement is communicated to all of the peer's neighbours. If the receiver of the movement update is a boundary neighbour then it will check to see whether it thinks that one of its neighbours should know about the moving peer. This is done by seeing whether a peer whose AoI previously did not overlap with the neighbour peer's Voronoi region now overlaps with the Voronoi region. If it does then it will notify the neighbour about the overlap and that peer will then check to see if the moving peer is indeed a valid neighbour or not. If it is a valid neighbour, it makes a connection to the peer and if not, it discards the information. Similarly, if the moving peer results in another peer not being an AoI, boundary or enclosing neighbour then it will disconnect from that peer. This connection/disconnection procedure requires a list of enclosing neighbours to be retrieved from the peer receiving the connection/disconnection and then have a list of all of the perceived neighbours of each enclosing neighbour sent to them to check whether they have the correct view or not. These enclosing neighbours then send back nodes that it thinks the original peer should know about as a result of the shifting of the Voronoi diagram due to the connection/disconnection. The peer then contacts the new neighbour and the neighbour discovery is complete.

Connection and communication between peers use the TCP protocol. Each client has a TCP listening server with a unique port with which it can receive incoming connections from a peer. A TCP socket is established between peers using the peer's unique port when it is established that they are neighbours. All communication happens between the

Stellenbosch University https://scholar.sun.ac.za


(a) Greedy forwarding of join request to closest peer

(b) Joining peer inserted into Voronoi diagram

Figure 3.2.2: The join procedure and its outcome

client over this connection. If a peer crashes or disconnects, the resultant timeout or error on the TCP socket tells the peer to disconnect from the client and remove it from its local view. The Voronoi diagram is then recalculated and neighbour discovery is redone. This allows the system to be robust against disconnections and peer failures.

Next, the VON application protocol interface (API) will be discussed. This is the protocol used between VON peers to communicate with each other. We will discuss the **JOIN**, **MOVE**, and **LEAVE** procedures.

3.2.2 JOIN procedure

The following list describes the **JOIN** procedure of a VON peer (that is not the gateway).

1. The joining peer sends a message to the *gateway*, informing it of its existence and requesting a new ID
2. The *gateway* assigns an ID to the *joining peer* and informs the *joining peer* of its new ID
3. The *joining peer* receives an ID and inserts itself into its local Voronoi diagram
4. The joining peer queries the *gateway* to find out who the *closest peer* to the *joining peer's* initial position is
5. The *gateway* offloads the join request to the *closest peer* from its own local view. The *closest peer* then tries the same procedure to find the *closest peer* to the *joining peer's* position and forwards this peer the **JOIN** request. This continues until the *closest peer* finds itself in a greedy forwarding manner (shown in figure 3.2.2)
6. The *closest peer* then handles the **JOIN** request that has been forwarded to it
7. The *closest peer* inserts the *joining peer* into its Voronoi diagram and calculates the *joining peer's* neighbours from the *closest peer's* own view
8. The *closest peer* sends a list of nodes that it thinks are the *joining peer's* neighbours to it and disconnects from any neighbours that are no longer its own neighbours as a result of the insertion

Stellenbosch University https://scholar.sun.ac.za

9. The *joining peer* inserts the new information about possible neighbours into its Voronoi diagram and calculates relevant neighbours, discarding information about neighbours that it deems irrelevant to it

10. The *joining peer* contacts the *new neighbours* and sends them their perceived neighbours that they should be connected to for their own neighbour discovery

11. The *new neighbours* acknowledge the *joining peer's* hailing message and send it any neighbours that it should know about

12. The *joining peer* repeats the process from step 9 until there are no new neighbours to connect to

The only differences that the gateway follows are in step 1. The gateway immediately sets its own ID instead of sending a message to itself and in step 4 the joining peer immediately marks itself as joined and does not follow from step 5 onwards as there are no neighbours to find when the gateway starts the system.

3.2.3 MOVE procedure

This section will describe the **MOVE** procedure of a VON peer:

1. The *moving peer* determines whether its movement overlaps with another client. It does this so that the correct number of Voronoi cells are calculated. Shift the *moving peer's* position if it does

2. Update the local Voronoi diagram and local position

3. Recalculate neighbours and send movement event to them

4. Send all enclosing neighbours a list of neighbours that the *moving peer* thinks they should be aware of. Neighbours check info, update their list of neighbours and send back neighbours they think we should know about

5. Neighbours that receive the **MOVE** request check for any overlap with any of their neighbours and adjust slightly if it does

6. Neighbours check for any neighbour changes and send these to the *moving peer*. These neighbours then disconnect from any nodes that are no longer their neighbours as a result of the movement

7. The *moving peer* inserts these neighbours into its local view and sends a list of neighbours it thinks they should know about and they return with neighbours that they think the *moving peer* should know about

Something to note in both this **MOVE** procedure and the **JOIN** procedure is that whenever a peer sends a list of neighbours to another peer that it thinks that they should know about then they always respond by sending back a list of neighbours. This is mutual notification in action and is what ensures a consistent and accurate view of a peer's correct neighbour.

Stellenbosch University https://scholar.sun.ac.za

3.2.4 LEAVE procedure

The **LEAVE** will now be detailed below.

1. *Leaving peer* sends **BYE** message to all of its neighbours
2. These neighbours send out a consistency check to all of its neighbours except for the *leaving peer*. This consistency check is a call to the other neighbours to check their perceived neighbours with those around them to ensure that they have the correct view
3. Neighbours remove the node from their Voronoi diagram and disconnect the TCP sockets connected to it
4. The *Leaving peer* then reinitialises itself and awaits reconnection to the VON layer

This **LEAVE** procedure happens when a node intentionally leaves the VON whilst there are still peers in the overlay. If a peer erroneously disconnects or crashes, this sequence is not followed and instead, the neighbours will sense something wrong with the socket and disconnect the peer automatically.

In the next section, the VSO with spatial publish/subscribe will be discussed.

3.3 Voronoi Self-organising Overlay and the entry server

The VSO layer sits above the VON layer in the VAST stack. It is the part of the system that manages client connections and update and event dissemination. Each VAST node consists of a *VON peer* and a *VSO peer* as seen in figure 3.3.1. The *VSO peer* uses an SPS system to control the information flow to and from clients. It matches publications that a client makes to subscriptions that have been declared and is why *VSO peers* are also known as matchers.

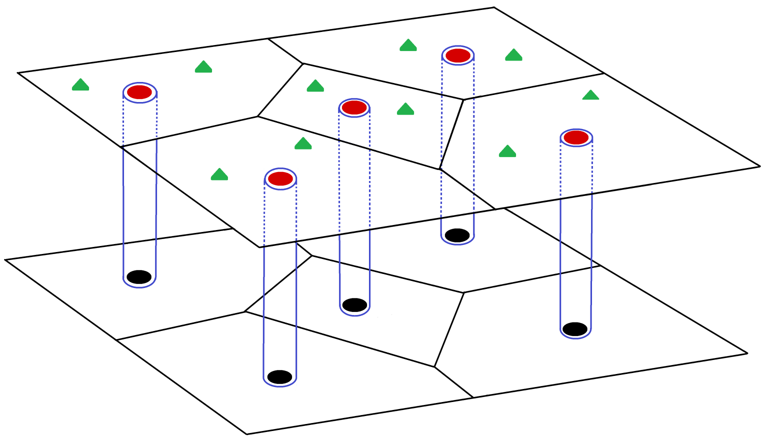


Figure 3.3.1: The VAST stack (○) with *matchers* (●) in the top layer handling *clients* (▲) whilst *VON peers* (●) handle network routing and connection

Stellenbosch University https://scholar.sun.ac.za

3.3.1 Architecture of VSO

The VSO layer consists of a collection of matchers each with their own virtual position. They are arranged with a Voronoi diagram similar to that of the VON layer so that they are aware of which part of the VE is under its jurisdiction and who its neighbouring matchers are. The position of the matcher is therefore mirrored by the VON peer as the VON peer is responsible for ensuring that the matcher is connected to its neighbours in the networking layer. If the VON peer discovers or disconnects from a neighbouring peer, then so does the matcher. This ensures that this layer does not need to do any processing concerning neighbour discovery and therefore improves its performance.

Matchers

The VSO layer uses an SPS system to manage update dissemination. The primary role of a matcher is therefore to handle subscriptions and publications made by clients in the system that falls within their Voronoi region.

When the first matcher is created, it records its message handling function with the VAST node's message handler (the same one used by the VAST node's VON peer). It then informs the ES of its ID and initialises its internal states, which include inserting itself into its local Voronoi diagram, determining its boundary regions and updating its local state with this information. These boundary regions are discussed further below, but their role is to create a buffer zone that prevents client connection thrashing.

Once this is done, the matcher marks itself as having joined the layer and the connection sequence is complete and the matcher is ready to receive connections from clients. If another VAST node joins the system as a neighbour to the current VAST node then the VON peer informs the matcher of this new neighbour and the matcher inserts the matcher into its own Voronoi diagram and stores its state in a list of neighbours. This allows it to propagate subscriptions and publications to the new neighbour, as well as transfer clients to it if the client moves out of its region.

The Entry Server

When a client connects to the VSO layer, it does so through the ES. In a practical implementation of VAST, there would be multiple ES servers located geographically far apart and connected with high-speed networks. The role of the ES is to route information between the correct client-matcher pairs and assist with load balancing. The reason that it routes information from the clients to the matcher is to keep the API simple and to sequester the client from any connection switching. This has scalability issues however as it can create a bottleneck within the system. It is chosen to be this way so that the API is the same as the C++ version of VAST. In future implementations, the API will be adjusted to allow the client to handle its own connections to the matcher it is communicating with, thereby alleviating the bottleneck.

The ES essentially acts as a super-peer. It stores the positions of the matchers in a Voronoi diagram and when a client connects with a position in the VE, the ES checks it against the Voronoi diagram to see which matcher's region it is contained in. It then

Stellenbosch University https://scholar.sun.ac.za

connects the client to the matcher and continues to route all the client information to this host matcher (shown in figure 3.3.2). If a client moves out of one region and into another region, then the matcher notifies the ES and it changes the routing of the information to the new region's matcher and relieves the old matcher of its connection to that client.

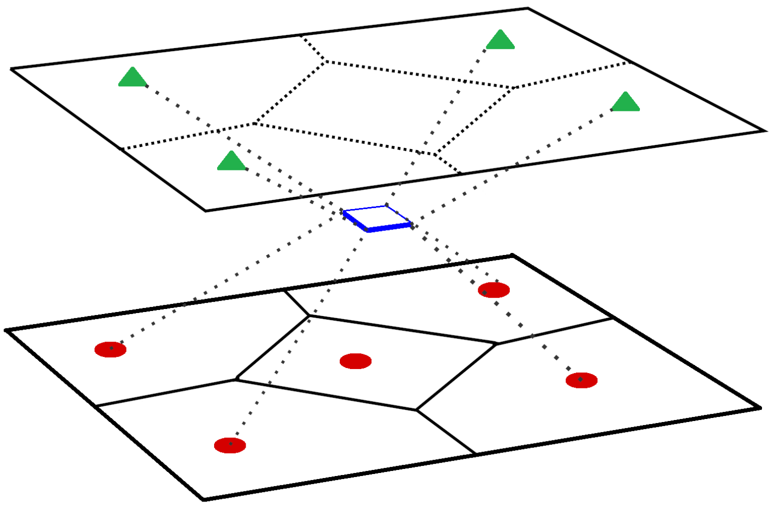


Figure 3.3.2: A figure showing the routing of information (- - -) from clients (▲) through the entry server (□) to their respective matchers (●) based on position

The calculations of how the matcher and ES determine whether a client is within a certain region is shown in the pseudo-code below in algorithm 1. To do this calculation, an array of the vertices x and y coordinates are needed. This array is denoted by V. The position of the client is also needed. This is denoted by **P**. The idea behind this algorithm starts with drawing a ray from the point in question in a certain direction that passes through the polygon. If the ray crosses an edge an even number of times, then it is originally outside of the polygon and if it crosses an odd number of times then it originates within the polygon [62]. This is shown in figure 3.3.3

Algorithm 1: Point within a region algorithm

```
1 int $c$ = 0 // This is the counter for how many crossings have happened
2 int $n = V$.length
3 for $i = 0; j = n - 1; i < n; j = i++$ do
4     if $((V[i].y > \boldsymbol{P}.y) \mathrel{!}= (V[j].y > \boldsymbol{P}.y))$ &
      $(\boldsymbol{P}.x < V[i].x + \frac{(V[j].x - V[i].x)(\boldsymbol{P}.y - V[i].y)}{(V[j].y - V[i].x)})$ then
5         $c = !c$ // alternate between even and odd for every crossing
6     end
7 end
8 return $c \mathrel{!}= 0$
```

Stellenbosch University https://scholar.sun.ac.za

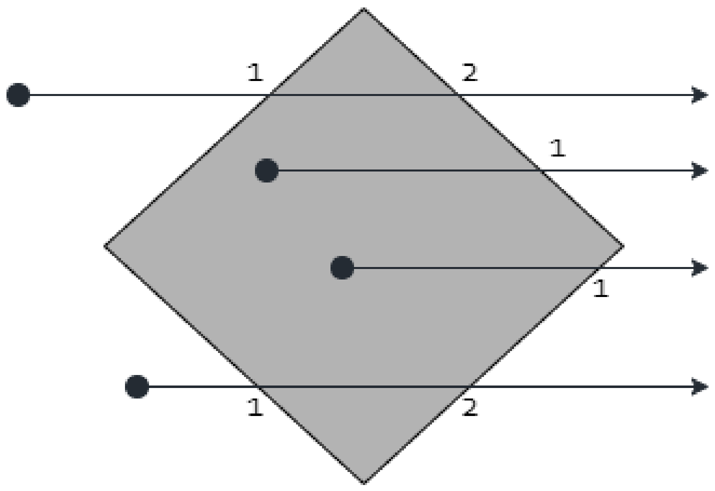


Figure 3.3.3: Figure showing ray casting of points through a polygon and their relative line crossings

Thrashing and boundary regions

Something to note here is that a client could cross the threshold of two regions multiple times in quick succession which results in thrashing (seen on the left of figure 3.3.4). This is solved by creating a buffer zone that extends a small distance past the region boundary that the client needs to cross in order to fully disconnect from the matcher whose region the client just exited and connect to the matcher whose region it is entering (Seen on the right of figure 3.3.4). This space prevents the client from connecting and disconnecting rapidly in a short period of time.

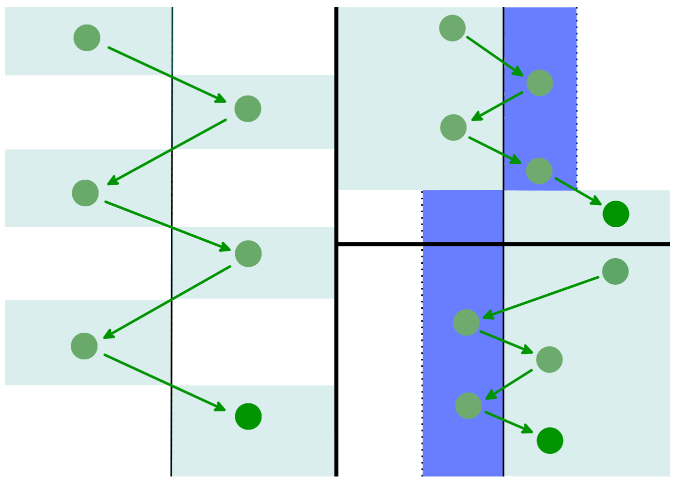


Figure 3.3.4: An example of thrashing caused by a client (●) moving between matcher regions, alternating host regions (🟩) (left). The solution is to have a buffer region (🟦) that extends beyond the boundary of the region (right)

Stellenbosch University https://scholar.sun.ac.za

Load Balancing

Matchers can only handle a certain amount of clients before reaching their scalability limits. A threshold is therefore set to ensure good performance of the matchers in the VSO layer. The managing of clients to prevent matchers from being overloaded is called *load balancing*. If a matcher has reached this threshold, the next client that connects to the matcher with a position that falls within an overloaded matcher's region is removed from the normal connecting sequence by the matcher and put into a queue of pending clients. The matcher then sends out a request for help to the ES. The ES calculates the positions that the neighbouring matchers need to move to and tells them to shift to these positions. This results in the overloaded matcher's region shrinking and therefore transferring clients close to the boundary of the region into their own region, thereby relieving the overloaded matcher. If no clients are transferred, the matcher continues to request help from the ES until the load is relieved. The matchers continually run neighbour discovery during this procedure to ensure that they stay aware of new or expired neighbours. When the load is relieved, the pending client is again allowed to continue with its connecting sequence.

There is a scenario that is common in MMVEs and MMOGs called clustering or flocking. This is where an event or point of interest draws players to a single spatial location within the world. If this happens, the above load balancing would fail as the combination of the number of subscriptions and clients within a small region would overload the resource mitigation techniques of VAST. What the ES does to counteract this is after matchers move a predetermined amount of times and the overloaded matcher is not relieved, then a backup matcher or new matcher joins the overlay close to the location of the overloaded matcher. This can be done numerous times until all of the overloaded matchers are relieved.

When a client connects to the ES, it first lets the ES know of what type of connection (client/server) it is since the ES is agnostic to this otherwise. Once acknowledged by the ES, the client sends a **JOIN** request with its position. The ES assigns the client a unique ID and then checks which matcher has jurisdiction over the region that the client is in. If load balancing needs to be done then the client halts its connection procedure until the ES completes the load balancing. Once authorised to continue, the ES maps the connection between the client and its matcher. The matcher is then notified of the incoming client and communication between the client and matcher is routed through the ES.

3.3.2 Spatial publish/subscribe (SPS)

The VSO layer uses SPS to efficiently route information between clients. As explained in section 2.3.1, the idea behind SPS is that entities are only interested in a certain spatial area within a VE. Any event or update that happens in the VE gets published with spatial coordinates attached to it. If these point or area publications intersect a subscription region, then the owner of the subscription region receives the publication.

In the realm of VSO, matchers are the custodians of subscriptions and publications. Every client needs to own at least one subscription in order to interact with the elements and entities within the VE. When the client subscribes to an area, it gives its host matcher a description of the subscription. This information is shown in table 3.3.1. The matcher

Stellenbosch University https://scholar.sun.ac.za

takes the subscription info and checks whether any other matchers' regions intersect with the subscription. The way this is calculated is shown below in equations 3.3.1- 3.3.4. This formula was derived by Paul Bourke [6]. Figure 3.3.5 shows the basic scenario that we are presented with in this case. $\mathbf{P}_1$ and $\mathbf{P}_2$ can be seen as vertices of a Voronoi section and $\mathbf{P}_3$ as the publication point.

Table 3.3.1: Contents of the "Subscription" data type

Field	**Data Type**	**Mask (#-number, $-string)**	**Purpose**
Host ID	Number	Matcher ID (#)	The current host matcher of the sub
ID	String	Host ID + '-' Client ID (#-#)	The Client's ID
Subscription ID	String	Host ID + '-' + ClientID + '-' + Channel (#-#-#)	The unique ID of the subscription that is a combination of the host ID, client ID and channel
Channel	String	Any name ($$$)	The specific channel that the client wants to communicate on
AoI	Number	Number (#>0)	The area of interest of the subcription
Time	Number	In milliseconds (#)	The last time this subscription was updated
Type	String	client' or 'server' or 'unknown'	The type of owner of the subscription
Username	String	Any name ($$$)	The display name of the client

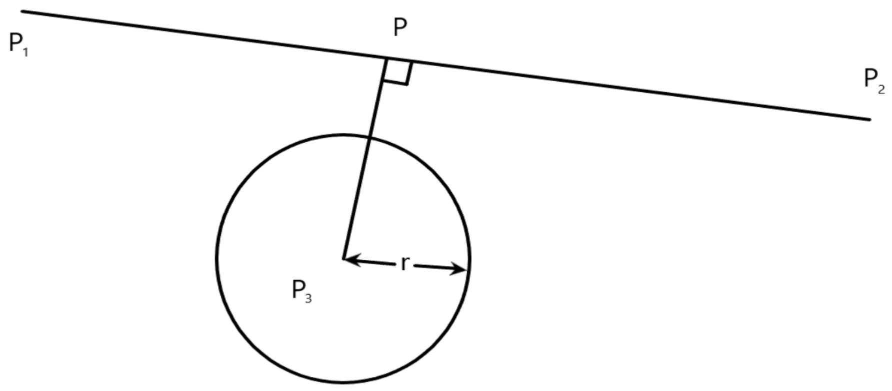


Figure 3.3.5: Calculation of the intersection between a line and a circle [6]

The equation of a line is given in equation 3.3.1. The second deduction is that if there is an intersection between the line and the circle then the intersection point ($\mathbf{P}$) between $\mathbf{P}_1$ and $\mathbf{P}_2$ to the point $\mathbf{P}_3$ is along a perpendicular line, shown in equation 3.3.2. Therefore:

$$lineequation: \ \mathbf{P} = \mathbf{P}_1 + u(\mathbf{P}_2 - \mathbf{P}_1) \tag{3.3.1}$$

$$(\mathbf{P}_3 - \mathbf{P}) \cdot (\mathbf{P}_2 - \mathbf{P}_1) = 0 \tag{3.3.2}$$

$$[\mathbf{P}_3 - \mathbf{P}_1 - u(\mathbf{P}_2 - \mathbf{P}_1)] \cdot (\mathbf{P}_2 - \mathbf{P}_1) = 0 \tag{3.3.3}$$

$$u = \frac{(x_3 - x_1)(x_2 - x_1) + (y_3 - y_1)(y_2 - y_1)}{(x_2 - x_1)(x_2 - x_1) + (y_2 - y_1)(y_2 - y_1)} \tag{3.3.4}$$

Equation 3.3.4 shows a calculation for u, a scalar value. The value of u is the scalar value that scales $\mathbf{P}_2 - \mathbf{P}_1$ to the magnitude of $\mathbf{P} - \mathbf{P}_1$. If u is between 0 and 1 then the closest point is between $\mathbf{P}_1$ and $\mathbf{P}_2$, but if it is not then we have to check for the intersection between the circle and the points $\mathbf{P}_1$ and $\mathbf{P}_2$. To do this, the distances between $\mathbf{P}_3$ and $\mathbf{P}_1$ and $\mathbf{P}_2$ are respectively calculated and is compared to the radius of the subscription.

