Samuel J. Sully

VoxelPopuli: A Decentralised Peer-to-Peer Voxel-Based World

Computer Science Tripos

Robinson College

2019-20

Proforma

Name:	**Samuel John Sully**
College:	**Robinson College**
Project Title:	**Voxel Populi: A Decentralised Peer-to-Peer Voxel-Based World**
Examination:	**Computer Science Tripos – Part II, July 2020**
Word Count:	[1]
Project Originator:	**Samuel John Sully**
Supervisor:	**Prof. Jon Crowcroft**
Director of Studies:	**Prof. Alan Mycroft**
Overseers:	**Prof. Marcelo Fiore & Dr. Amanda Prorok**

Original Aims of the Project

My project aimed to create a peer-to-peer 3D world using a distributed hash table (DHT), namely *Kademlia* [1]. I aimed to explore this decentralised, peer-to-peer approach for Massively Multiplayer Online games (MMOs) to see if such an approach is viable. This was motivated by the advantages of the decentralised approach, such as better load balancing and longevity for the game.

Work Completed

I have completed all the work set out in my proposal, the three parts of my project are all functioning correctly. I implemented *Kademlia* with some modifications to better suit the virtual world application; I implemented the game server to run above the DHT and process the computation for a set of chunks of the world and I implemented the graphical client in *Unity* which connects to the world and allows a user to move around and interact with it. I also completed the test client which was used in the evaluation stage.

[1] This word count was computed by `command?`

Special Difficulties

None.

Declaration

I, Samuel John Sully of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Samuel John Sully of Robinson College, am content for my dissertation to be made available to the students and staff of the University.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Project Summary

My project explores a peer-to-peer architecture for MMOs or large scale simulations. This is in contrast to the more commonly used centralised approach. My project is build upon a distributed hash table which is used to locate in the peer-to-peer network the server responsible for handling any particular part of the world.

My project consists of three parts: the distributed hash table which is a modified version of the *Kademlia* [1] specification; the game server which runs the computation for certain segments of the world and the *Unity* client used to interact with the world. All these have been completed in adherence to the success criteria in my project proposal, as well as the evaluation client used in the evaluation stage. The project culminated in a large scale test on my dedicated server.

1.2 Motivation

The Massively Multiplayer Online Game (MMO) genre is very popular[1] in modern gaming, as an increasing proportion of the populace have access to high speed broadband the prevalence of these games continues to increase. Most of these games employ a centralised client-server mode where the creators of the MMO have a relatively small number of expensive and powerful machines which they use to handle all players.

This centralised approach often requires some form of 'sharding' [2], whereby players are separated into separate, independent instances ('shards') of the same world. Meaning that players can only interact with others connected to the same

[1] *World of Warcraft* – a popular MMO – had 7.7 million subscribers in 2019.

shard. The centralised approach also means that the game creators have total authoritative control over the game.

An alternative approach is a decentralised, peer-to-peer approach which I explore in this project. In this approach the world is separated into segments (or 'chunks') and each peer in the network is responsible for handling the load for a number of chunks. This approach implicitly performs load balancing and is highly failure tolerant, as a node failure can be dealt with by simply having another take over.

This has a number of advantages over the centralised, sharded approach. One significant advantage is that the world is able to be explicitly mutable (such as the voxel-based world I have implemented), with the sharded approach if a player makes a change in one shard then we may need some way of propagating these changes to the other shards while maintaining consistency. However, in my approach there is only one server which is authoritative for the state of any part of the world so there is no need for complex consensus mechanisms.

A further advantage is that the system has improved longevity. When large-scale MMOs cease to be profitable or useful for the developers, who operate the centralised servers, they often shut them down, as recently happened with the popular MMO *Club Penguin* [3] in 2017. With my approach, if we allow individuals to create their own servers to join the peer-to-peer network then, provided there exists a community dedicated to keeping the MMO running, it can continue to exist at no cost to the developers. It would even be possible to have multiple, separate networks running or even networks running modified versions of the game.

1.3 Related Works

There are very few large-scale, peer-to-peer MMOs, likely due to the security issues I will present in the evaluation chapter and due to the fact that it limits the ability for the developers to monetize the MMO post-release. However, it is possible that techniques similar to mine may be used behind the scenes on a number of large-scale MMOs.