Stellenbosch University https://scholar.sun.ac.za

If the distance is less than or equal to the radius for either of the distance calculations, then there is an intersection. If not, then one can be certain that there is no intersection between the subscription and another region.

If the subscription does intersect a different region then the matcher propagates the subscription to the other relevant matchers. This is an important step as even though the other matchers are not the client's host matcher and therefore do not have a connection to the subscribing client, if a publication within their region falls within or intersects the subscription then it would need to propagate the publication to the client's host matcher. The host matcher would then send the propagated publication to the client.

An example of why this is necessary follows: if a client were to stand right next to the boundary of a matcher region and there was no publication propagation, then something could happen right next to it and the client would be oblivious to it because it would not receive the event/update. The client would have both a negatively impacted experience as well as be aware of the underlying architectural boundaries which is something to be seriously avoided. The client should be unaware of the underlying workings of the architecture to preserve a smooth gameplay experience.

The subscriptions and publications must be robust and work as seamlessly as a single C/S architecture. Subscriptions need to be on the correct matchers and in the correct position at all times. When a matcher joins the system, the matcher's neighbours compare the subscriptions that they have on record with the new region that the joining matcher owns within the Voronoi diagram. If the subscription AoI intersects with the region of the joining matcher, then the subscription is propagated to the joining matcher.

When the client leaves the system or unsubscribes from the SPS layer, the matcher removes the subscription from its records and, if it is the host of the subscription, sends out a message to other matchers that have a record of the subscription that tells them to remove their records of the subscription as well.
In the next section, the rest of the VAST API will be discussed, namely the **JOIN**, **MOVE**, **SUBSCRIBE**, **UNSUBSCRIBE**, **PUBLISH** and **LEAVE** procedures.

3.4 VAST API

The VAST API is the API that the client will use to interact with VAST. It is separate to the inner APIs that VON uses to communicate with each other.

3.4.1 JOIN procedure

The first procedure is **JOIN** that the client uses to join the VAST system.

1. The client sends a **JOIN** request with its username, client type and its position and AoI to the system through a specific port that the entry server is listening to.

2. The entry server determines which matcher's region the client's position falls under and stores this relationship. It also states whether this is a new connection. The ES then internally assigns a client ID and forwards this information to the matcher.

Stellenbosch University https://scholar.sun.ac.za

3. The matcher checks how many clients it is handling against its maximum client threshold. If it is above the threshold then the load balancing procedure described in section 3.3.1 is followed until the load is relieved.
4. If it is lower than the threshold then it notes the client's attributes and sends an acknowledgement to the entry server.
5. The entry server forwards the acknowledgement to the client, informing it that it has joined the system and may now move and subscribe.

3.4.2 SUBSCRIBE procedure

The next procedure discussed is the **SUBSCRIBE** procedure. This method allows clients to join the SPS layer and receive publications from other clients that are already subscribed and publishing. These subscriptions can be further layered by the idea of channels. A channel is a way to further finesse the SPS system by layering the communication into different "rooms" or "channels". Subscriptions can only receive publications that match the channels that they are subscribed to.

1. The client sends a **SUBSCRIBE** request to the entry server with its client ID, the channel it wishes to subscribe to, its username, and its position and AoI which gets forwarded to the matcher.
2. The matcher checks if the subscription has already been made before and if it has, just updates the existing subscription.
3. If it does not exist then it stores the subscription in a map of subscriptions.
4. It then checks whether the subscription overlaps with any other matchers' region so that it can propagate it to those matchers.
5. The receiving matchers then follow the same procedure from 2, ensuring not to propagate to matchers that already have the subscription.

3.4.3 MOVE procedure

After subscribing the client unlocks the ability to move within the system. The **MOVE** procedure is now discussed below. This also includes what happens if a client moves outside of the matcher's region and a client transfer to another matcher is needed.

1. The client sends a **MOVE** request with its client ID, username, position and AoI, the channel it wants to publish the movement on and the information of the MMVE packet that describes the movement to the ES (usually in the form of a byte buffer), who forwards it to the matcher.
2. The matcher updates its internal view of the client's position and then forwards the information to the matcher.
3. The matcher takes this information and generates the ID of the subscription that belongs to the client. It uses this to check whether the client has subscribed and if it has not then it discards the move information.

Stellenbosch University https://scholar.sun.ac.za

4. The matcher checks if the movement takes the client past the boundary region that describes the matcher's jurisdiction. If it is outside of its boundary region then it requests a client transfer.
   (a) The matcher deletes the information it has pertaining to the client and sends a transfer request to the new matcher as well as a subscription update to the new matcher telling it that it now owns the moving client's subscription.
   (b) The ES removes the connection between the moving client and the requesting matcher and re-routes the connection to the new matcher.
   (c) The new matcher receives the connection and treats it like a client connecting to the system, sending the successful join acknowledgement to the ES which signifies the successful transfer of the client.
5. If is within the region, then the matcher updates the information of the relevant subscription.
6. The matcher then checks for an intersection between the subscription AoI and its own regional boundaries to see whether the subscription update needs to be propagated.
7. If the subscription AoI newly intersects another matcher's boundary then the subscription gets propagated to that matcher. This subscription may only be propagated if the matcher owns the subscription. If the intersected matcher was already aware of the subscription, then it just updates the subscription with the new information.

3.4.4 PUBLISH procedure

The client now has the ability to move its subscription around within the system and receive publications through this. It can also publish to the SPS and this procedure is described below.

1. The client sends a **PUBLISH** request to the ES with its client ID, username, position and AoI, channel and MMVE packet information.
2. The ES finds the matcher that the client is connected to and forwards this information to it.
3. The matcher runs through the subscriptions it is aware of and checks whether the subscriptions match the right channel and whether the publication radius intersects the subscription AoI. If it does not intersect then that subscription is ignored.
4. If it does match then the matcher determines if it owns the subscription or not (if the position of the subscription is within its region). Two things can happen depending on this fact:
   (a) If it owns the subscription, then it checks that the subscription does not belong to the publishing client and if it has a connection to the client whose subscription intersects the publication. The valid clients are sent the publication.

Stellenbosch University https://scholar.sun.ac.za

   (b) If it does not own the publication then it propagates the publication to the matcher who owns the subscription for them to send to the client whose subscription it is.

5. The propagated publication gets handled by the matcher in the same way until all of the relevant clients have received the publication. No acknowledgements are sent to the ES so that bandwidth and computational costs are minimised.

3.4.5 UNSUBSCRIBE procedure

If a client wishes to remove itself from the SPS layer, then it can send a **UNSUBSCRIBE** request todo so. This procedure goes as follows:

1. The client sends an **UNSUBSCRIBE** request with its client ID, channel and username to the ES.

2. The ES finds the relevant matcher and forwards the information to it.

3. The matcher checks whether it has a subscription that matches the client's **UNSUBSCRIBE** request. If it does not, then the procedure ends here and nothing happens.

4. If the matcher owns the subscription then the **UNSUBSCRIBE** request is propagated to all of the known matchers who have a record of the subscription.

5. The matcher then deletes the subscription record.

3.4.6 LEAVE procedure

Finally, if the client wishes to leave the system entirely, a **LEAVE** request is sent and the client terminates its connection to the system. This sequence is described below:

1. The client sends a **LEAVE** request with its client ID to the ES.

2. The ES determines which matcher is linked to the client and sends the **LEAVE** request to the matcher. It then terminated the connection to the client.

3. The matcher finds the subscriptions relating to the leaving client and removes them. If it is the host of the subscription then it propagates the removal of the subscription to the matchers who have a record of the subscription.

4. The matcher finally removes all local information about the client and the client is considered to have left the system.

All of these procedures and their arguments can be seen below in table 3.4.1.

Stellenbosch University https://scholar.sun.ac.za

Table 3.4.1: The VAST API [4] [7]

Command	Arguments	Function
JOIN	Username (String) Client type (String) X position (Number) Y position (Number) Radius (Number)	Connects client to the VAST system
SUBSCRIBE	ClientID (String) Channel (String) Username (String) X position (Number) Y position (Number) Radius (Number)	Subscribes client to a certain area and channel which inserts it into the SPS layer
MOVE	ClientID (String) Username (String) X position (Number) Y position (Number) Radius (Number) Channel (String) MMVE Packet (Byte array)	Moves client's subscription's position and changes its AoI
PUBLISH	Client ID (String) Username (String) X position (Number) Y position (Number) Radius (Number) Channel (String) MMVE Packet (Byte array)	Sends client message to other intersecting subscriptions
UNSUBSCRIBE	Client ID (String) Channel (String) Username (String)	Unsubscribes a client from the SPS layer
LEAVE	Client ID (String)	Disconnects a client from the VAST system

Stellenbosch University https://scholar.sun.ac.za

3.5 Implementation challenges and considerations

This section will discuss the design challenges and considerations faced in developing and implementing this system and how they were solved.

3.5.1 Migrating from C++ to JavaScript

VAST was originally written in C++ by Hu [2] but was migrated to JavaScript (JS) in this project. The reason for this is that, with the creation of Node.js between when the system was first created in 2005 and now, the functionality of having VAST being implementable in both browsers and on the server-side of the browser in Node.js gives VAST a much wider practical applicability than any other language.

Migrating from C++ to JS resulted in some issues. JS is a dynamically typed language as opposed to C++ being statically typed. JS does not define strong types when using variables and this required a strong testing methodology (unit and integration tests) in order to ensure that each part worked as intended before using it in the larger system.

Working with large scale systems

One of the issues of working with scalable systems is the debugging of such systems. When there are hundreds of nodes running asynchronously in a system and an error occurs, it can be difficult to pinpoint exactly where and how to fix such a problem. To solve this, a custom logger that allowed the log output of a single node in the system was developed. This allowed finer control over the information that was captured to debug problems that arose. A visualiser was also created to display the global view of the VON and VSO layer, as well as the individual views of the VON peers in order to visually verify the neighbour discovery was working correctly. Incorporated into this is also a debugging tool that allowed the controlling of when messages were sent. This allows the user to choose when to send individual messages as well as which clients do this. This assisted in the debugging of scenarios where the specific erroneous client was identified.

Another issue with large scale systems is the reproducibility of errors. When there is a large system with tight timings, the smallest difference in computing time due to fluctuations in processor performance or other such things can result in a different result for the same setup. This can make reproducing errors very difficult as the scenario cannot be accurately replicated. The solution applied in these scenarios was to try and determine the manner of the error and what the possible cause was and then try to slow down the timing of the communication between peers as well as try have as small of a scale as possible whilst still causing the error.

Finally, when working with large systems where it systematically has to cycle through long lists of items in order to complete an action (eg. iterating through subscribers to test for publication validity or through matchers to look for subscription propagation targets), it is of the utmost importance to either keep the lists as short as possible or to increase the efficiency of the search. VAST's neighbour discovery's primary role targets this problem. It aims to keep the list of neighbours as short as possible. But it still needs to use an efficient way of iterating through and retrieving these neighbours. For

Stellenbosch University https://scholar.sun.ac.za

this reason, computations are kept to a minimum within loops and are used as sparingly as possible. Hash maps are also used over arrays to speed up the accessing of specific values.

Fragmentation of Voronoi diagram

There is a scenario where VAST can fragment into two halves where one half is unaware of the other half and there is no way for the nodes within the system to mend the rift between them. Whilst this is a serious issue, it was only encountered once throughout thousands of tests and thus was not handled due to the cost-benefit of implementing the solution versus how many resources it would take to implement a solution. However, if a solution were to be implemented, it would be an infrequent check done by all the nodes to the gateway periodically that asks the gateway which nodes it thinks are closest to it. A greedy forwarding of such a request (similar to the querying described in section 3.2.2) could be done and once the endpoint is reached, the connection could be established if it is not already there, resulting in a mended Voronoi diagram.

3.5.2 Overlapping clients and VAST nodes

Whilst the Voronoi diagram is a very powerful tool, when two or more nodes or clients are on the same spot the Voronoi diagram creation algorithm breaks down as it cannot split the space between these points. To rectify this, a small random offset with a magnitude of 10^{-1} is given to the peer that moves onto another peer in order to give them a small difference in position. This difference only affects the local slave copies and not the master copies of the peer states. This results in a small difference in perceived location but not large enough to make a meaningful difference to neighbour discovery or player experience.

3.5.3 Consistency and neighbour discovery

Consistency in the view of a VAST node relative to its neighbours has several complications. Whenever a node connects, leaves or moves a consistency check needs to be performed every time. This consistency check can be resource costly (described in section 3.2.1) and therefore cannot be done all of the time due to the asynchronous nature of these events. Similarly, the outdated neighbours need to be removed, which requires iterating through neighbours and checking whether each one is still a neighbour or not. There is also still some inconsistency that appears after much activity by nodes and so a periodic function is set up that ensures consistency through these two checks.

3.5.4 Robustness

In a fluid system such as VAST, it is imperative that the system is robust against failing nodes. This proves quite a challenge when there are numerous asynchronous events and updates, as a single unhandled error can result in a chain reaction of failing nodes. The system, therefore, had to be designed to be impervious to such events.

Stellenbosch University https://scholar.sun.ac.za

3.5.5 Load balancing

Load balancing is a problem that every MMVE with a distributed architecture faces. When a matcher is overloaded, the question that is faced is two-fold: does one move a matcher closer to the overloaded matcher or should an additional matcher be added? If the matcher is moved, then one needs to decide how much it should move per load balancing tick. If the movement is too large, then there can be large changes in client connections and can lead to erratic behaviour in the system. If the movement is too small, then the overloaded matcher may not be relieved quickly enough and can lead to clients being disconnected from the system for too long.

In this implementation, the matchers take the distance between them and divide it by a predetermined scalar value to get the distance that the matcher needs to travel in that tick. This scalar value was practically tested at multiple values and was finalised to be 20. This means that every time the matcher needs to shift, the distance between the matchers is divided by 20 to get the delta coordinate values. What was observed is that the matcher moves a sufficient amount when this scalar value is chosen so that the client is disconnected for less than 100ms. The load balancing check is done whenever a client moves and therefore is tied to the movement tick rate of the clients connected to the system.

One issue that was observed and posed a challenge to rectify is the issue of matchers converging but never diverging, and consequently causing the Voronoi regions to become erratic as the distance between matchers became infinitesimally small. This resulted in matchers becoming overloaded with no possible way to relieve them. This was rectified by adding a minimum distance that the matchers had to stay away from each other, as well as by implementing a timed interval that allowed a matcher to check whether it was still overloaded and if it was not, to tell its neighbours to slowly back away from it towards their original positions. They move to their original positions to ensure that the opposite does not happen where the matchers congregate at the edges of the VE. This both allowed for the Voronoi regions to never become infinitely small or too large as well as to prevent the entire system from converging to a point.

The last issue faced with load balancing was the handling of clients whilst the load balancing techniques were being applied. When the region on the side of the client awaiting connection moved enough, it would result in the pending client's joining position falling within the region it was attempting to leave. This means that instead of just connecting once the overload was eliminated, the pending client would erroneously continue connecting to the matcher since the ES is the one that checks whether the joining position is correct for the matcher. To solve this, the position is checked before the connecting sequence is continued and if it no longer falls within the matcher's region then the ES is immediately notified that it should transfer the client to the correct matcher.

Stellenbosch University https://scholar.sun.ac.za

3.6 Summary

This chapter focused on the design of VAST and how the logic of various functions within the system work. It discussed the background of VAST as well as what it aims to achieve. The VON, VSO and ES layer were discussed in detail with their APIs described and summarised. Finally, the implementation challenges and their solutions were discussed. In the next chapter, a background of Minecraft will be discussed to understand how it works as an MMVE and to understand what design challenges need to be worked around to integrate VAST into the Minecraft structure.

Stellenbosch University https://scholar.sun.ac.za

Chapter 4

Minecraft

Minecraft is set in a VE called The Overworld which is built up of individual blocks of different types. Players can break these blocks and place them elsewhere in arrangements of their choosing, introducing an element of creativity to the game. There are many different components to Minecraft, but only a few of them are relevant to this project and will be focused on. These components are:

- The server
- The world state
- The player entity
- Non-playable character (NPC) entities
- Networking including packets, the Minecraft Protocol and the login sequence
- Interest management in Minecraft

The effect that each of the components has on the scalability will be discussed. The information is summarised from [13].

4.1 The server

The Minecraft server is responsible for processing all events and generating the state updates of a Minecraft world. It is the host for the different states of The Overworld and governs any changes made to them. All traffic from clients goes through the server before any changes are shown on the clients' portal. The server processes this traffic at regular intervals in a game loop called a server tick. The Minecraft server runs at 20 ticks per second, or one tick every 50 milliseconds. Everything that happens in the world is propelled by the tick of the server, except for graphics, which is separately rendered after this update happens. The benefit of this is that the frame rate of a client does not affect the performance of the processing of the game. The server takes all of the updates in the time between ticks (called a frame) and sends these updates to the connected clients for them to process and update their local copies of the states.

The Minecraft server allows for modifications (known as *plugins*) by developers. These plugins allow for the developer to modify the information flowing into and out of the server. Plugins do not modify any game files, but rather add functionality to the server,

Stellenbosch University https://scholar.sun.ac.za

such as player “factions” or implementing leader boards. It is useful for developers as it allows for data packets to be intercepted and altered or parsed for information. This can be useful when trying to modify the server without having to change the server code. Modifications, or “mods”, are similar to plugins except for the fact that they are client-side and modify the game assets instead of the information that is going to the client. For example, new blocks and items with specific functionality can be added to the client. The problem with modifications is that they require every client to have them installed or else this can cause erroneous behaviour for those clients that do not have the specific modification installed.

4.2 World state

The world is defined as the blocks that make up the environment that the Minecraft clients spawn into and exist in. The blocks are placed in a 3D grid using Cartesian coordinate and are spawned procedurally in 16x16x256 sized columns called chunks. When a player spawns into the world, chunks are generated around the player within the user-specified vision radius, known as the AoI. Chunks are generated when a player is interested in it and is unloaded when there are no players interested in it. This loading and unloading of a chunk is dependent on a client’s view distance intersecting the chunk. Only loaded chunks will have entities and other environmental aspects updated in a server tick. This means that as soon as a chunk is unloaded, no further state updates can be processed for anything that resided in that chunk before it was unloaded.

4.3 The player entity

The player entity is what the players control to manipulate the world state. They can break the blocks around them by mining them in order to get that specific type of block into their inventory. They can then place the blocks that they have collected into the world around them to create structures limited only by their own creativity. Whenever the player inputs an action, the client generates a packet that it then sends to the server hosting the world to inform the server of the change it has made to the state of the game. These updates include movement, interface interaction such as inventory management or chat messages, combat actions and block manipulation.

The most important of these updates is movement, as that is how the server knows where in the world the client is and therefore how it reports this location to other entities and players in the world. When the client spawns into the world, it is given a specific spawn location. The server sends an initial orientation packet to finalise the spawning of the client. Any movements thereafter are performed by the client. These movements generate packets that are sent to the server as soon as the movement is enacted. The server takes these movements and uses them to update the player state on the server. Every second, the server sends a packet to the connected client with its position as the server perceives it so that the client can correct its position if it gets out of sync with the server. This also allows the server to keep tabs on how the client is moving and whether the client is moving illegally. If the client moves more than 100 meters (1 block = 1 meter) away from

Stellenbosch University https://scholar.sun.ac.za

the server's last known position of the client in a single tick, the client will be kicked for "moving too quickly" and is there to prevent the tampering of positional information by malicious players.

4.4 Non-playable characters

NPCs, also known as entities in Minecraft, are server-controlled avatars of varying types. Players are considered human-controlled entities to the server. These entities are randomly spawned by the server into chunks according to the biome of the chunk. They are dependent on the chunk being loaded into memory in order to have their states updated. Each entity is given a unique identifier on the server upon being spawned, but when the entity information is sent through to the player, the player gives it its own internal entity ID that is specific to its local copy of the entity state.

4.5 Networking and the Minecraft protocol

Minecraft uses transmission control protocol (TCP) with standard WebSockets for connections between the server and the clients. The bytes of information are collected into defined containers called packets. The Minecraft protocol is the specific rule set that defines what each packet means to the recipient as well as how the client and server communicate with each other. Each packet is identified by a packet ID which tells the recipient what kind of information is contained in the packet. The packets inform the client and server about any changes to states. This protocol is updated occasionally to add more functionality to the game as well as to improve legacy code. Protocol versions must be the same across client and server for the game to run as intended.

The client and server exchange a specific sequence of packets when logging in [63]. This sequence is shown in figure 4.5.1 and the description follows. These packets follow a subprotocol and define what state the connection between the two is in. There are four different states:

- Handshaking
- Login
- Play
- Status

Stellenbosch University https://scholar.sun.ac.za

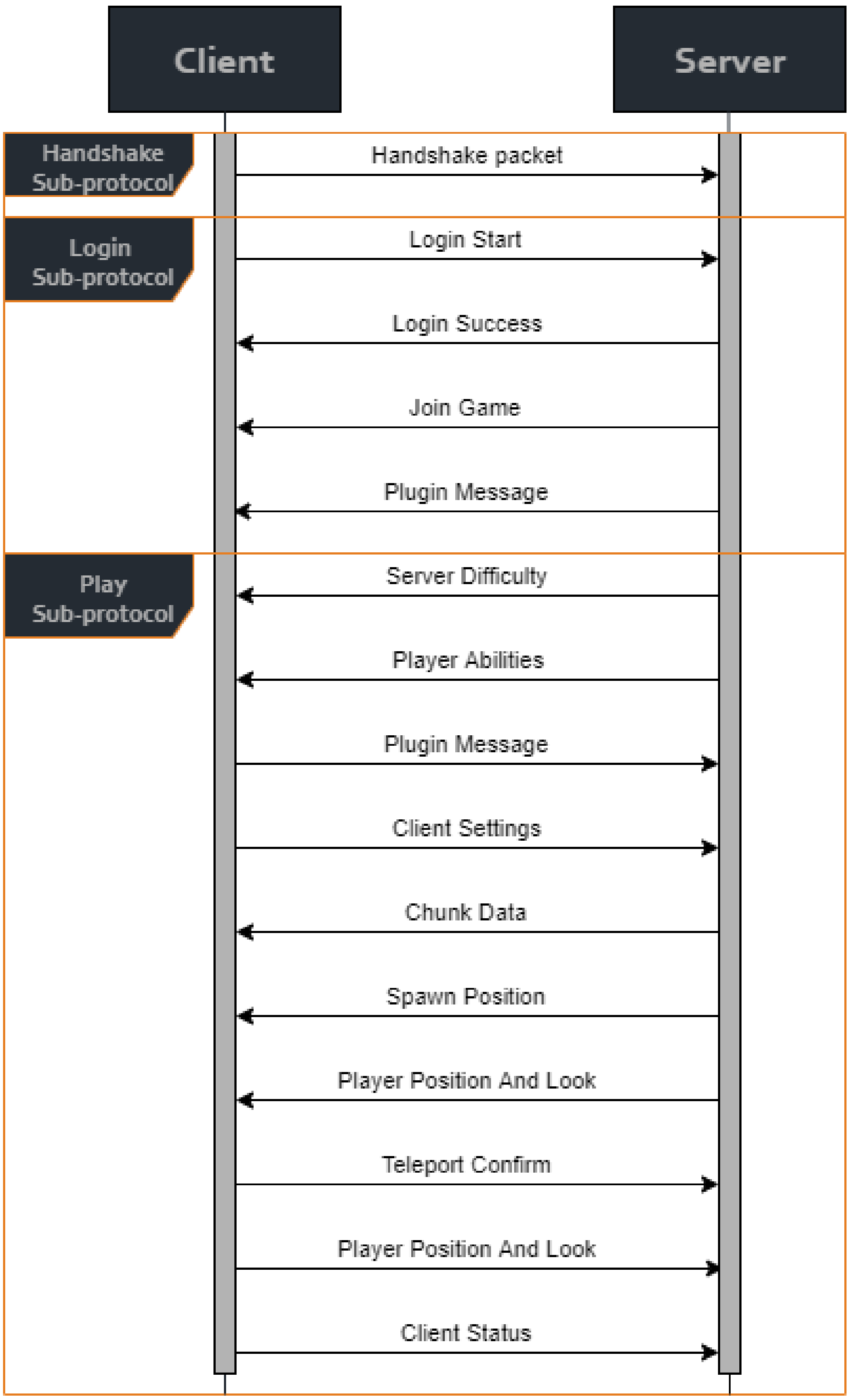


Figure 4.5.1: Login sequence diagram between a Minecraft client and a Minecraft server integrated with VAST

Stellenbosch University https://scholar.sun.ac.za

4.5.1 Handshaking sub protocol

The *handshaking* state facilitates the initial connection between the client and the server. The client sends an initial **Handshake** packet to the server, containing information about the protocol version of the client, the server internet protocol (IP) address and port and the next state that the client will be in (either status or login). It also sends a **Login Start** packet to start the login process, after which it immediately switches its sub-protocol state to *login*. The server receives these two packets and sets the connection sub-protocol to the *login* state on the server-side.

4.5.2 Login state sub protocol

Once in the *login* sub-protocol state, the server sends an **Encryption Request** to the client, to which the client authenticates itself with minecraft.net (to see if the account has purchased Minecraft) and sends a response of the same type back to the server. Once the server authenticates the response, both of them enable encryption of the packets. Custom servers such as Spigot give the option to turn authentication with minecraft.net off in a so-called "offline" mode. When this option is turned on, and authentication is off, this encryption process does not happen. The next sequence, in which the server sends a **Set Compression** packet, is also optional as the packets can be sent without compression.

Following this is a mandatory sequence where the server sends a **Login Success**, **Join Game**, **Plugin Message**, **Server Difficulty**, and **Player Abilities** packet. The **Login Success** packet contains the client's universally unique ID (UUID) which is used by the server to uniquely differentiate between clients. The **Join Game** packet contains the game mode which could be survival (normal), creative (infinite blocks in inventory, invulnerable, flight activated), adventure (good for story modes, can only destroy blocks with tools) and hardcore (similar to survival, with difficulty set to "hard" and no respawns for any player who dies). It also contains the player's entity ID (EID), the dimension the player is joining (there are two others besides The Overworld, the Nether and the End) and some other, less important info. The last packet, **Player Abilities**, tells the client if they are invulnerable, flying if they are allowed to fly, whether they can instantly break blocks (in creative mode), the client's flight speed and a field of view modifier for when the client is under effects such as potions.

The client responds with a **Plugin Message** packet - which contains the plugin channel that is used to send the data as well as whatever data a plugin may need to send - and the **Client Settings** packet. This gives the server some settings that the client is using such as whether the client should be left- or right-handed and the client's render distance. This render distance is important as it controls what the range entities need to be at before the client starts receiving updates about them.

After receiving the **Client Settings** packet and the render distance is received, the client transitions to a "Loading terrain..." screen which signals that the server has started sending **Chunk Data** packets, which is the information about how the world around the client looks. If a 64-bit client has the default render distance of 12, then this means that 12 chunks would be loaded in each direction around the client's spawning chunk. This means that a block of 25 x 25 chunks would be sent to the client, resulting in 625 **Chunk**

Stellenbosch University https://scholar.sun.ac.za

Data packets. This takes up the majority of the client's loading time. When a chunk is generated, there is a chance that an entity is spawned within that chunk. When this happens, a **Spawn Mob**, **Entity Metadata** and **Entity Properties** packet is sent to make the client aware of the relevant entity. When the **Chunk Data** packets have been sent, the server sends a **Player Position and Rotation** and **Spawn Position** packet is sent to set the client's initial position in the world. The server then sets the client connection state to the *play* sub-protocol.

4.5.3 Play sub protocol

Once the client receives its initial position, it changes its sub-protocol to *play* and confirms the position with a **Teleport Confirm** and a **Player Position and Rotation** packet of its own. This completely the login sequence, after which the client receives normal gameplay packets as needed. This is the subprotocol that the majority of packets are sent in and mainly convey state change information. Event and update dissemination happens through this subprotocol.

4.5.4 Status sub protocol

The *status* subprotocol is used to send periodic control packets such as **Keep Alive** packets to the client and **Server Time Update** packets. These **Keep Alive** packets ensure that the clients are still connected to the server. The client sends a packet with a unique number generated using a system-dependent time value in milliseconds (known as a timestamp). The server must respond with a **Keep Alive** packet that has the same number in it or the client will automatically disconnect from the server.

4.6 Event and update dissemination

Update dissemination is handled by the Minecraft server. The clients generate events through actions they commit within the VE and send these actions to the server. The server then takes these events and applies it to the master state of the entities and world that is affected. Once this is complete, these updates are sent to any player that has the chunk where the changes occurred loaded into memory. Each player's connection is sent an update packet for every entity and world state change that occurred. The clients use these update packets to adjust their internal slave states.

This model is simple and effective for ordinary C/S interactions with a moderate number of users but fails when there are a large number of clients connected to the server and near each other. It also fails when there is a sudden event that produces a surge of updates, such as a collection of TNT all going off at once.

4.7 MMVE requirements in Minecraft

In section 1.1.6, four requirements for MMVEs were listed. These requirements include:

- interactivity and fairness between players and in-game objects

Stellenbosch University https://scholar.sun.ac.za

- consistency in the view that a player has of the world around them
- persistence of the world after an indefinite period of time
- scalability of the architecture as more players join the system

This section will now look at these requirements with respect to Minecraft and how well it fulfils each one and the areas that it could be improved.

4.7.1 Interactivity and fairness

Minecraft uses a bucket synchronisation technique (described in section 2.3.2) to aggregate updates across a 50ms window. This allows small fluctuations in latency to be handled and provides fairness to different computational capabilities of connected clients but does not address larger latencies. When latency larger than 50ms is experienced by the client, there can be instances where the interactivity of the client is affected.

This manifests in instances where a block that was destroyed by the client reappears as if it was not destroyed, being attacked by entities that should be too far away to attack and in the worst scenarios, rubber-banding of the client as it is moving. Rubber-banding is when the client moves normally but the server either does not receive the updates or the acknowledgement does not reach the client. This results in the client teleporting back to the last known legal position.

Rubber-banding can also be the case when packet loss is experienced. Because the server holds the authoritative copy of the client's state, if the movement packets are lost and do not update the state on the server then it can result in resetting the client position when it eventually receives its position update from the server. This is very jarring to the player's experience.

4.7.2 Consistency

The server ensures that the world and entity states stay consistent as it is the sole authority of master states. This allows the server to accept or deny updates and also allows it to ensure that conflicting actions cannot occur on a single master state.

The problem with this centralised authority is that when too many clients are connected or if there is a high latency connection then the client's consistency can be affected. The clients do not stay inconsistent for long however as the server quickly rectifies any inconsistencies that the clients may have with the next server tick. The server's ability to deny an update and send the corrected state gives the system a strong consistency. As stated above in section 4.7.1, this consistency appears as blocks disappearing or reappearing, rubberbanding movement or sudden entity positional jumps and attacks being experienced by the player without seeing what caused it.

Stellenbosch University https://scholar.sun.ac.za

4.7.3 Persistence

Persistence in Minecraft is exceptional. Since the Minecraft server is in control of everything from states to logins, there is little chance of losing any data or world persistency if the server behaves as intended. Inherent in this strength as a centralised authority is the weakness of relying completely on a single point of failure. If the server irrecoverably crashes or corrupts in any way, there is no backup scheme or way to recover the information about anything within the world. It is simply lost.

Minecraft worlds that are transferred across updates do not completely update to the new content. Chunks that have already been loaded before the update will appear the same in the new update, but new chunks will load with any new features present in the new update. This is an elegant solution across updates but means that the world is not versatile unless the chunk data is partially deleted and reloaded.

4.7.4 Scalability

As already alluded to, scalability in Minecraft is not very good due to the centralised nature of the architecture. When there are a lot of clients, they take up server resources and bandwidth just to be present in the system.

According to [64], for 1-3 players in a Minecraft server in a wide area network (WAN) configuration, 750 KB/s upload and 375 KB/s download are required as a minimum to host them. Going up to 8 or more players, 3.75 megabytes per second (MB/s) upload and 1.875 MB/s download bandwidth is required. Local area networks (LANs) have much higher bandwidth capabilities than WANs, but limit players to being within close proximity to each other by definition and support limited numbers of users. Therefore WANs are what will be discussed as they have global connectivity.

Something of note in the WAN bandwidth requirements of a Minecraft server is that each client does not require a linear amount of bandwidth. The bandwidth requirements scale up the more clients connect. In a server with 3 players, each player effectively uses 250 KB/s of upload bandwidth whereas when there are 8 players they then use 470 KB/s. The reason for this is because of the *player-player* interactions discussed in 2.1.2. For every player added, there is another player to interact with and therefore more bandwidth required by the client, over and above the bandwidth required for *player-object* and *player update* interactions. According to [18], communication demands caused by the linear increase of clients can increase quadratically. This is clearly not scalable.

4.8 Interest Management

Interest in the realm of MMVEs is defined as the objects, entities or events that the client receives state updates about. Interest management is therefore how the interest for a client is determined. As stated in section 2.3.1, MMVEs face challenges with regards to scaling as the more clients there are, the more local states need to be updated and therefore more network traffic is produced. The aim of IM is therefore to decrease the

Stellenbosch University https://scholar.sun.ac.za

number of states the client is interested in in order to minimise the amount of generated state updates.

In Minecraft, there is natural IM already in place due to the large, procedurally-generated world having a high amount of entities and states. This IM is enacted by the server and is controlled by the client's rendering distance. In the client settings, the client view distance can be set. This is sent to the server via the **Client Settings** packet. The server uses this as a boundary line, sending updates to the client about a state change or event if it falls within this boundary and ignoring everything else. This IM, however, is inadequate when one is scaling to the level of modern-day MMOGs and MMVEs. This is because the scalability bottleneck of a Minecraft server is the bandwidth of the machine hosting the server. Therefore, to scale Minecraft, the bandwidth of the system needs to be reduced. The highest recorded count of concurrent players on a single Minecraft server is 2,622 which occurred on the 1st of August 2011. When looking at this number, it is approximately 0.001% of the total player count of Minecraft, which sits at 176 million [65]. While not all of those players are ever going to play together, the demand to be able to play with more people at a time is always great.

4.9 Research in Minecraft

There has already been research into improving the scalability of Minecraft. One such approach, Manycraft, which incorporates Kiwano, is presented below.

4.9.1 Manycraft and Kiwano

Manycraft is a system that allows the Minecraft server to host many more clients than the standalone server could with the help of Kiwano. This system is seen in figure 4.9.1, taken from [66]. Kiwano is explained first, followed by Manycraft.

Stellenbosch University https://scholar.sun.ac.za

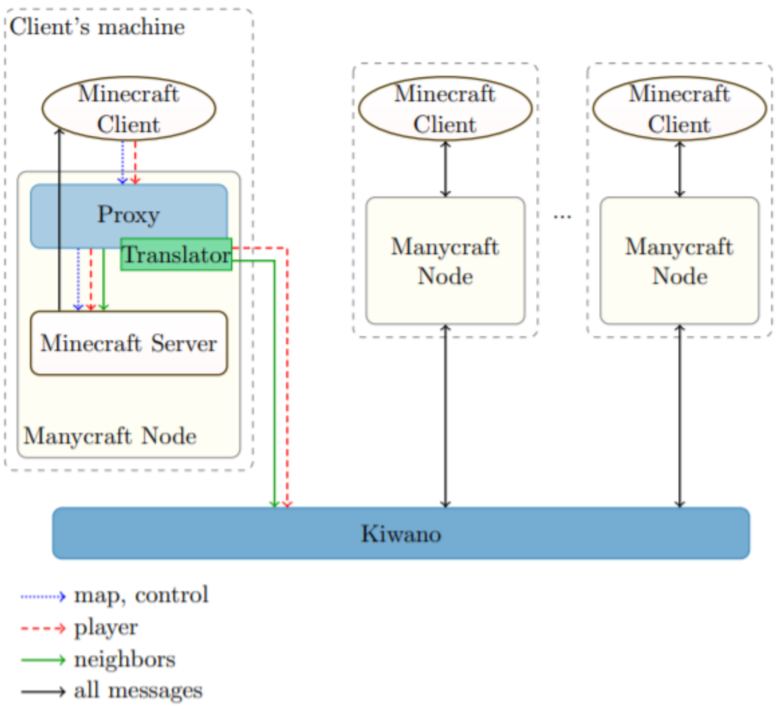


Figure 4.9.1: A Manycraft node with Kiwano integrated to handle event and update dissemination

Kiwano

Kiwano [20] is a scalable distributed infrastructure for VEs, designed to be scalable whilst allowing random, high-frequency movement. It uses as many nodes as needed as it assigns groups of clients to each one based on their close spatial proximity. The clients are indexed according to a Delaunay triangulation and are split into zones that are indexed the same way according to entity density. These zones shift dynamically as the entities move. Each client is assigned a proxy to communicate with Kiwano for it to translate the Minecraft packets to the communication protocol used by Kiwano.

Manycraft

Manycraft [66] uses Kiwano as its method of event and update dissemination between clients. The Manycraft node is installed by a client on their machine with the appropriate Minecraft map and consists of a Minecraft server and a proxy. The proxy bridges communication between the client and the server, but only one way as the server communicated directly with the Minecraft client. The proxy also handles communication with the Kiwano server nodes. Kiwano responds in kind with movements from other clients that are separate to the Manycraft node.

4.10 Summary

In this chapter, a background of Minecraft and its inner workings were discussed in detail. The server and its modification were discussed, followed by a description of the world state and its representation in Minecraft. This was followed by the player entity and a

Stellenbosch University https://scholar.sun.ac.za

description of NPCs. Next, the Minecraft protocol and the networking behind Minecraft, in general, was discussed, including the login sequence that the client performs to join a Minecraft server. Following this, it looked at event and update dissemination as well as how well the Minecraft server and client fulfilled the requirements of MMVEs and discussed the limitations present within the game. Lastly, interest management and research done on Minecraft were discussed, ending in a description of Koekepan.

This project uses Koekepan as the integration platform between VAST and Minecraft in order to test VAST's performance in a real game environment. VAST takes over the networking as well as the server node P2P overlay management. The design and integration will be looked at and discussed in detail in the next chapter. 1

Stellenbosch University https://scholar.sun.ac.za

Chapter 5

Minecraft and VAST

To test VAST's performance in a real-world environment, incorporating it into an established MMVE will give an accurate evaluation of VAST's potential. Minecraft was chosen as the MMVE because of its versatility in the modification of how it operates. Koekepan gives the researcher the ability to intercept packets that are being sent between the client and the server and modify how these packets are sent. It also allows the distribution of the server across multiple hosts which is exactly what Minecraft needs to improve its scalability.

This chapter will discuss the integration of Minecraft and VAST. It starts by discussing how VAST and Minecraft fit together using Koekepan as an interpreter. The layout is discussed and then the different modules of the system are discussed in detail. This is finalised with a discussion on the challenges faced in turning Minecraft into a system that can use SPS for its event and update dissemination.

5.1 VAST and Minecraft

The system consists of a custom Minecraft server, a server and client proxy that wraps the Minecraft packets into packets consistent with VAST's API, and emulated Minecraft clients that function as lightweight vanilla Minecraft clients. These clients are used due to their lack of a graphical user interface (GUI) which significantly reduces the load on the host, which is ideal for running scaling tests. They work like normal Minecraft clients and therefore will not be discussed, but can be read about in [67] at the reader's discretion. This system can be seen in figure 5.1.1.

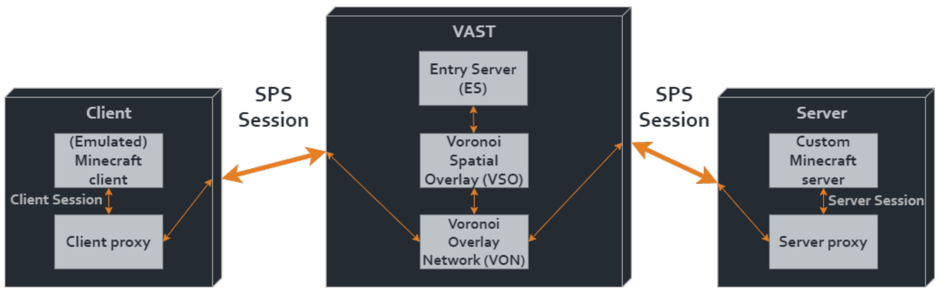


Figure 5.1.1: An overview of the VAST system connected to a Minecraft client and server module

Stellenbosch University https://scholar.sun.ac.za

The client and server proxy design and implementation will be discussed first and then the login procedure will be looked at to see how the system runs.

5.1.1 Client proxy design

The client proxy sits between the Minecraft client and VAST. It accepts communication from the Minecraft client, wraps it and sends it through to VAST for publishing to the SPS.

The proxy is a standalone component in the ecosystem. Upon start-up, it is given the IP address and host of the SPS layer it should connect to as well as the IP address and port with which it should start a listening server at. This server listens for incoming connections from Minecraft clients. It also establishes a socket connection to the SPS layer along with an initialisation of the Minecraft protocol and sub-protocol. These protocols, interpreted and reconstructed by "SteveIce10's" MCProtocolLib [68], describe each packet type's class variables, the accessor functions and the functions that write the fields to a byte buffer as well as read them from a byte buffer. This is useful when reconstructing packets from packet IDs or when the packet information needs to be communicated over a WebSocket connection.

When the listening server receives one of these client connections, the proxy establishes a line of communication between the client and the SPS. It does this by creating a channel of communication, called a session, between the client and itself and the SPS and itself. These sessions should be linked to create a single line of communication between the client and the SPS.

For the information running through this channel to be of the right format on either side, it needs to be interpreted and manipulated along the way for either side to understand. It does this by using a behaviour handler to register how the session should behave when a specific packet is received.

Behaviours

A behaviour is a description of how the system should handle a specific packet. The system starts up a packet behaviour handler with both the client session as well as the SPS session and registers pre-defined behaviours for all of the packets the session expects to receive. These registered behaviours both route the information to where it needs to go as well as allow the system to do packet-specific tasks. It is this that allows the system to listen to each packet and decide which packets will only need basic wrapping and which ones will require some extra modification or actions before sending it through to the recipient.

SPS session

The SPS session requires more parts than the client session because it has to interface with the VAST API, whereas this is done inherently within the Minecraft client when receiving Minecraft packets. The SPS connection sets up listeners that are used to interface

Stellenbosch University https://scholar.sun.ac.za

with VAST. This includes the **JOIN**, **SUBSCRIBE**, **UNSUBSCRIBE**, **PUBLISH**, **MOVE**, and **LEAVE** procedures defined in section 3.4.

During the initial connecting procedure, the SPS session's channel is initialised. This defines what channel the session will publish on. This channel can be changed when the client needs to publish on a different channel. Channels are described in section 3.4.2. It also sends its type (server or client) to the VAST system, informing it of whether it should treat its connection and subsequent subscriptions as a client or server subscription. This will affect whether it gets packets from the server or the client. The client proxy will state that it has a client connection and will, therefore, receive packets published by the server. The client proxy should also subscribe to the *lobby* channel when connecting a client to the system. This is so that the initial login packets reach the lobby server and the login procedure can be followed. The lobby server is defined as the first server that joins the system. It subscribes to the *lobby* channel to accept incoming messages from clients that are logging in. Once the client receives confirmation that it has joined the VE, it can unsubscribe from the *lobby* channel, assuming no other clients are still in the process of logging in.

Each SPS session that connects to VAST receives a unique ID. This is the client ID that VAST uses to identify different clients and is mapped to the SPS session for use whenever a packet is sent to VAST. When receiving packets from VAST, the client's username is used to look up which session the received packet should be sent through.

5.1.2 Server Proxy

The server proxy works in a very similar way to the client proxy, with an SPS session and a server session that communicates to the Minecraft server. Upon instantiation, the connection to the SPS is established. It immediately connects to VAST and subscribes to the *lobby* channel so that it can await incoming Minecraft client connections through the SPS. When receiving a connection, it creates the SPS session and server session for that specific client's connection. It registers the server-bound packet behaviours with the packet handler and registers the session with the username of the connecting client. Deviating from the way that the client proxy similarly sets up its sessions, the server proxy also sets up an entity tracker.

Entity tracking

To allow Minecraft to integrate with an SPS architecture, the ability to spatially track entities and publish their updates from a specific location is necessary so that entities within Minecraft are seen and interacted with only when they fall within a player's AoI. The entity tracker is responsible for this. When the server proxy creates the sessions it uses to facilitate communication between the SPS and the server for a specific client connection, it also starts up a client-specific entity tracker. This entity tracker stores the positional states of entities perceived by the client and updates them upon receiving state updates. It does this by using a hash map data structure to store entity IDs with their states as a key-value pair. This allows for efficient lookup of entities and updating of their states.

Stellenbosch University https://scholar.sun.ac.za

These entries are initiated by the receiving of the **ServerSpawnMob** packet which contains the entity ID and UUID, their Cartesian coordinate position and velocity vector, as well as the metadata associated with the entity (such as entity type and unique effects). An entity object stores all of the entity information and this is linked with the entity ID in the hash map.

When a **ServerEntityPosition**, **ServerEntityPostitionRotation** or any other entity-modifying packet is received, the entity is updated with the content of the packet. These packets then get published using the position of the entity as the point of publication to give the packets a spatial aspect in the dissemination procedure. This is how entity-related packets are modified to be SPS-compatible. The entity is removed from tracking when a **ServerEntityDestroy** packet is received.

Player tracking

Similar to entity tracking, player tracking is done by the entity tracker. The player entity is stored when a **ServerSpawnPlayer** packet is received and removed when a **ServerEntityDestroy** packet is received. Once the player is spawned, it is treated like any other entity in the world and therefore is updated with packets such as the **ServerEntityPositionRotation** packet. These packets are published using the same positional technique as the entities described above and allow for the state updates of the client to be published with spatial coordinates, therefore completing the conversion of the update dissemination to an SPS functionality.

5.1.3 Minecraft client login procedure

Section 4.5 describes the login sequence of a Minecraft client. Figure 5.1.2 shows the sequence diagram of the login procedure. The way that the client proxy handles this sequence is as follows:

1. The **Handshake** packet is received by the client proxy and forwarded directly through to the SPS, which publishes the packet on the *lobby* channel. The position of the publication is $(0, 0)$ since the position of the client is not yet known.

2. The client follows this up with a **ClientLoginStart** packet. Its behaviour tells the SPS session to initiate the connection procedure, which prompts the creation and forwarding of the **EstablishConnection** packet which is a wrapper of the **ClietLoginStart** packet. This tells the server proxy that a new session is connecting and allows it to create the SPS session on the server proxy's end, the session that communicates with the server as well as the entity tracker.

3. The login packets now flow back and forth on the lobby channel through the established pipeline until the client receives the **ServerJoinGame** packet. When this is received, it subscribes to the *ingame* channel in order to start receiving packets related to the **Play** sub-protocol.

4. When the client receives the **ServerPositionRotation** packet, it starts publishing its packets to the position of the client.

Stellenbosch University https://scholar.sun.ac.za

5. Once the server has sent all of its **Login** sub-protocol packets, it changes its channel of publication to *ingame* and continues sending packets through to the client. This subscription to the *ingame* channel is player-specific. On the client-side, when it receives the **ServerChat** packet, it knows that the server has switched to the *ingame* channel and therefore, if there are no other simultaneously connecting clients, can unsubscribe from the *lobby* channel.

6. Once this is complete, the login procedure is done and the client can interact with the game as normal.

7. Upon receiving **ClientPlayerPosition** or **ClientPlayerPositionRotation** packets, the SPS session associated with the client updates its position of publication so that all publications come from the client's position.

Stellenbosch University https://scholar.sun.ac.za

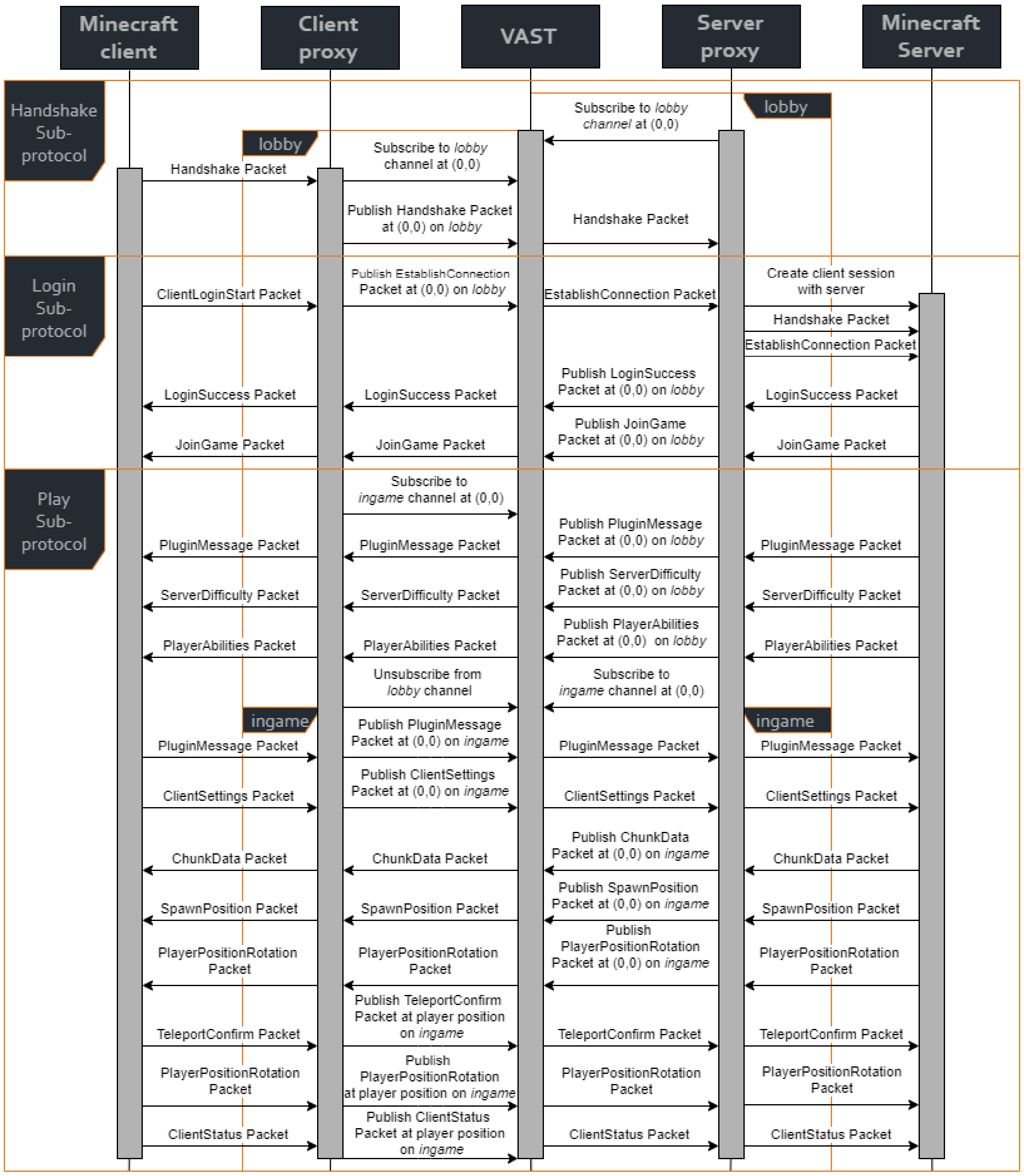

Minecraft client
Client proxy
VAST
Server proxy
Minecraft Server
Handshake Sub-protocol
lobby
Subscribe to *lobby channel* at (0,0)
lobby
Handshake Packet
Subscribe to *lobby* channel at (0,0)
Publish Handshake Packet at (0,0) on *lobby*
Handshake Packet
Login Sub-protocol
ClientLoginStart Packet
Publish EstablishConnection Packet at (0,0) on *lobby*
EstablishConnection Packet
Create client session with server
Handshake Packet
EstablishConnection Packet
LoginSuccess Packet
LoginSuccess Packet
Publish LoginSuccess Packet at (0,0) on *lobby*
LoginSuccess Packet
JoinGame Packet
JoinGame Packet
Publish JoinGame Packet at (0,0) on *lobby*
JoinGame Packet
Play Sub-protocol
Subscribe to *ingame* channel at (0,0)
PluginMessage Packet
PluginMessage Packet
Publish PluginMessage Packet at (0,0) on *lobby*
PluginMessage Packet
ServerDifficulty Packet
ServerDifficulty Packet
Publish ServerDifficulty Packet at (0,0) on *lobby*
ServerDifficulty Packet
PlayerAbilities Packet
PlayerAbilities Packet
Publish PlayerAbilities Packet at (0,0) on *lobby*
PlayerAbilities Packet
Unsubscribe from *lobby* channel
Subscribe to *ingame* channel at (0,0)
ingame
ingame
PluginMessage Packet
Publish PluginMessage Packet at (0,0) on *ingame*
PluginMessage Packet
PluginMessage Packet
ClientSettings Packet
Publish ClientSettings Packet at (0,0) on *ingame*
ClientSettings Packet
ClientSettings Packet
ChunkData Packet
ChunkData Packet
Publish ChunkData Packet at (0,0) on *ingame*
ChunkData Packet
SpawnPosition Packet
SpawnPosition Packet
Publish SpawnPosition Packet at (0,0) on *ingame*
SpawnPosition Packet
PlayerPositionRotation Packet
PlayerPositionRotation Packet
Publish PlayerPositionRotation Packet at (0,0) on *ingame*
PlayerPositionRotation Packet
TeleportConfirm Packet
Publish TeleportConfirm Packet at player position on *ingame*
TeleportConfirm Packet
TeleportConfirm Packet
PlayerPositionRotation Packet
Publish PlayerPositionRotation at player position on *ingame*
PlayerPositionRotation Packet
PlayerPositionRotation Packet
ClientStatus Packet
Publish ClientStatus Packet at player position on *ingame*
ClientStatus Packet
ClientStatus Packet

Figure 5.1.2: Figure showing the login sequence of a Minecraft client through VAST

Stellenbosch University https://scholar.sun.ac.za

5.2 Implementation challenges

This section describes some of the challenges faced in implementing the design and how these challenges were addressed and overcome if they were.

5.2.1 Connection sequence

VAST expects the connecting client to connect using the **JOIN** procedure, giving its type, username and position in order to determine which matcher it is routed to and to allows the matcher to identify it accordingly. However, in Minecraft, the **Handshake** packet does not contain any of this information yet is the first packet that gets published and therefore needs a way to reach the server.

The challenge posed by this is how does one create a generic connection to VAST and not have it be a part of the VSO layer or be represented as a client in the system? The server needs to be connected initially to receive the packet. How this was solved was with the idea of a lobby server and a new method **TYPE** in the VAST API. This lobby server connects to VAST upon creation and uses the **TYPE** method to tell VAST that it is a server. It then subscribes to the '*lobby*' channel in order to await incoming connections. This lobby server maintains this subscription to VAST and is always looking to receive new connections from clients. Its subscription has a position of $(0, 0)$ and an extremely large radius (currently 10000 but could be made larger if there are clients that extend beyond that radius).

Any other connection that is instantiated first calls the **TYPE** method to tell VAST what kind of connection it is. They subscribe to the '*lobby*' channel to communicate with the lobby server and publish from their last known position, defaulting to the same position as the lobby server. When they receive their first positional update, only then do they move away from this initial position and publish from their actual location. This ensures that the connecting client will always be able to access the lobby server.

5.2.2 Duplicate packets

When two clients have an entity or any other world object in their view, and an event occurs that affects the state of said entity or object, the Minecraft server generates an update message for both clients and sends it to them individually. Since this update message is published at the point of origin of the entity/object, both clients, whose subscription regions overlap this point, receive both of the publications. The Minecraft client disconnects from the server when it receives packets that are from the incorrect sub-protocol compared to what it is listening out for, or if it receives a packet that it does not understand and thinks something is wrong.

To solve this channel, the publication contains the username of the client for whom it is destined. This allows the proxy on the other end to use this username to find the specific session that handles the communication for that client connection and sends the packet that way. The problem with this implementation is that it does nothing to reduce the number of packets being sent overall and thus reduces VAST's benefits. This is an issue with how Minecraft disseminates its updates and its compatibility with SPS and is discussed below.

Stellenbosch University https://scholar.sun.ac.za

5.2.3 Update dissemination by the Minecraft server

Minecraft, like any other game, is designed to operate singly and does not take easily to being tampered with or adjusted. The way it disseminates its updates is by generating an update for each player connection and sending it out that way. It is for this reason, among others (discussed in section 4.7.4), that Minecraft is currently not scalable in its base form. In order to reduce how these updates are generated for the players, the server back-end would need to be modified so that only a single update message is generated for an event, and all of the players could get their information from this update message that would be disseminated by the SPS. This is no simple task however.

The way that reverse-engineered Minecraft servers such as Spigot work is that there is not a central managing class that handles update dissemination to all clients. Every event that is created separately determines whom it should affect and notifies the individual player connections that they should update the state for that specific object. This decentralised approach to update dissemination means that every entity event type, world event type and other events would need to be individually adjusted to change it from sending to individual player connections to a single update with a list of players that should receive it. With there being over 100 different instances of event processing points, coupled with the fact that the code is obfuscated to protect it from being copied by others, it was decided that a theoretical analysis of Minecraft's packets and which ones could be compatible with SPS would be a more efficient use of time, leaving the conversion of the Minecraft server for future work. This theoretical analysis is shown below.

5.3 Minecraft SPS potential

The purpose of SPS is to offload the event and update dissemination from the server to the SPS layer which is built to be scalable. This allows the server to focus on event processing and the SPS layer to alleviate resource usage from the server. Using VAST, this technique can theoretically increase Minecraft's scalability possibilities.

5.3.1 Theoretical analysis

As described in section 4.8, Minecraft currently implements its own IM in order to determine which clients need to receive updates after an event has occurred. When this happens, it generates a separate packet for each client to disseminate the update. This can be very inefficient as information is increasingly duplicated the more clients and entities that are in the system and within each others' AoIs.

In theory, these duplicated packets can instead be combined into a single packet with multiple recipients and the SPS can publish this single packet to all of the relevant subscriptions. This is based on the idea of uni-casting versus multi-casting (discussed in section 2.3.3) which uses multi-cast groups (subscriptions in this case) to reduce the processing and number of packets sent between the server and the clients. Since the SPS is designed to efficiently distribute the workload of sending multiple packets across many nodes, it handles the multi-casting better than the server would.

Stellenbosch University https://scholar.sun.ac.za

Since the packets would need to be published with a spatial coordinate, the packets that would be the most applicable to this type of dissemination are ones that originate within the VE. For example, an entity that moves produces a **ServerEntityPosition** packet. This packet contains an entity ID, relative x, y and z movement from its previous position and a boolean variable saying whether it is grounded or not. This packet gets sent to every client that can see the entity and the only difference is the recipient. The movement packet could instead be published at the final location of the entity movement and each subscriber could receive the update. Compare this to a **ServerUpdateTime** packet that is devoid of any point of origin within the packet structure. This kind of packet would need to be published globally without a position, and is the kind of packet that would not benefit from SPS, but could still benefit from having a single packet being disseminated to multiple clients.

To estimate how much Minecraft would benefit from implementing SPS, one would need to look at the packets that are sent and see what percentage of them have spatial coordinates that could be exploited as well as which of them are duplicated by the server and could be combined into a single publishable packet.

5.3.2 Hypothesis and setup

In this test, the hypothesis is that most of the packets that are sent by a Minecraft server are both related to entities and entity updates and are relevant to a specific location in the VE. Of these packets, since most of the traffic is speculated to be related to objects or entities, they should be combinable into a single publishable packet.

An emulated Minecraft client is placed into a random, normal Overworld. A vanilla Minecraft client is then made to run around in the world around it and perform random tasks such as destroying blocks, attacking entities, placing blocks, and running and jumping around as a normal player would. This creates a variety of event updates that get sent to the emulated client, which logs all of the packets received from the server. It only monitors the packets that are sent by the server as those are the packets that would be replaced by the SPS and reduction functionality.

Stellenbosch University https://scholar.sun.ac.za

5.3.3 Practical analysis and results of SPS potential

In figure 5.3.1 we see the number of each sent packet in the session. Overwhelmingly we see that the number of **ServerPlayerPositionRotation**, **ServerEntityHeadLook**, **ServerEntityPosition**, **ServerEntityPositionRotation**, and **ServerChunkData** packets constitutes the majority of the packets that are sent to the client from the server. All of these packets have spatial locations and therefore would benefit from being disseminated using SPS. In the next 14 highest packet counts, only the **ServerUpdateTime** and **ServerKeepAlive** packets do not have a spatial location.

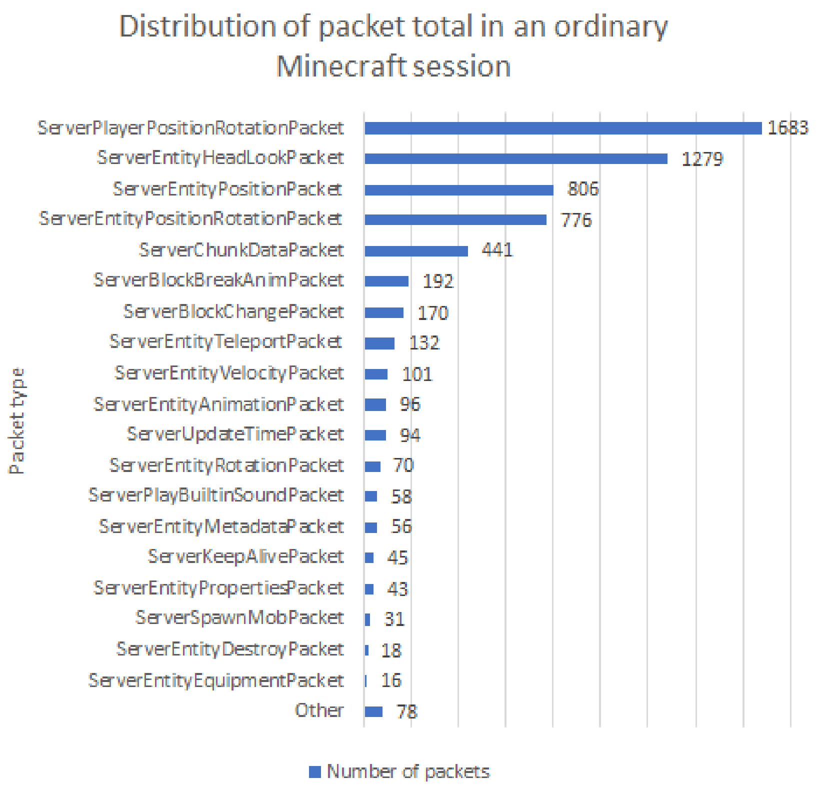


Figure 5.3.1: The distribution of packets in an ordinary Minecraft playing session

In figure 5.3.2 one can see the percentage of the total number of packets sent that each packet comprises of. The top 7 packets comprise of over 85% of the total number of packets sent, and all of these packets can be spatially published using SPS.

Stellenbosch University https://scholar.sun.ac.za

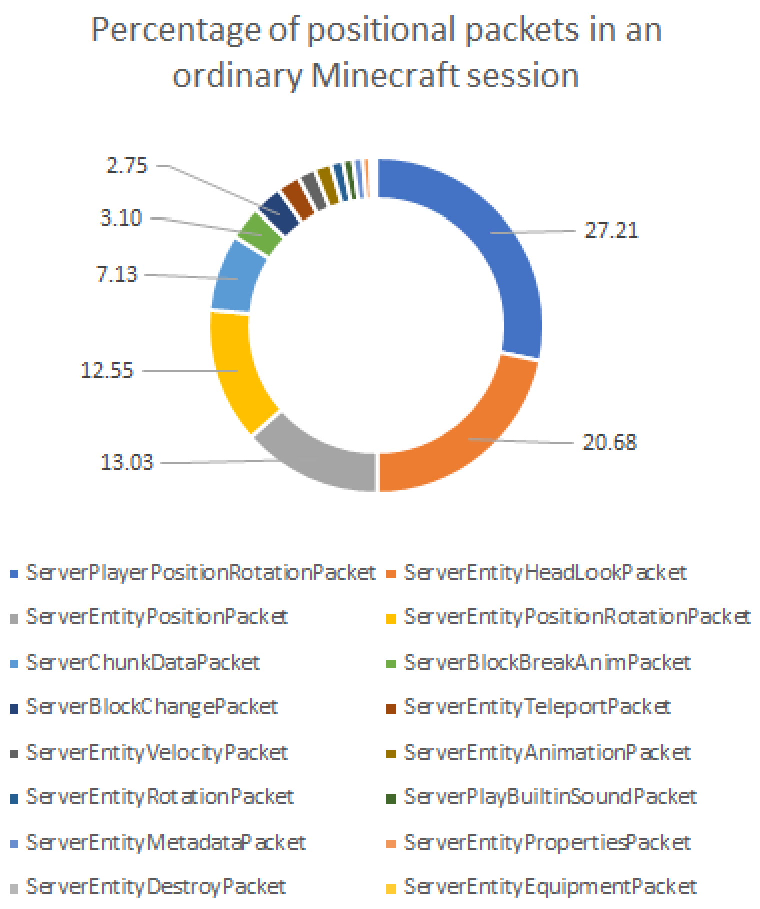


Figure 5.3.2: The percentage of packets in an ordinary Minecraft playing session

5.3.4 Packet reduction potential

Of the packets that were mentioned above, the only packets that could not be reduced and sent as a single packet to multiple clients are the packets that are action-confirmation packets that get returned to the client that created the action such as **ServerPlayerPositionRotation** and **ServerBlockChange** packets. That is because these packets are sent to the player to confirm its action has been registered by the server.

When it comes to the percentage decrease that SPS could provide to Minecraft, one needs to take each packet that is sent and determine how many packets would not be sent if they were to be published as a single packet. An equation for the packet decrease could then be determined.

The packets that can be reduced depends on the number of clients in the system. Let the number of clients interested in an action-initiating entity be n. This does not include the player as they do not receive updates about themselves in the same way that others do. For every reducible packet, a single packet needs to be published to the players interested in the affected entity. This means that the number of packets that can be reduced by SPS is $n - 1$ for each instance of an action (chosen to be x) that requires an update to

Stellenbosch University https://scholar.sun.ac.za

be sent to the players. Every instance of a packet x is duplicated n times and then can be reduced by $n - 1$ times the instance of the packet to get the number of packets that are sent.
This makes it clear that the number of packets that are sent stays constant no matter how many clients there are which indicates scalability.

For a single client in this test containing two clients and the Minecraft server, the number of packets sent to a single client from the server is 6185. In this test, the number of packets that are sent to both of the clients overall amounts to approximately $6185 * 2 = 12370$. Since at least a single instance of a packet needs to be sent at least, the amount of packets that could be reduced is the total number of packets sent to a single client minus the number of packets that are client-specific (contain information that only affects that client) which amounts to 4147. Therefore, the percentage packet reduction is $(4147/12370) * 100 = 33.52\%$.

5.3.5 Conclusion on Minecraft SPS viability

This test has shown that Minecraft network packet distribution would fit an SPS implementation and therefore shows that VAST can indeed be integrated into Minecraft with positive results in theory. Of all of the packets sent by the Minecraft server, only 5% of packets would need to be globally published overall and therefore has great potential for scalability and performance improvements. The packet reduction depends on the number of clients in the system, but the lowest it could be is 33.52% at two clients.

5.4 Summary

This chapter described the integration of Minecraft and VAST, as well as the theoretical viability and benefit of Minecraft with an SPS implementation handling the update and event dissemination. First, an overview of how the modules of the system work together is given. This is followed by a description of the client and server proxy design. The Minecraft login procedure was then presented. The implementation challenges involved in integrating Minecraft and VAST was then discussed with the SPS viability and benefit of Minecraft concluding the chapter. In the next chapter, a performance evaluation of the system is presented.

Stellenbosch University https://scholar.sun.ac.za

Chapter 6

Performance Evaluation of VAST and Minecraft

This chapter sets out to evaluate the performance of VAST as a system, firstly by looking at the VON layer and its scalability performance in lieu of its requirements for consistency, interactivity and accuracy in its view of the neighbours around it. Subsequently, the chapter aims to evaluate the performance of the VSO layer and the matchers' ability to organise themselves in a way that prevents them from being overloaded, while still providing the consistency and interactivity expected in an MMVE such as Minecraft. Finally, the performance of Minecraft using VAST for its event and update dissemination is discussed, along with the challenges of fitting Minecraft into an SPS architecture.

6.1 VON scalability performance

The VON scalability performance test aims to determine how the system performs as more VON peers are added to it, how well the system performs when the peers are moving, and the effect of peer density (PD) on the system. It analyses the scaling performance and potential of the system and determines whether there are scaling limits of the current version of VAST. An important factor to note is that the peers that are moving in this test are different to clients that connect to the VAST system and move. The moving of peers is a test of how well the system performs when the system is being load-balanced and the peers are having to rearrange themselves to handle varying loads. Firstly, a discussion on the metrics used will be undertaken, followed by the test set up and then the results, culminating in a discussion of the scalability of VAST.

6.1.1 Metrics

The following metrics are used to evaluate the performance of the architecture:

Bandwidth

Bandwidth is the amount of data being sent at a fixed amount of time. Whenever a peer communicates with a neighbour a packet is sent and the total size of this data is measured over the duration of the test. Bandwidth is important to monitor because it is one of the bottlenecks when it comes to scaling systems. The more clients in a traditional

Stellenbosch University https://scholar.sun.ac.za

system, the higher the bandwidth and the more processing power the peer requires. It is therefore desirable to have as low bandwidth as possible to minimise computation time and power, but more importantly, the bandwidth of each peer should not increase as more peers connect to the system. The bandwidth should increase to a point and then hit a limit. Each client's bandwidth is measured in KB/s.

Round-Trip Time (RTT)

RTT is the amount of time it takes a packet to travel from one peer to another and back. There is a physical limit to how low the latency can be because of the limitations of the electric signals travelling through a sub-ideal medium of the wires. There is also a latency associated with the packets being transferred through the router to the remote host. The round-trip time is measured by sending a ping packet with a timestamp from the peer to another peer. The recipient peer immediately sends a pong packet with the original timestamp back. The client then compares the time of receiving the pong packet to the sending time and calculates the RTT (which is measured in milliseconds (ms)).

Latency

Another way of determining the bottlenecks of VON is to calculate how long a packet takes to be processed after being sent. This requires all the hosts to have synchronised time as timestamps are compared when sending and receiving on two different hosts. This is done using one of the hosts as an NTP (Network Time Protocol) server for the other hosts' time server. This indicates how much the system is lagging in terms of handling incoming packets. Latency is measured in milliseconds.

Drift distance

Drift distance is the distance of the perceived location of a peer from another peer's point of view relative to its actual position. Due to latency in the packet transfer and the latency of the system, the peer can have a view of a peer that is outdated and so it could have already taken its next step of movement by the time the previous move is processed. The drift distance is measured as the average distance that the perceived and actual position differs over the course of the test in units across all peers. It is given in two ways: the absolute value of the drift distance and the actual drift distance. This is done to accurately represent how far the client has drifted from where it should be as well as the average distance using both the positive and negative directions.

Consistency

Consistency in this context refers to how accurate a peer's view is of its neighbours. Because of drift distance, the incorrect positioning can cause the peer to be connected to or disconnected from peers that it should not be. This would give the peer an inaccurate view of its neighbours, resulting in it being inconsistent with the correct state of the system. Consistency is measured by taking each peer's position and their neighbours at intervals of 500ms over the course of the test. This is then compared to a reconstruction of the test peers' accurate movements in order to get the percentage of how often the view was consistent with the reconstruction.

Stellenbosch University https://scholar.sun.ac.za

6.1.2 Test bench and set up

The following simulation was done using 11 hosts with identical specifications, all connected in a local-area network. The specifications are given in appendix A. Each computer runs three instances of Node.js with each instance hosting 30 peers resulting in 991 VON peers in total. The extra peer comes from a single host which runs another instance of Node.js with a single VON peer which acts as the gateway. This is done to ensure that Node.js does not throttle the system with its inner thread handling and slow down the peer's connection to the gateway. This happens when it takes longer to process the connections than the time between the clients connecting, resulting in a backlog. This is undesirable as if the system is throttled important processing that needs to happen (such as consistency management) cannot, and the performance of the system will drop. The test starts using a shell script that automatically sets the system up. Once all of the instances are started, the test runs for 5 minutes and then shuts down.

The test is started by creating an instance of Node.js for the gateway on the first host and then proceeding to start instances of 30 peers on each host, leaving enough time between them for the 30 peers to connect at 600 ms intervals. These intervals give the gateway enough time to process the connection and for the peers to join the network. Once 331 peers have connected, the process is repeated - barring the gateway client - to give a total of 991 peers in the system. Once the peer has joined the system, it moves one unit space every 500 ms to simulate the adjustments needed to handle load balancing and the connections to peers. This is meant to simulate the system under high levels of stress and evaluate its performance in this state. It is assumed that the system will work better if there are less peers or they move less frequently, thus testing the upper limits of the system is desirable. Two tests are run: one with the radius, representing the AoI, changing to determine the effect of different peer densities on the system and another with a single density but a different number of peers to determine the effect of the number of peers on the system. The data is collected for the duration of the test and is then processed to gather the results.

6.1.3 Test 1: Varying the peer density

The first test evaluates the metrics as a function of a changing PD. This is calculated by giving the peers a specific AoI radius. This radius is calculated by running several separate simple simulations until the radius that gives the desired average peer density is found. The test is then run with this peer radius and the peer density is calculated by taking the average of all of the peers' average neighbours across the test. A peer's average neighbour count is found by calculating its neighbours every 500ms throughout the testing. The peer density as a function of radius is shown in figure 6.1.1. The varied peer density is analysed according to the aforementioned metrics below.

Stellenbosch University https://scholar.sun.ac.za

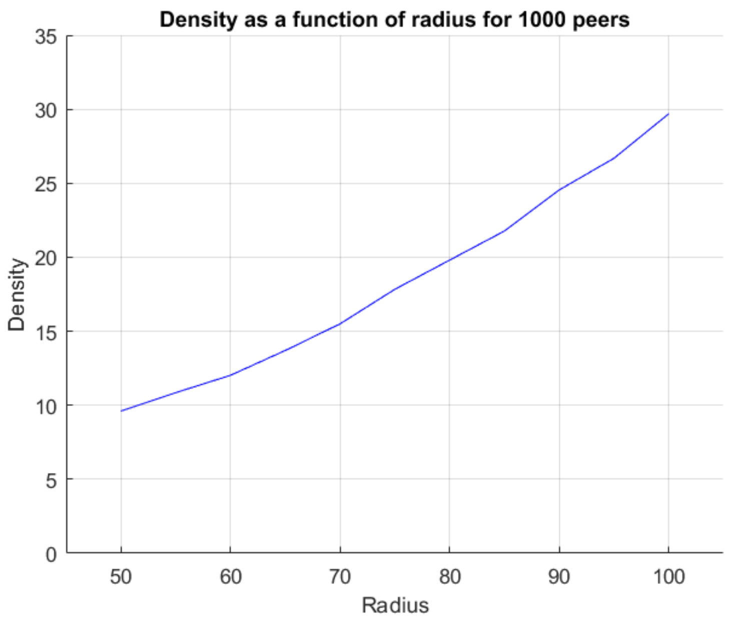


Figure 6.1.1: Figure showing the change in peer density as a function of radius

Bandwidth

The bandwidth can be seen in figure 6.1.2 to have a slightly increasing trend as the PD increases. This trend is to be expected, as the more neighbours that a peer has the more information it will be receiving and the more traffic there will be over the network as a result. It starts at 12.168 KB/s at a PD of 8.71 and ends at 36.1 KB/s at a PD of 23.73 with negligible standard deviations of around 40-50 bytes per second. These numbers are feasible for purely networking functions as is proven to be scalable at reasonable PDs (which is determined to be at a PD of 15 at 19.5 KB/s).

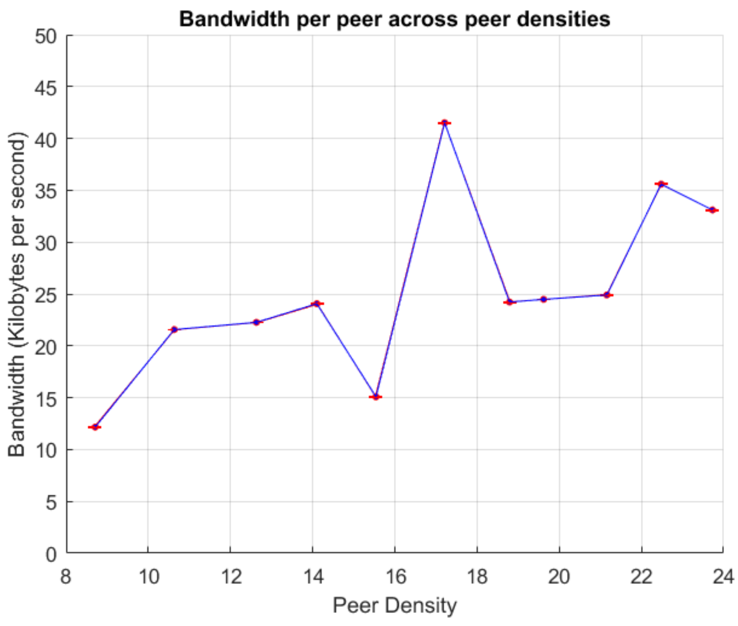


Figure 6.1.2: Bandwidth per peer across varying peer densities

Stellenbosch University https://scholar.sun.ac.za

Round-trip Time

Immediately it is apparent that the RTT is low for all peer densities in figure 6.1.3, not going above 4.2ms. This is a favourable sign as it indicates that the network was not saturated by all of the PDs tested and leaves room for scalability. It is noted that all of the traffic was routed through a router that was local to the hosts and greatly reduced the travel time of the packets between the hosts. The standard deviation is minimal and adds no value to the results and therefore is left out for clarity in the figure.

The results indicate that despite the RTT being low, the increase from a low density of 9 to a high density of 24 results in the RTT doubling. This is significant and shows that the density cannot increase in a scalable manner, but for all reasonable peer densities, the RTT is not the bottleneck of the system.

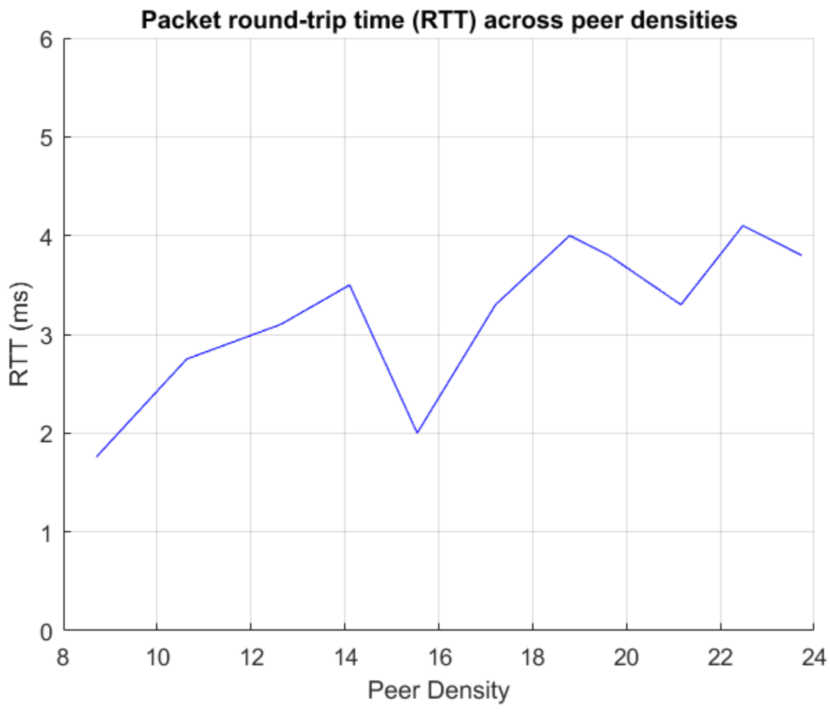


Figure 6.1.3: Round-trip time as a function of peer density

Latency

The effect of an increase in PD on the latency is shown in figure 6.1.4. There is a strong upward trend to the data, starting at 19.61 ms at a PD of 8.71 and increasing to 355 ms at a PD of 23.73. This trend shows that the more peers that are connected to a single peer, the longer it takes to process information.

This is to be expected, as there are more peers doing neighbour discovery and consistency checks as well as all the messages that are sent and received due to moving neighbours that are connecting and disconnecting. In this we see that the system is not scalable with regards to PD as the growth in latency is linear relative to PD growth.

Stellenbosch University https://scholar.sun.ac.za

This increase in latency is partly a result of the procedure where the peer join request is forwarded to an acceptor and the subsequent join procedure. The more peers that join the system, the more hops that the join request needs to do in order to find the correct acceptor and therefore the higher the latency. This could be improved by using a more efficient forwarding system for the finding of an acceptor node, as well as the join procedure. However, for the intents and purposes of this project, it is sufficient as this is a test of the upper limits of the system.

The other contributor to the latency is the actual process of discovering neighbours and keeping consistency in the peer's view. Since figure 6.1.4 is showing the average peer density, some peers have a higher peer density and therefore those peers experience more latency than the peers who are below the average peer density due to having more neighbours to interact and do consistency checks with.

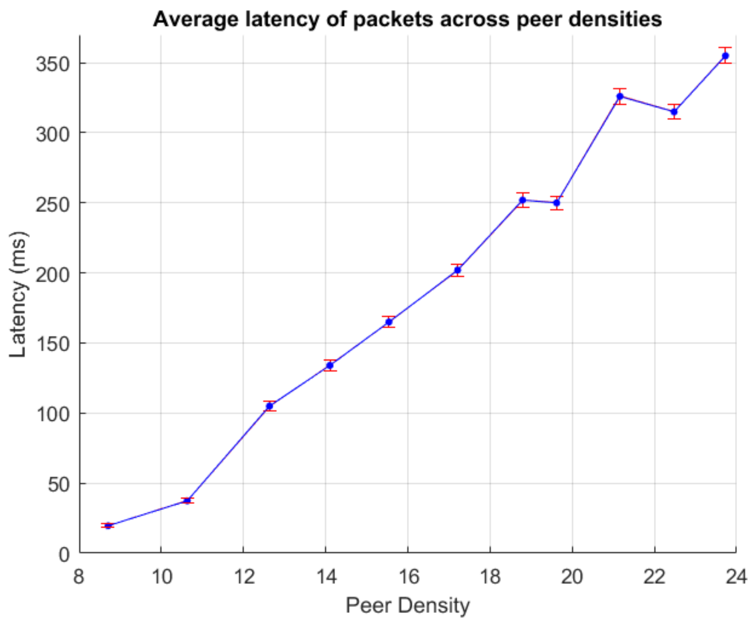


Figure 6.1.4: Latency as a result of varying peer densities

Consistency

The consistency checked here refers to the topology consistency of the peers. Figure 6.1.5 shows a consistent flat line across all densities with the greatest fluctuation being a 2% change. This shows that this metric is not affected by PD. That is because it usually takes a large difference in peer positions for the topology to be altered, and even if there is one client that is inconsistent when compared to even the lowest density of 8.71, that only results in a loss of 11.5% for a single instance of a consistency check, of which there are usually upwards of 10 million across the course of a five-minute test.

When we consider the fact that the topology fixes itself relatively quickly as the position of the peer is recovered one can understand why consistency is not affected too much. This does show in the standard deviation showing a spread of about 5% across densities. This is considered to be scalable.

Stellenbosch University https://scholar.sun.ac.za

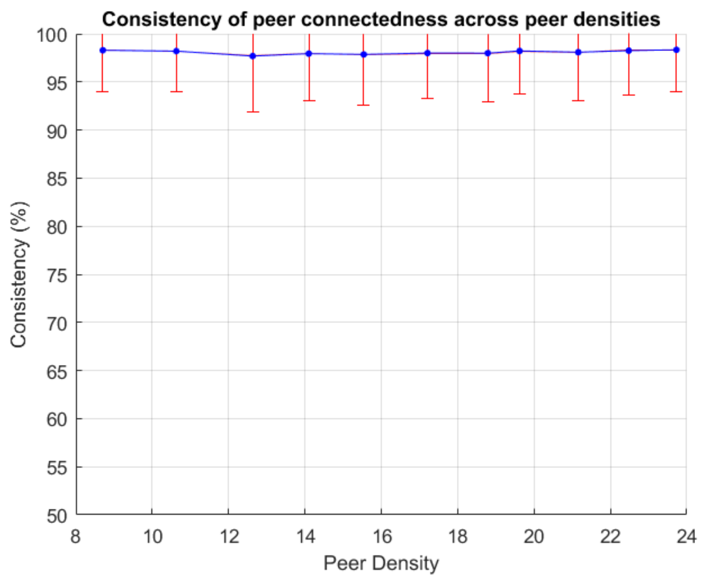


Figure 6.1.5: Consistency of topology as a result of peer density

Drift Distance

In figure 6.1.6, we can see that the drift stays consistent but has a very large standard deviation in the 20's. The negative drift numbers indicate a drift in the other direction and do not indicate negative distances. The standard deviation stays fairly consistent as well, which shows scalability. The large drifts are a result of the latency being high as we can see that at the lower densities, the drift is lower but once the latency climbs to a level above 100 ms, the drift jumps to around 5. The absolute value of the drift distance shows how the drift distance reaches an average of 11.1 units, but stabilises around this range. This is a sub-ideal result in terms of the working of the system, as the drift is quite high, yet it does prove scalability with respect to PD changes, which is sufficient.

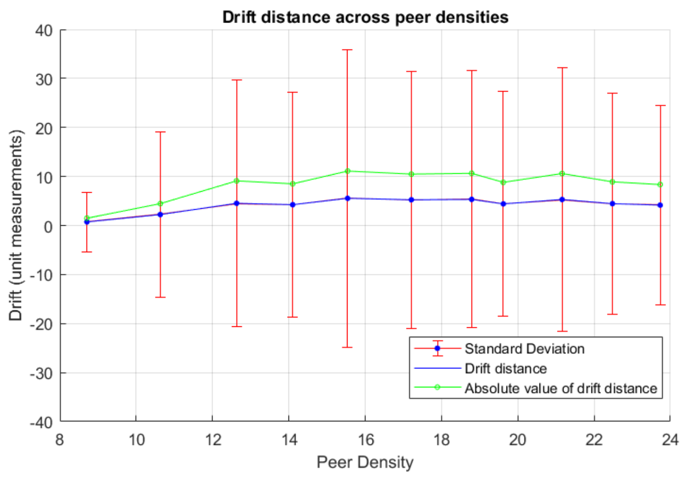


Figure 6.1.6: Drift distance relative to peer density (negative numbers indicate opposite direction)

Stellenbosch University https://scholar.sun.ac.za

6.1.4 Test 2: Varying the number of peers at a set density

This second test investigates the effect of varying the number of peers in the system at different density ranges. These effects are measured using the same metrics as in the previous test.

The running of the test is the same as the previous test - except that instead of the radius being adjusted to change the density with the same number of peers, both the number of peers and radius is adjusted. The change of radius is needed to keep a relatively constant density so that the change in the number of peers is the only variant. Below in figure 6.1.7 one can see the different densities for the two different iterations of the same test where two varied ranges of densities were tested. The red line represents the higher density (13-16) and the blue line represents the lower density (11-14).

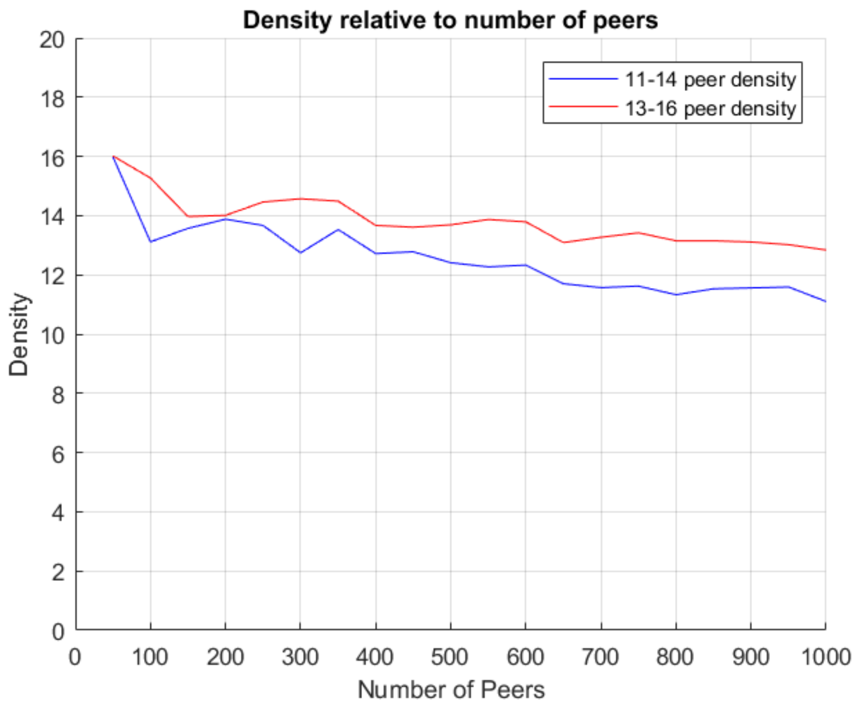


Figure 6.1.7: Density levels of tests across the number of VON peers

Bandwidth

One can see in figure 6.1.8 that the bandwidth does not follow any regular pattern. There are peaks at 200 and 650 peers from the higher and lower densities respectively. This is due mainly to the complexities of how often peers connect and disconnect and the checks involved, rather than it being dependent on density or the number of peers in the system. Since it is not dependent on density or the number of peers in the system and the trend is not increasing, we can say that the bandwidth is not a limiting factor in the scalability of the system.

Stellenbosch University https://scholar.sun.ac.za

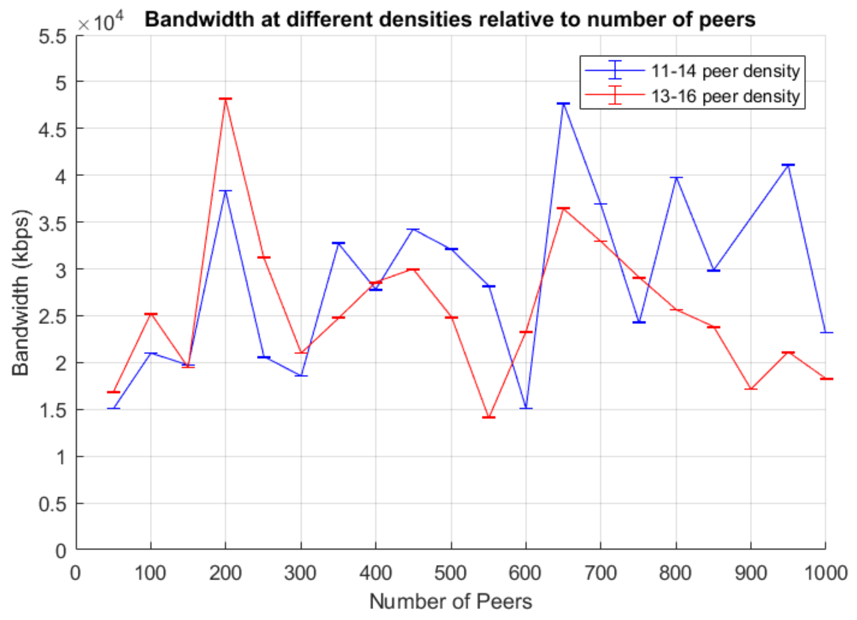


Figure 6.1.8: Bandwidth relative to number of VON peers

Round-trip Time

The round trip time, seen in figure 6.1.9, is seen to stay consistent at a very low latency (in the microseconds). This is, again, evidence of the fact that the network was not saturated and therefore did not affect any other metrics in this test.

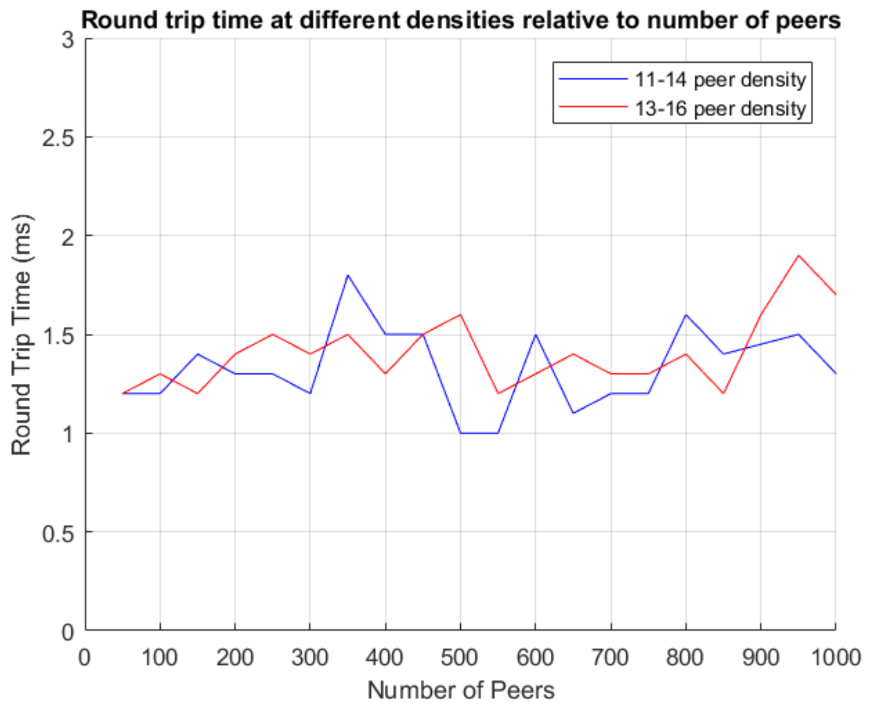


Figure 6.1.9: Round-trip time relative to number of peers

Stellenbosch University https://scholar.sun.ac.za

Latency

The latency is consistent across the number of peers for both density ranges, as seen in figure 6.1.10. The latency is lower for the lower number of peers due to the nature of Voronoi diagrams with fewer peers resulting in less fluctuation in neighbours and therefore fewer computations for neighbour recalculation. At higher peer counts, the latencies stay consistent with only a 9 ms difference between the 1000 peer test and 250 peer test for the blue line and a 10 ms difference between the 850 peer test and 250 peer test for the red line.

For the red line, we see an outlier at 600 peers. This is because of an inconsistency in the running of that specific test. The inconsistency was in the performance of the test bench. It does show, as will be discussed later, the effect of latency on drift distance and is the reason the result was not omitted or redone. It is not, however, taken into account in the discussion of the indication of the latency results effect on scalability.

The increase in the last three latency data points for the red line is due to the system not being able to handle the processing of the movement of that number peers at that density. This shows the importance of maintaining a PD that the system can handle as the 11-14 PD line could handle the movement of the peers at these peer count. The general trend, however, shows the system to be scalable at the lower density.

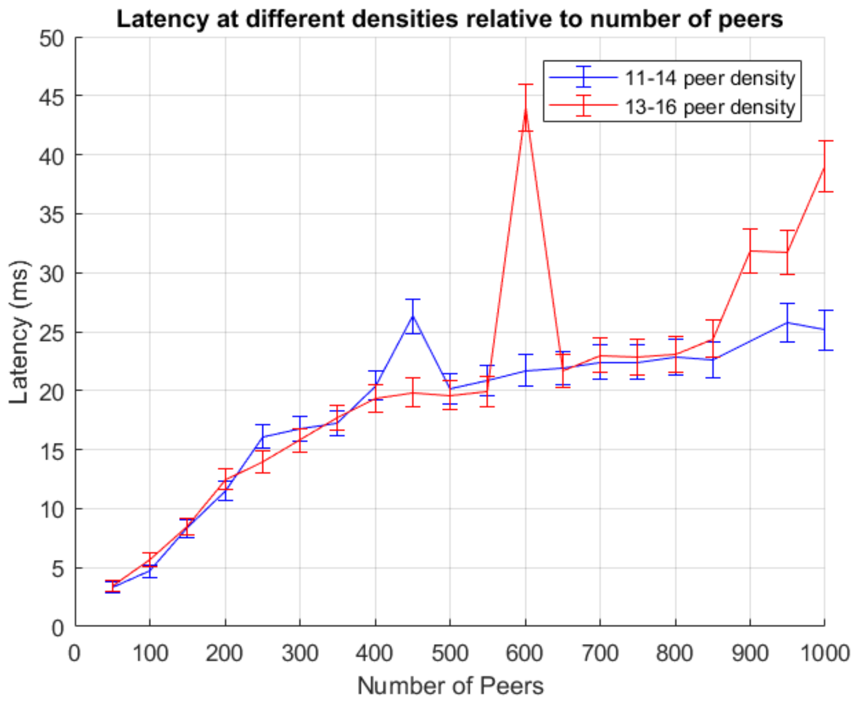


Figure 6.1.10: Latencies relative to number of peers

Consistency

In figure 6.1.11, we can see that the consistency stays above 98% for both density ranges tested across the varying number of peers. The slight decline is due to a higher rate of change in neighbours due to a greater concentration of peers in the VE. This is seen to

Stellenbosch University https://scholar.sun.ac.za

be scalable as although the consistency of some clients is below 90% (as seen by the high standard deviations) and these consistency levels are acceptable but sub-optimal, the average consistency did not drop below 98.6% and therefore means that the majority of the clients experienced 100% consistency. Whilst this is not fair to some of the clients experiencing lower consistency, this lower consistency is distributed randomly and can happen to any of the clients, which makes it fairer as one client would not always experience low consistency.

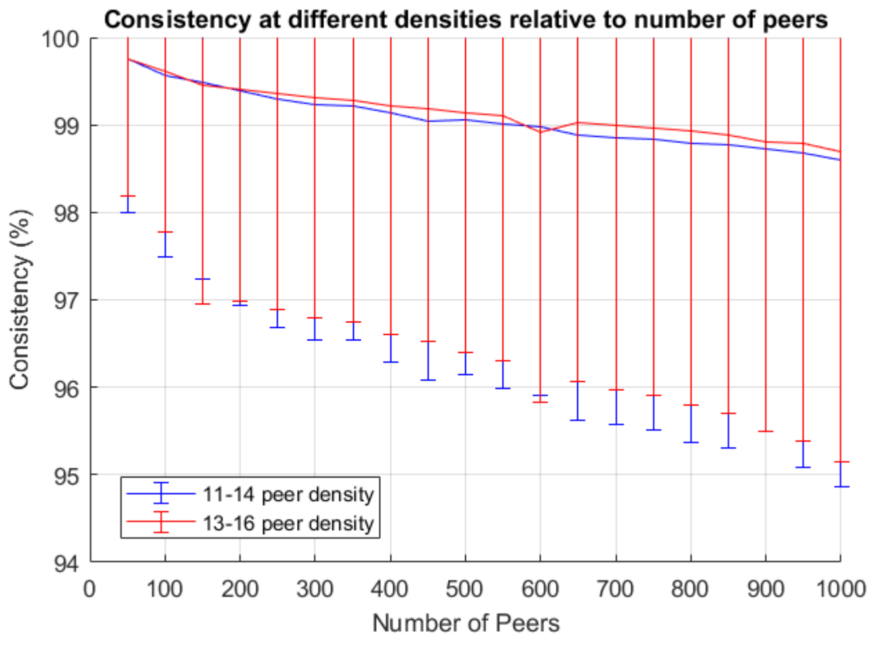


Figure 6.1.11: Consistency across number of peers

Drift distance

The drift distance is seen to be fairly consistent with a slight increase in variance when the density range is increased, as seen in figure 6.1.12. A closer view of the results is shown in figure 6.1.13. The average drift distance is quite low, staying below 1, except for the red line which jumps at 600, 900, 950 and 1000 peers. This is consistent with the latency at these peer numbers. The 600 peer test results (which was the poorest result in the test) were induced by testing the system in a worst-case scenario. This worst-case scenario is when single peers are heavily overloaded and fall behind in the tasks that need to be processed and therefore increase the latency which increases the drift distance. This is considered an outlier and can be seen in results surrounding it that are vastly different. These results are consistent when observing the absolute value of the drift distances but at a higher average level.

We can see that an increase in latency results in a direct increase in drift distance standard deviation and average as the peer takes longer to process movements and therefore lags behind the movement updates sent through by the peer, resulting in an outdated view of neighbour positions. This is seen to be scalable with an increase in peers in the system.

Stellenbosch University https://scholar.sun.ac.za

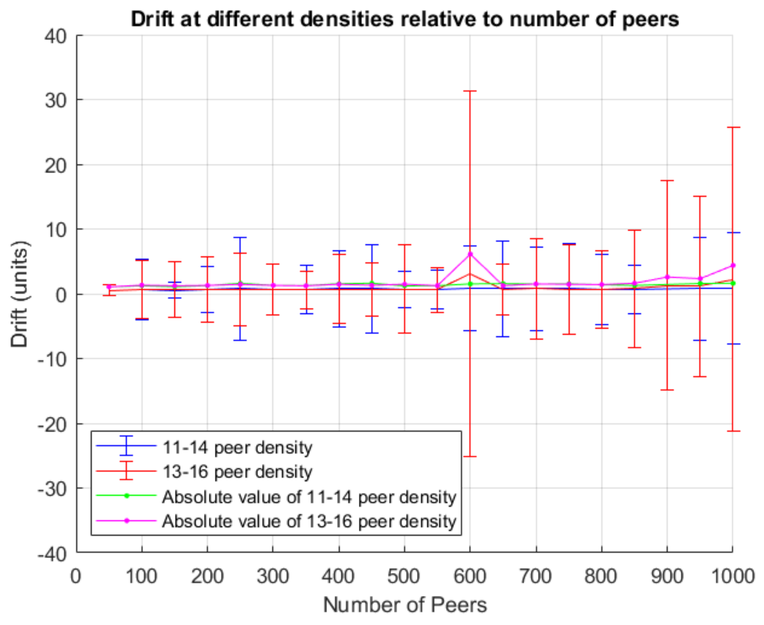


Figure 6.1.12: Drift distance versus number of peers for a set peer density

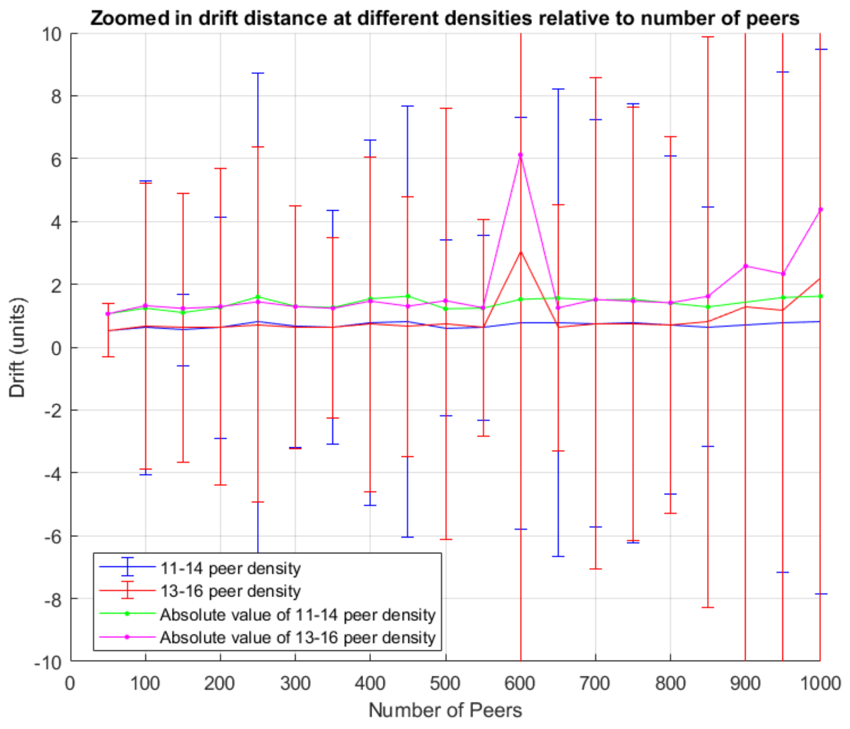


Figure 6.1.13: Zoomed-in figure of drift distance versus number of peers for a set peer density

Stellenbosch University https://scholar.sun.ac.za

6.2 VAST scalability test

With a scalable networking system serving as the underlying architecture for VAST, the update and event dissemination system that is added on top of this layer needs to continue this scalable trend in order for VAST to be considered scalable. This test aims to show that the entire VAST ecosystem can work cohesively as a scalable whole.

6.2.1 Test overview

This test aims to determine the effect of the number of clients versus the number of matchers in the system and the performance benefits of adding more matchers to determine the scalability potential of VAST.

The test first utilises a single matcher and increases the number of clients to determine the limits of the system. Then it increases the number of matchers in multiples of five whilst testing over the same range of client increases to determine the benefits of having more matchers in the system.

6.2.2 Test setup

A single moderately powerful computer is used to host the system (specifications in appendix B). An entry server is hosted on a Node.js server and listens on a single port as a connection point for the clients. The VAST clients are then started, first with the gateway client and then with the following clients connecting to the system through the gateway. Once all of the VAST clients are connected to the system, clients are connected to the system. These clients are clients developed to behave like Minecraft clients but without the stringent aspect of the Minecraft protocol. The clients are given a random waypoint that they move to within ten seconds. These movements are made every 50ms and are the distance of the movement is calculated in equation 6.2.1. The $MOVEMENT\_SPEED$ constant is defined as 500 in this test.

$$Movement = (destination - startingposition)/MOVEMENT\_SPEED \qquad (6.2.1)$$

The client sends movement packets and publications at the same interval as the movements. This approximates the network traffic that a single Minecraft client would send in a normal session. These clients connect one-at-a-time, with each client beginning its connection procedure once it gets an acknowledgement from the previous client's join success callback. The test is run for two minutes once all of the clients have joined and then all of the clients collectively disconnect from the system.

The clients log the packets that they send and receive. This is used to determine the performance of the system. These packets contain the position that the packet is published to, the client that sent the packet and the type of packet. When the packet is received, the recipient is logged as well. The test is concluded and the results are processed thereafter.

Stellenbosch University https://scholar.sun.ac.za

6.2.3 Performance evaluation

The results of the testing of the system are shown below. First, the latency experienced by the clients is discussed, followed by the drift distance of the clients within the system. Finally, the consistency of the client connectedness is discussed.

Latency

In figure 6.2.1, one can see the average latency that each client experiences as a function of the increasing number of clients in connection to the system. This is shown in different coloured lines depending on how many matchers were in the system.

As one can see, the latencies were in the 0-5ms range at every point except for four spikes. The first two, at 250 and 300 clients for five matchers, shows the limits of the five matchers. When the connection process of the clients reaches a level where the matchers are constantly having to adjust in order to not be overloaded due to incoming connections, the clients who are already connected to the system are frequently being transferred to other matchers. This process, in conjunction with the high amount of packets being sent, results in the system not being able to process events fast enough and therefore results in latency. Five matchers could not handle 350 clients and therefore there is no data for it.

The other spikes are due to the positioning of the join locations of the clients in the test. Since the joining positions of the clients are random, a large number of clients can connect to a specific matcher's region. This results in the matcher being overloaded more quickly than usual and results in some initial latency as the matchers rearrange themselves and clients are shifted to other matchers. This can be seen by the small spike for 35 matchers at 300 clients and 30 matchers at 350 clients.

One can see that for every other test, the latencies were minimal. The peaks for the other tests could not be reached due to the test bench not being powerful enough to handle all of the separate entities without running out of resources.

As one can see, when a matcher reaches its limit, the latency increases drastically but when that limit is not reached, the latency stays at the same level no matter how many matchers are in the system. The limit is reached at 5 matchers, but adding more matchers allowed the system to perform better. This can be seen as a scalable response with respect to latency.

Stellenbosch University https://scholar.sun.ac.za

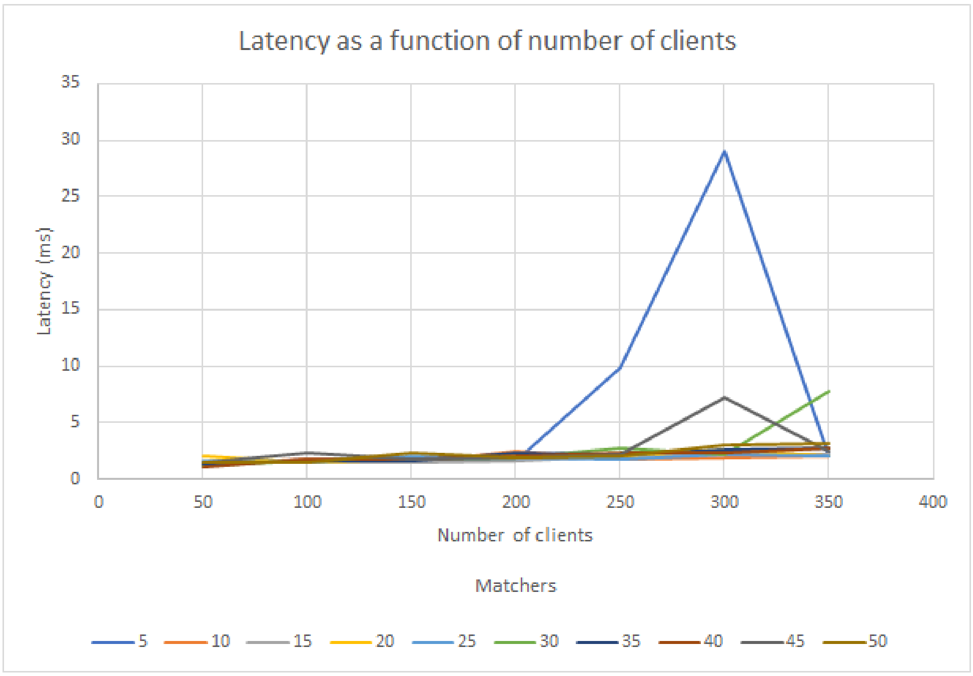


Figure 6.2.1: Average latency that each client experiences as a result of the increase in clients across different numbers of matchers

Drift distance

In figure 6.2.2, the absolute value of the drift distance of the clients is shown. This drift distance is given as a function of the number of clients connected to the system. Each colour of line displays the number of matchers that were in the system.

As one can see, at 50 clients the drift distance was the most erratic across the different number of matchers, ranging from 0-0.0034 units. These distances are relative to distances within a Minecraft world. They are different from the drift distances shown in figure 6.1.13 as this is the client drift distance and not the VON peer drift distance. The more clients that were in the system, the more the lines converge. The reason for this is that when a client moves, the precision of the movement is recorded to a much higher degree on the moving client as opposed to the client receiving the positional update. The more clients that there are, the less these small inconsistencies matter and therefore the more the lines converge. This is not affected by the number of matchers and is the reason that the number of matchers does not affect this metric. If the latency were to increase to the point where it is higher than the frequency of the positional updates (ie the latency is higher than 50 ms) then the drift distance would increase in kind as the positional updates would be received late.

This figure 6.2.2 shows that the density is a result of the latency of the system and therefore shows scalability as long as the latency is scalable.

Stellenbosch University https://scholar.sun.ac.za

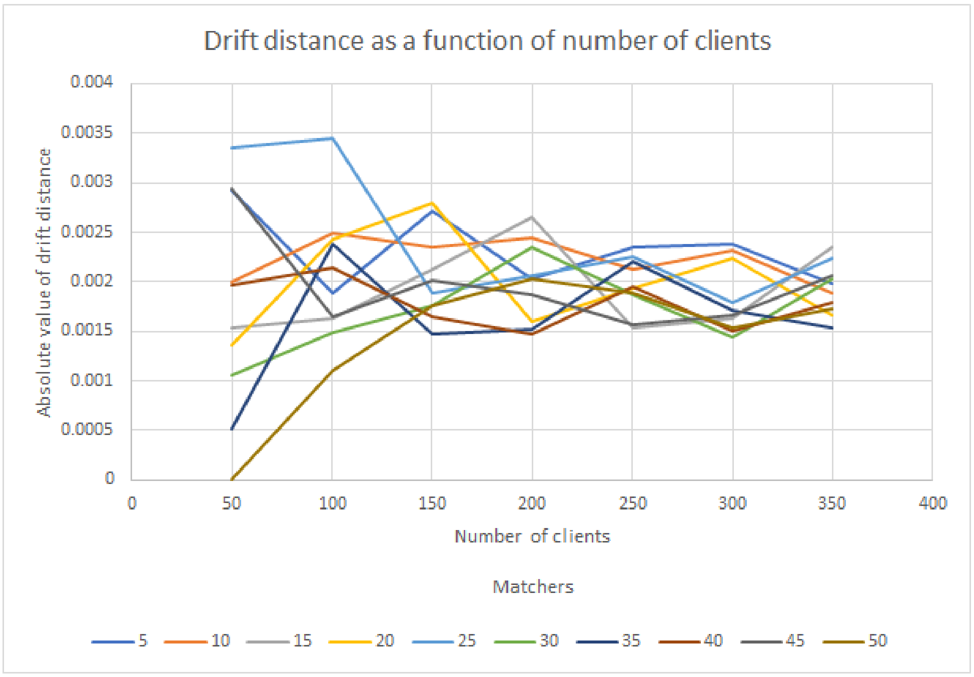


Figure 6.2.2: Drift distance of clients across different numbers of matchers

Consistency of client connectedness

Figure 6.2.3 shows the consistency of clients' connectedness to each other as the number of clients increase and with different numbers of matchers within the VAST system.

Immediately, it is observable that the slope of the consistency decreases as the clients increase, as does the consistency with an increasing number of matchers. At 50 clients, 5 matchers resulted in consistency of above 98% whereas for 35 matchers the consistency was at just above 82%. At 300 clients, 5 matchers were at a consistency of 60% whereas 45 matchers were at a consistency of 36%.

These consistencies are not desirable but are a result of the transferring of clients from one matcher to another. The more clients that there are, the more clients disconnect from one matcher and reconnect to another matcher or the more chance there is for a single matcher to get overloaded and need the other matchers to move closer to it to resulting in clients taking longer to be reconnected to the system. Similarly, the more matchers there are, the smaller the regions for each matcher are and therefore the more likely the clients must transfer from one matcher to another. This disconnection time during the transfer from matcher to matcher is the reason that the consistency is lower as the client is briefly not a part of the system, resulting in a disconnection from its peers and a drop in consistency.

To fix this, the implementation needs to be adjusted so that the client stays connected to the matcher until the handover has been completed. This will allow the client to stay connected to the system and not result in a consistency drop when being transferred to a different matcher.

Stellenbosch University https://scholar.sun.ac.za

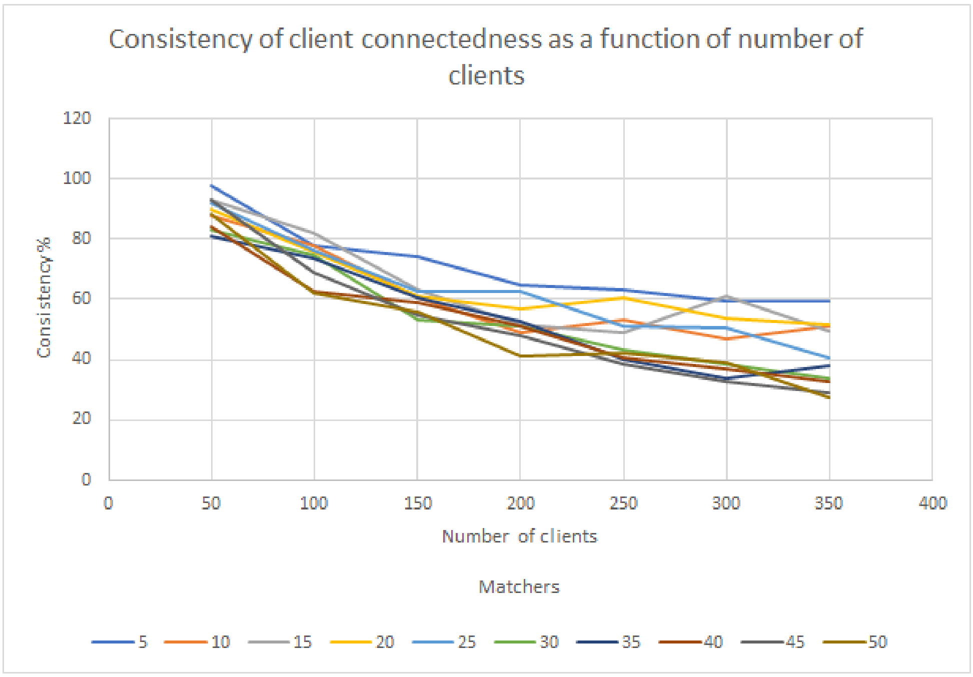


Figure 6.2.3: Consistency of client connectedness across different numbers of matchers in VAST

The results do show, however, that the consistency differs less as more matchers are added. In the bottom five lines at 350 clients, represented by 30, 35, 40, 45 and 50 matchers, are all clustered within 10% of each other as opposed to the top four lines, represented by 10, 15, 20 and 25 matchers, which are spread across a 20% range. This shows that the system is reaching a saturation level where the number of matchers is no longer the issue for consistency. This is due to the number of clients that a single matcher can handle stabilising, and results in fewer movements of the matchers which increases client connectivity time and results in a stabilisation of the consistency loss.

Whilst this consistency graph does not show scalability, the proposed implementations should increase the consistency of the system and result in better performance.

6.3 Minecraft integrated into VAST

In this test, the performance of Minecraft with the event and update dissemination being handled by VAST is evaluated. The test overview is described as well as the test set-up, followed by a discussion of the results and finally the conclusion.

6.3.1 Test overview

This test aims to determine whether Minecraft can function when using VAST as an SPS system to disseminate updates and events and whether it can do so with playable performance.

Stellenbosch University https://scholar.sun.ac.za

The test consists of a Spigot Minecraft server with a plugin that logs the movements of players that are connected to the system. It also has a server and client proxy that serves to interpret between the Minecraft server and VAST and the emulated Minecraft clients that connect to the system. These emulated clients are light-weight clients that are command-line based and behave and work exactly as a normal Minecraft client would. Finally, the different number of VAST nodes needed are run.

6.3.2 Test setup

The test commences with the starting of the Spigot server. The entry server is started and it listens for incoming connections on a specific port (2999 in this case). The desired number of VAST nodes are then started according to what is required in the test and the server and client proxy are started, giving them the required ports that they need to connect to. The server proxy connects to the Spigot server and the entry server whilst the client proxy connects to the entry server and the Minecraft clients connect to this. Once all of this has started up, the desired number of clients is connected to the system and the test is run for two minutes before the clients are disconnected one-at-a-time. This setup is shown in figure 6.3.1

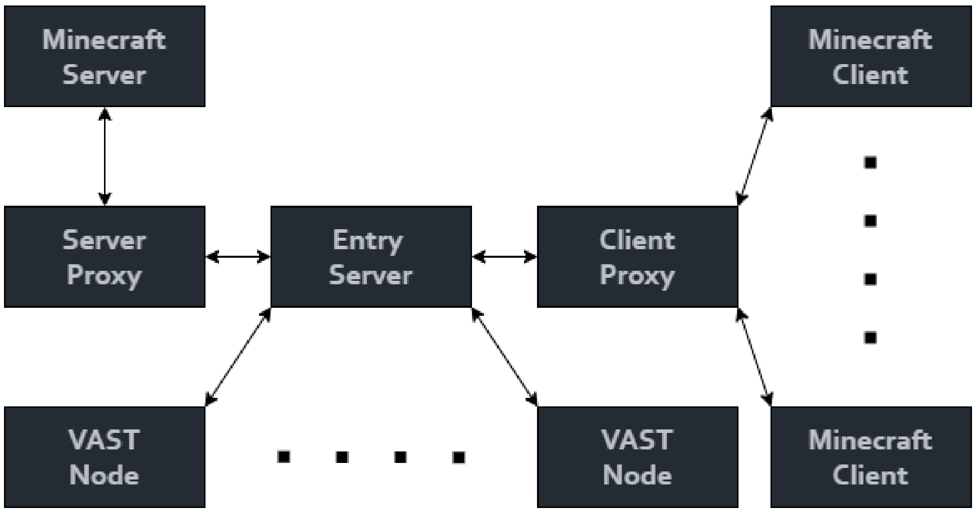


Figure 6.3.1: Test set up for the integration of Minecraft and VAST

This test uses a moderately powerful computer to host all of this (described in appendix B). There is no result for the five matcher 35 client test due to the lack of access to additional computational capacity in the test bench.

6.3.3 Performance evaluation

The results of the test are shown below. First, the latency results are discussed, followed by the drift results and finally, the consistency of the system is discussed. More matchers and more clients could not be added due to test bench limitations as well as time limitations.

Stellenbosch University https://scholar.sun.ac.za

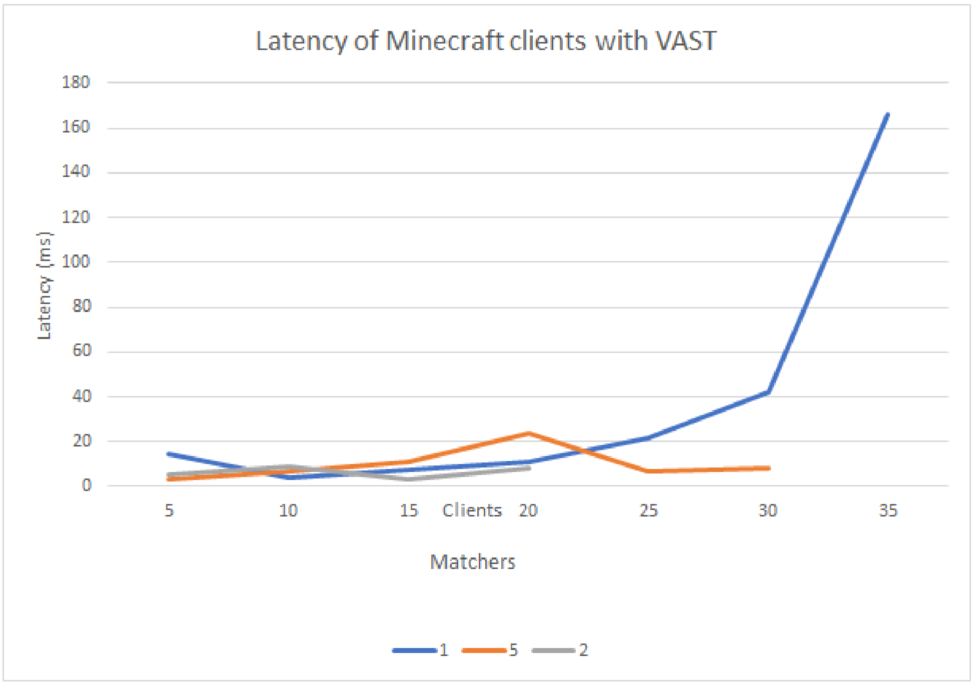


Figure 6.3.2: Latency of Minecraft clients with different number of VAST nodes

Latency

As can be seen in figure 6.3.2, the latency is plotted against the number of clients in the system, with each line representing the number of VAST nodes joined to the system.

The latency for a single matcher climbs dramatically when it reaches 30-35 clients whereas, for five matchers, the latency decreases from 20 to 25/30 clients. This shows that the system performs better when there are more matchers due to all of the traffic being distributed between the matchers instead of being processed by a single matcher.

When there is a single matcher, the latency climbs to levels that are unacceptable for normal play, however, for five matchers the latency is low and the game is playable. This shows the viability of Minecraft running through VAST when it comes to interactivity due to latency and shows that there is potential for the system to grow even further than the test bench allows. It serves as a proof of concept that this could be taken even further.

Drift

Figure 6.3.3 shows the absolute value of the drift distance that the clients perceive of other clients. This is plotted against the number of clients and the different lines represent the number of VAST nodes within the system.

As one can see, the drift distance for a single matcher starts quite low at 0.12 units and climbs to 0.34 units at 15 matchers before levelling off. This shows that the drift distance is not affected by the number of clients in the system for the ranges tested. The reason for this is that even though the latency increased, the subscription was still being updated quickly enough on VAST so that the publication of updates continued to reach the clients.

Stellenbosch University https://scholar.sun.ac.za

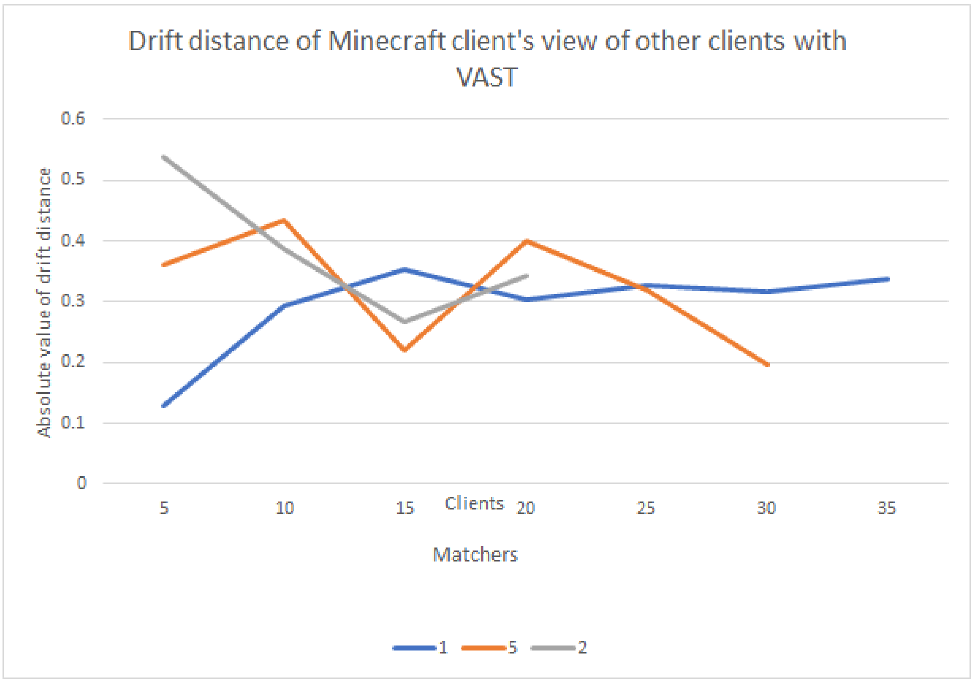


Figure 6.3.3: Drift distance of Minecraft clients connected through different number of VAST nodes

For five matchers, the drift distance stays relatively stable and decreases for 15, 25 and 30 clients to being less than the drift distance of a single matcher. This is a result of the clients crossing boundary thresholds less often and therefore not being disconnected for as long. This results in the clients having a more accurate view of where the other players are. The distribution of load across matchers also results in clients receiving information faster as opposed to a single matcher. This can be seen in the lower latencies in figure 6.3.2 for 25 and 30 clients for 5 matchers relative to a single matcher. This shows that from what was tested, the number of matchers decreases the drift distance of the clients.

Consistency

Figure 6.3.4 shows the consistency of client connectedness between the Minecraft clients when there is an increasing number of Minecraft clients connected to the system. The different lines show the number of VAST nodes in the system at the time.

The consistency of a single matcher is satisfactory, remaining above 95% for the entire range of clients. This means that as the latency increased since the drift distance stayed constant, the clients still had an accurate view of where the other clients were and therefore still received updates from them as they should have.

The consistency of five matchers decreased for 25 and 30 Minecraft clients. As before in section 6.2.3, the more matchers that there are in the system, the more chance that a client can be transferred from one matcher to another and therefore the more time it spends disconnected from the system. This results in the consistency decreasing due to

Stellenbosch University https://scholar.sun.ac.za

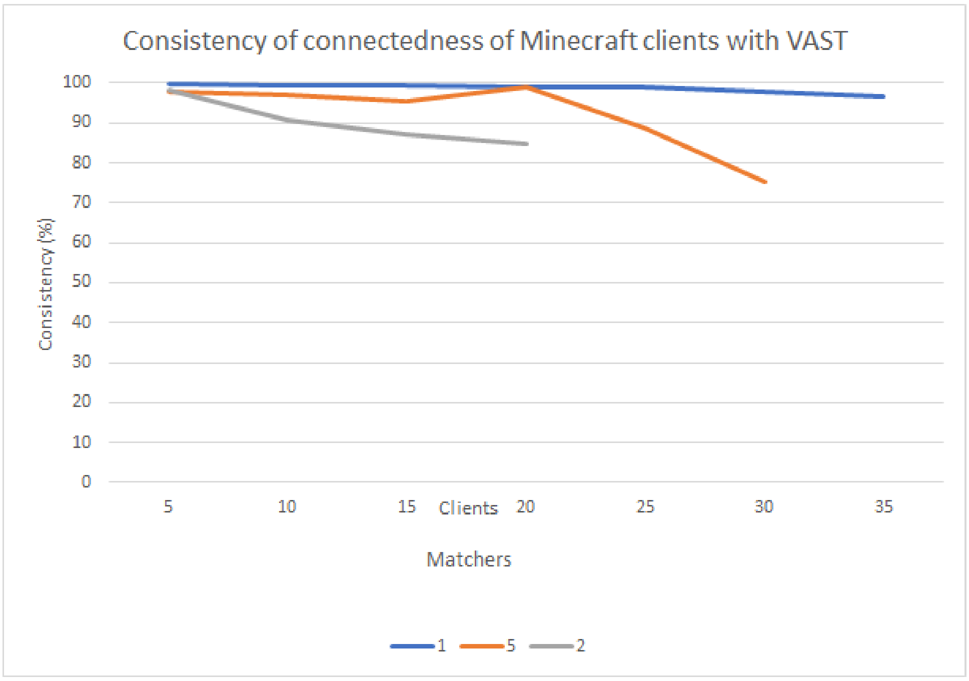


Figure 6.3.4: Consistency of Minecraft client view when connected through different numbers of VAST nodes

the client no longer sending and receiving updates until it reconnects to its destination matcher.

The consistency results show that Minecraft is playable with multiple clients and whilst the consistency is lowered to 70% when multiple matchers are introduced, the game is in a playable state. By changing the implementation in the future and allowing the Minecraft client to stay connected to the system whilst being transferred, the consistency could be improved.

6.3.4 Summary of Minecraft results

As a whole, the Minecraft results when running through the VAST system show that Minecraft is playable and that by adding more matchers, some results improve. With the proposed changes and solutions, it is believed that all results could be improved. In theory, this allows more clients to join the system than when a single matcher is in the system. The latency improved as more matchers were added to the system. Additional matchers allowed for acceptable levels of latency. Drift distance decreased when more matchers were added and showed acceptable levels whether there was a single matcher or multiple matchers. The consistency was the only metric that suffered when more matchers were introduced but a solution was given to rectify the results to improve the system in the future. The proposed solution is to have the clients stay connected to the matchers when transferring from one matcher to another rather than be disconnected during the transfer.

Stellenbosch University https://scholar.sun.ac.za

6.4 Summary

This chapter first tested the performance of the VON layer and whether it displays scalable properties. It looked at the bandwidth, RTT, latency, drift distance and consistency of the system. It tested for a varying peer density to see the effect on scalability as well as varying the number of peers at a set density to observe the effect that that had. From these results, VON was seen to be scalable.

The next evaluation was that of VAST as a whole, with the entry server included. The drift distance, latency and consistency of the clients were evaluated. The drift distance and latency showed scalable results, whilst consistency showed diminishing returns. This was reduced down to the disconnecting of clients when being transferred between matchers and a recommendation of how to improve it was given.

Finally, Minecraft integrated into VAST was evaluated. The drift distance, latency and consistency were observed and discussed. The latency showed improved performance when more matchers were added to the system whilst the drift distance decreased when more matchers were added. The consistency decreased as expected when more matchers were added due to the same reason of clients disconnecting when being transferred between matchers.

The next chapter concludes this thesis by summarising the aforementioned research and results and suggests future work that could be done to improve the system.

Stellenbosch University https://scholar.sun.ac.za

Chapter 7

Conclusion

7.1 Introduction

This thesis aims to integrate Minecraft with an SPS system to replace its update and event dissemination. This is done to show that a traditional MMVE can be made to be scalable, and could furthermore benefit from using VAST to handle its message dissemination. MMVEs were investigated to determine the key requirements that are common to the genre as well as the major obstacles currently faced by researchers. VAST as a system was discussed and the design of the different aspects of the system were explained. To test VAST's performance in a real-world environment, Minecraft was adapted to integrate with VAST. The viability of Minecraft benefiting from SPS functionality was investigated before the performance of the system was evaluated. Firstly, the different layers of VAST were tested to explore how scalable the system was and then the integration between Minecraft and VAST was evaluated to see how well it performed.

7.2 System performance

The first test was whether Minecraft would benefit from an SPS update and event dissemination method. It was found that 95% of packets that were sent in a normal player session were suitable to be sent as a publication with a specific location. For two clients it was found that the 95% of SPS-viable packets could be reduced by 33.52% and would only increase with more clients in the system.

The second test was for VON's scalability potential in isolation to the rest of the Minecraft system. It was evaluated to be scalable at a peer density of 15 with the bandwidth amounting to 24.5 KB/s, an RTT of 2ms, a latency of 150ms which could be improved with a better forwarding algorithm, a consistency of above 95% and a drift distance of just over 3 units.

The next test evaluated VON's scalability when increasing the number of peers at the set peer density of 16 and below. The bandwidth stayed within the 1.5-4.75kbps range, the RTT stayed below 2ms, the latency stayed lower than 45ms, the consistency higher than 98% and the drift distance remained under 5 units. However, the drift distance standard deviation was high as a result of the increased latency but was seen to be acceptable for

Stellenbosch University https://scholar.sun.ac.za

scalable purposes due to the drift distance standard deviation staying within a constant band of 30 units. The layer was determined to be scalable.

The next test evaluated the scalability of VAST as a whole. It used clients that simulated a Minecraft client's network traffic to stress test the VAST system. The latency, consistency and drift distance were measured. The latency spiked when the 5 matchers got overloaded but stayed low when increasing the matchers to 10-50. The drift distance converged to a range of 0.0008-0.0012 units when more matchers were added and consistency decreased from a range of 80-100% to 27-60%. This decrease was attributed to the disconnecting that a client experiences when transferring across matchers and could be rectified if the client stayed connected until the transfer to the new matcher was complete. The system was determined to be scalable for the ranges that the test bench could handle, which was 50 matchers with 350 clients.

Finally, the integration of Minecraft with VAST was tested. The latency, consistency and drift were evaluated. The latency increased for one matcher as it hit its limit at 30 clients within the system, but stabilised when five matchers were introduced into the system. The drift distance was unaffected due to the latency not being high enough for the clients to miss positional updates. The consistency dropped in the same way that it did for the previous test for the same reason of clients being disconnected from the system for a small period of time. The metrics showed that the game was playable and therefore that the system was a success.

7.3 Concluding analysis

The objectives stated in 1.7 were partially met and are each discussed below.

7.3.1 VAST port from C++ to JavaScript

VAST was successfully converted to JavaScript in that it is working with the same level of functionality that the C++ version did. However, the conversion is not optimised to its full potential. Optimisations that should be implemented include:

- Changing the forwarding algorithm of VON's acceptor finder to search more efficiently.
- Changing the way that messages are routed from client to matcher so that they do not have to go through the entry server. This requires a modification of the API so that the client can reconnect to a different matcher whilst already being connected to the system.
- Allow for multiple ESs so that the load of multiple TCP connections does not overload any single ES (load balancing on ES). It could be another layer in VAST controlled by a Voronoi diagram.

This objective is considered to be achieved based on these conclusions.

Stellenbosch University https://scholar.sun.ac.za

7.3.2 VAST showing scalable properties

As discussed in section 7.2, VON was determined to be scalable in all of the tests that were performed on it. VAST as a whole was determined to be scalable for the ranges that the test bench could handle. Larger tests using more powerful and robust test benches could be done to observe how the system performs with greater client and matcher scales. However, in theory, and the observable results given in this thesis the system will be able to handle these larger tests, and this objective is considered to be met.

7.3.3 SPS viability and benefits in Minecraft

Section 5.3 describes the investigation into the viability and potential of Minecraft using SPS for its update and event dissemination. This found that 95% of packets could be published with coordinates and at minimum, the reduction in packets using SPS for a normal usage session is 33.52%. This is a large reduction in network traffic and Minecraft could, therefore, benefit from using SPS for its event and update dissemination.

7.3.4 Interface between Minecraft and VAST

The client and server proxies developed allowed Minecraft packets to successfully be published and for subscriptions to be made. This objective has therefore been met.

7.3.5 Minecraft functioning through VAST

Minecraft could successfully be played through VAST in its most basic form. However, the reduction in packets could not be implemented due to foundational work that is needed to be done on the Minecraft server to change the way that it handles player connections. The player connections would need to be independent of player sessions so that the Minecraft server could publish a single packet for an event that happens rather than multiple packets for individual player connections. Minecraft was also not implemented with Koekepan. The server modifications are necessary before Koekepan would benefit from the scalability of Minecraft integrated with VAST. This objective is partially met, yet further work in future research could be aimed at improving the success of the system in this area.

7.4 Future work

The recommended future work is as follows:

- The VAST system could be optimised by replacing the greedy forwarding method of the VON layer's **JOIN** procedure with a procedure that produces fewer hops and finds the acceptor more efficiently.

- The routing of information through the entry server and the bottleneck that it would inevitably create could be circumvented by creating a way to have multiple entry servers as well as a way for the entry server to hand over the connection it has with a client to the relevant matcher so that all Minecraft packets do not get routed through the entry server. This would leave the entry server to only find the relevant matcher for the connecting clients and manage the positions of load balancing matchers.

Stellenbosch University https://scholar.sun.ac.za

- The transferring of clients could be streamlined to not disconnect completely from the system whilst in the process of being transferred to another matcher.
- The system could be properly integrated with the Koekepan system so that it can be truly scalable with access to multiple servers running the same Minecraft world.
- The Minecraft server needs to be modified so that it fully utilises the SPS functionality that it has been evidenced to be compatible with. This would involve going to each point in the code at which updates are sent to multiple clients and change it so that it only sends a single update to an SPS connection that is made. It would also need to make only a single connection, as opposed to a single connection for every client that connects to it. This is essentially an amalgamation of the server proxy and the Minecraft server. Until this is done, the Minecraft server cannot fully utilise the power of SPS.

While the Minecraft server cannot fully utilise the power of SPS currently, the benefits presented in this thesis show how MMVEs can utilise SPS and VAST to become more scalable.

Stellenbosch University https://scholar.sun.ac.za

Appendix A

Cluster computer specifications

Table A.1: Specifications of cluster test computer

Part	**Specification**
CPU	Intel Core i5-2400S CPU @ 2.5GHz (4 cores)
RAM	3.7 GB
Network interface	Intel 82579LM Gigabit Network interface
OS	Linux Mint 18.3 Cinnamon 64-bit (version 3.6.6)
Kernel	Linux Kernel 4.10.0-38-generic

Stellenbosch University https://scholar.sun.ac.za

Appendix B

Single computer specifications

Table B.1: Specifications of moderately powerful test computer

Part	**Specification**
CPU	Intel Core i7-8700K @ 3.7GHz base (12 cores, 64-bit, 12MB cache, 4.7GHz Max frequency)
RAM	16 GB
Graphics Card	NVidia GeForce GTX 750 (2GB GDDR5 VRAM)
OS	Windows 10 Professional

Stellenbosch University https://scholar.sun.ac.za

Appendix C

Source code

Source code for VAST can be found at `https://github.com/imonology/VAST.js`.

Source code for the Minecraft proxies can be found at `https://bitbucket.org/hebrecht/herobrineproxy/src/v1.11.2_sps/` on the v1.11.2_sps branch at the time of writing.

Stellenbosch University https://scholar.sun.ac.za

Bibliography

[1] Hu, S.-Y., Chen, J.-F. and Chen, T.-H.: Von: a scalable peer-to-peer network for virtual environments. *IEEE Network*, vol. 20, no. 4, pp. 22–31, July 2006. ISSN 1558-156X.

[2] Hu, S.-Y., Wu, C., Buyukkaya, E., Chien, C.-H., Lin, T.-H., Abdallah, M., Jiang, J.-R. and Chen, K.-T.: A spatial publish subscribe overlay for massively multiuser virtual environments. *2010 International Conference on Electronics and Information Engineering*, vol. 2, pp. V2–314–V2–318, 2010.

[3] Engelbrecht, H.A. and Schiele, G.: Koekepan: Minecraft as a research platform. *Annual Workshop on Network and Systems Support for Games*, pp. 10–12, 2013. ISSN 21568146.

[4] Hu, S.-Y. and Liao, G.-M.: Scalable peer-to-peer networked virtual environment. In: *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '04, pp. 129–133. ACM, New York, NY, USA, 2004. ISBN 1-58113-942-X.
Available at: `http://doi.acm.org/10.1145/1016540.1016552`

[5] Kawahara, Y., Morikawa, H. and Aoyama, T.: A peer-to-peer message exchange scheme for large scale networked virtual environments. In: *The 8th International Conference on Communication Systems, 2002. ICCS 2002.*, vol. 2, pp. 957–961 vol.2. Nov 2002.

[6] Bourke, P.: Circles and Spheres. 1992.
Available at: `http://paulbourke.net/geometry/circlesphere/`

[7] Hu, S. and Chen, K.: Vso: Self-organizing spatial publish subscribe. In: *2011 IEEE Fifth International Conference on Self-Adaptive and Self-Organizing Systems*, pp. 21–30. Oct 2011. ISSN 1949-3673.

[8] Entertainment Software Industry: Essential Facts About the Computer and Video Game Industry 2019. *Entertainment Software Association*, 2019.
Available at: `https://www.theesa.com/wp-content/uploads/2019/05/ESA_Essential_facts_2019_final.pdf`

[9] Entertainment Software Industry: Essential Facts About the Computer and Video Game Industry 2018. *Entertainment Software Association*, 2018.
Available at: `https://www.theesa.com/wp-content/uploads/2019/03/ESA_EssentialFacts_2018.pdf`

[10] Cecin, F., Jannone, R., Geyer, C., Martins, M. and Barbosa, J.: Freemmg: a hybrid peer-to-peer and client-server model for massively multiplayer games. p. 172. 01 2004.

[11] Electronic Arts: Black & White 2.
Available at: `https://www.ea.com/games/black-and-white/black-and-white-2`

[12] Electronic Arts: Spore. 2009.
Available at: `https://www.spore.com/`

Stellenbosch University https://scholar.sun.ac.za

[13] Official Minecraft Wiki - The ultimate resource for all things Minecraft. 2017.
Available at: `https://minecraft.gamepedia.com/Minecraft_Wiki`

[14] Gilmore, J. and Engelbrecht, H.: A survey of state persistency in peer-to-peer massively multiplayer online games. *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, pp. 1 – 1, 05 2012.

[15] Yahyavi, A. and Kemme, B.: Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Comput. Surv.*, vol. 46, no. 1, pp. 9:1–9:51, July 2013. ISSN 0360-0300.
Available at: `http://doi.acm.org/10.1145/2522968.2522977`

[16] Young, R.D.: Npsnet-iv: a real-time, 3d distributed interactive virtual world. 1993.
Available at: `https://calhoun.nps.edu/handle/10945/40017`

[17] Knutsson, B., Honghui Lu, Wei Xu and Hopkins, B.: Peer-to-peer support for massively multiplayer games. In: *IEEE INFOCOM 2004*, vol. 1, p. 107. March 2004. ISSN 0743-166X.

[18] Bharambe, A., Douceur, J., Lorch, J., Moscibroda, T., Pang, J., Seshan, S. and Zhuang, X.: Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. vol. 38, pp. 389–400. 01 2008.

[19] Funkhouser, T.A.: Ring: A client-server system for multi-user virtual environments. In: *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, I3D '95, pp. 85–ff. ACM, New York, NY, USA, 1995. ISBN 0-89791-736-7.
Available at: `http://doi.acm.org/10.1145/199404.199418`

[20] Diaconu, R. and Keller, J.: Kiwano: Scaling virtual worlds. In: *2016 Winter Simulation Conference (WSC)*, pp. 1836–1847. Dec 2016. ISSN 1558-4305.

[21] Stoica, I., Morris, R., Karger, D., Kaashoek, M. and Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. vol. 149-160, pp. 149–160. 01 2001.

[22] Rowstron, A.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Middleware*, 07 2001.

[23] Lawver, B.: Minecraft Now Has Over 100 Million Monthly Active Players. 2019.
Available at: `https://screenrant.com/minecraft-100-million-monthly-active-players/`

[24] Chen, J.: Locality aware dynamic load management for massively multiplayer games. pp. 289–300. 01 2005.

[25] Schiele, G., SÃ¼selbeck, R., Wacker, A., HÃ¤hner, J., Becker, C. and Weis, T.: Requirements of peer-to-peer-based massively multiplayer online gaming. pp. 773–782. 05 2007.

[26] Claypool, M. and Claypool, K.: Latency and player actions in online games. *Commun. ACM*, vol. 49, no. 11, pp. 40–45, November 2006. ISSN 0001-0782.
Available at: `http://doi.acm.org/10.1145/1167838.1167860`

[27] Beigbeder, T., Coughlan, R., Lusher, C., Plunkett, J., Agu, E. and Claypool, M.: 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, pp. 144–151. ACM, New York, NY, USA, 2004. ISBN 1-58113-942-X.
Available at: `http://doi.acm.org/10.1145/1016540.1016556`

Stellenbosch University https://scholar.sun.ac.za

[28] Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J. and Hauser, C.H.: Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, pp. 172–182, December 1995. ISSN 0163-5980.
Available at: `http://doi.acm.org/10.1145/224057.224070`

[29] Gilbert, S. and Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, vol. 33, no. 2, pp. 51–59, June 2002. ISSN 0163-5700.
Available at: `http://doi.acm.org/10.1145/564585.564601`

[30] IEEE: IEEE standard for distributed interactive simulation application protocols. Tech. Rep., IEEE, 2012.

[31] Hamilton, J.A., Nash, D.A. and Pooch, U.W.: *Distributed simulation*, vol. 8. CRC Press, 1997.

[32] Engelbrecht, H.A. and Schiele, G.: Transforming Minecraft into a research platform. In: *2014 IEEE 11th Consumer Communications and Networking Conference, CCNC 2014*, pp. 257–262. 2014. ISBN 9781479923557.

[33] Smed, J., Kaukoranta, T. and Hakonen, H.: Aspects of networking in multiplayer computer games. 2001.

[34] Castro, M., Druschel, P., Kermarrec, A.-M. and Rowstron, A.I.: Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, vol. 20, no. 8, pp. 1489–1499, 2002.

[35] Cordasco, G., De Chiara, R., Erra, U. and Scarano, V.: Some considerations on the design of a p2p infrastructure for massive simulations. pp. 1–7. 10 2009.

[36] Shaikh, A., Sahu, S., Rosu, M.., Shea, M. and Saha, D.: On demand platform for online games. *IBM Systems Journal*, vol. 45, no. 1, pp. 7–19, 2006. ISSN 0018-8670.

[37] Chen, K.-T., Huang, P., Huang, C.-Y. and Lei, C.-L.: Game traffic analysis: an mmorpg perspective. pp. 19–24. 06 2005.

[38] Carlini, E., Coppola, M. and Ricci, L.: Integration of p2p and clouds to support massively multiuser virtual environments. In: *2010 9th Annual Workshop on Network and Systems Support for Games*, pp. 1–6. Nov 2010. ISSN 2156-8138.

[39] Buyukkaya, E., Abdallah, M. and Simon, G.: A survey of peer-to-peer overlay approaches for networked virtual environments. pp. 276–300, 2015.

[40] Wang, C. and Li, B.: Peer-to-peer overlay networks: A survey. 09 2004.

[41] Henninger, A., Gonzalez, A. and Reece, D.: *Predicting Agent Spatial Information: A Comparison Between Neural Networks and Dead Reckoning Algorithms*, pp. 459–464. 04 2019. ISBN 9781315782379.

[42] Mauve, M., Vogel, J. and Hilt, V.: Local-lag and timewarp: Providing consistency for replicated continuous applications. *Multimedia, IEEE Transactions on*, vol. 6, pp. 47 – 57, 03 2004.

[43] Loral Systems Company: *Distributed Interactive Simulation Architecture Description Document Volume II: Supporting Rationale Book II: DIS Architecture Issues*. 2nd edn. ADST Program Office, Florida, 1992. ISBN ADST/WDL/TR-92-003010.

Stellenbosch University https://scholar.sun.ac.za

[44] Benford, S. and Fahlén, L.: A spatial model of interaction in large virtual environments. In: *Proceedings of the Third European Conference on Computer-Supported Cooperative Work 13–17 September 1993, Milan, Italy ECSCW'93*, pp. 109–124. Springer, 1993.

[45] Iimura, T., Hazeyama, H. and Kadobayashi, Y.: Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. pp. 116–120. 01 2004.

[46] Yu, A. and Vuong, S.: Mopar: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. pp. 99–104. 01 2005.

[47] Macedomia, M.R., Zyda, M.J., Pratt, D.R., Brutzman, D.P. and Barham, P.T.: Exploiting reality with multicast groups: A network architecture for large-scale virtual environments. In: *Proceedings of the Virtual Reality Annual International Symposium (VRAIS'95)*, VRAIS '95, pp. 2–. IEEE Computer Society, Washington, DC, USA, 1995. ISBN 0-8186-7084-3.
Available at: `http://dl.acm.org/citation.cfm?id=527216.835997`

[48] Jaramillo, J., Escobar, L. and Trefftz, H.: Area of interest management by grid-based discrete aura approximations for distributed virtual environments. 01 2003.

[49] Boulanger, J.-S., Kienzle, J. and Verbrugge, C.: Comparing interest management algorithms for massively multiplayer games. In: *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06. ACM, New York, NY, USA, 2006. ISBN 1-59593-589-4.
Available at: `http://doi.acm.org/10.1145/1230040.1230069`

[50] Buyukkaya, E. and Abdallah, M.: Efficient triangulation for p2p networked virtual environments. *Multimedia Tools and Applications*, vol. 45, pp. 291–312, 10 2008.

[51] Hu, S.-Y., Chang, S.-C. and Jiang, J.-R.: Voronoi state management for peer-to-peer massively multiplayer online games. pp. 1134 – 1138. 02 2008.

[52] Online, E.: EVE-online Status Monitor. 2019.
Available at: `https://eve-offline.net/?server=tranquility`

[53] Stubbings, D.: EVE Online gamers set new record for taking part in a huge video game battle. 2018.
Available at: `https://www.guinnessworldrecords.com/news/2018/4/eve-online-gamers-set-new-record-for-taking-part-in-a-huge-video-game-battle-522213`

[54] Hu, S.-Y.: Spatial publish subscribe.

[55] Gautier, L. and Diot, C.: 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)*, pp. 233–236. July 1998.

[56] Abrams, H., Watsen, K. and Zyda, M.: Three tiered interest management for large-scale virtual environments. 1998.
Available at: `https://calhoun.nps.edu/handle/10945/41588`

[57] Schmieg, A., Stieler, M., Jeckel, S., Kabus, P., Kemme, B. and Buchmann, A.: psense - maintaining a dynamic localized peer-to-peer structure for position based multicast in games. In: *2008 Eighth International Conference on Peer-to-Peer Computing*, pp. 247–256. Sep 2008. ISSN 2161-3567.

[58] Keller, J. and Simon, G.: Solipsis: A massively multi-participant virtual world. pp. 262–268. 01 2003.

Stellenbosch University https://scholar.sun.ac.za

[59] Buyukkaya, E. and Abdallah, M.: Efficient triangulation for p2p networked virtual environments. *Multimedia Tools and Applications*, vol. 45, pp. 291–312, 10 2008.

[60] Ghaffari, M., Hariri, B. and Shirmohammadi, S.: A delaunay triangulation architecture supporting churn and user mobility in mmves. pp. 61–66. 06 2009.

[61] Guibas, L. and Stolfi, J.: Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. vol. 4, pp. 221–234. 01 1983.

[62] Franklin, W.R.: PNPOLY - Point Inclusion in Polygon Test. 2018.
Available at: `https://wrf.ecse.rpi.edu//Research/Short{_}Notes/pnpoly.html`

[63] Protocol FAQ. 2019.
Available at: `https://wiki.vg/Protocol{_}FAQ`

[64] Server/Requirements/Dedicated. 2019.
Available at: `https://minecraft.gamepedia.com/Server/Requirements/Dedicated`

[65] Persson, S.: Celebrating 10 Years of Minecraft. 2019.
Available at: `https://news.xbox.com/en-us/2019/05/17/minecraft-ten-years/`

[66] Diaconu, R., Keller, J. and Valero, M.: Manycraft: Scaling minecraft to millions. In: *2013 12th Annual Workshop on Network and Systems Support for Games (NetGames)*, pp. 1–6. Dec 2013. ISSN 2156-8138.

[67] Rossouw, J.K.: Development of a Minecraft Multiple Client Emulator. Tech. Rep. November, University of Stellenbosch, 2015.

[68] SteveIce10: McProtocolLib. 2019.
Available at: `https://github.com/Steveice10/MCProtocolLib`