One similar piece of work is *SpatialOS* [4], this is a platform for managing online games or simulations in the cloud. It works in a similar way to my project, by splitting up the world into segments which are administrated by separate servers. *SpatialOS* is produced by the startup Improbable and is still fairly new, however, it is being used in the development of a number of games.

It's worth noting also that while my implementation of *Kademlia* is custom, I used a *Kademlia* library [5] for *Python* as a reference for a fully functioning Kademlia implementation. However, this implementation uses the approach outlined in the second Kademlia paper, while my approach uses the slightly different approach from the first paper.

Chapter 2

Preparation

2.1 Starting Point

Prior to this project I had limited experience in implementing distributed systems, my knowledge on such systems mainly comes from the Part IB courses Concurrent and Distributed Systems and Computer Networking. Computer Networking introduced the concept of distributed hash tables (DHTs) which are used extensively in my project. Concurrent and Distributed Systems introduces most of the overarching principles of distributed systems, such as RPCs, which are essential in my project. Furthermore, my project relies on knowledge from a number of other courses, such as Part II Principles of Communication and Part IA Introduction to Graphics. I have some limited experience with 3D graphics from my own hobby programming as well.

2.2 Requirement Analysis

My project aims to implement a suite of software for the operation, interaction with and testing of a 3D world which is distributed over a number of peers in a peer-to-peer network. The success criteria set out in my proposal is as follows:

1. My DHT must adhere to the Kademlia specification. It is possible I will need to make some changes to fit the specification better to my needs and this is acceptable.
2. The peer-to-peer node program must join the network, bootstrapping via some known node, and then will be able to participate in hosting the game world as it becomes part of the DHT.
3. It must be possible to interact with the world using a simple 3D graphical client, which is able to place and remove voxels from the world. These changes must persist.

4. The system must handle player moving between separate chunks (and thus, separate peers) seamlessly, with no loading screen.
5. There must be a simple test agent which connects to and interacts with the world in some notional way to emulate the behaviour of a human user. This is for the purposes of quantitative evaluation.

In addition to these criterion, the project will need to fulfil a number of other requirements:

- **Robustness:** the system must be very robust, handling node failures with minimal disruption to the overall system, minimising disruption to users connected to the system at a given time.
- **Deployment:** the implementation must run as a cloud application, being easily deployable to a large number of machines. In my testing I will be using my dedicated server running *Ubuntu 18.04*.
- **Decentralisation:** the implementation must be designed to be entirely decentralised, nodes in the P2P network must be entirely equal, there must be no authoritative entity in the system.
- **Mutability:** the game world must emulate that of voxel-based games such as *Minecraft*. As such, users must be able to edit the world and have these changes persist, users' locations must also be stored so that when they log out and back in at another time (or to a different server), they return to where they left off.

2.3 Kademlia

My project is built using a DHT at its core, a DHT is a decentralised storage system based on the commonly used hash table data structure. DHTs store <key,value> pairs, these are distributed among the nodes in the network, with there existing some method to partition the set of keys between the nodes, preferably in such a way that node joins or leaves require minimal changes to this partition (i.e. a node leaving does not cause the entire key-node mapping to change). The DHT maintains an *overlay network* where each node maintains a set of links to other nodes in the DHT according to the topology of the network, this set of links is used in routing queries around the DHT.

2.3.1 XOR Metric

The *Kademlia* specification sets out that identifiers be 160bit integers. Nodes IDs and keys for the DHT occupy this ID space. The notion of distance between identifiers, $d(x, y)$, is given by the bitwise XOR of the two (i.e. $d(x, y) = x \oplus y$). This is a valid metric as it obeys the following properties:

1. $d(x, x) = 0$, that is, the distance from any identifier to itself is 0.
2. $d(x, y) > 0$ if $x \neq y$, that is, the distance between any two distinct identifiers is larger than 0.
3. $d(x, y) = d(y, x)$, that is, distances are symmetric.
4. Distances obey the triangle inequality, i.e. $d(x, z) \leq d(x, y) + d(y, z)$.

The set of keys which a node 'owns' is given by all those which are closest to its ID using the above notion of distance[1].

2.3.2 Node State

Each node maintains some amount of information about other nodes in the network in order to route messages. Each node maintains a k-bucket for each i in $0 \leq i < 160$, a k-bucket is simply a sorted list (of length k) of <IP address, UDP port, node ID> triples of nodes between 2^i and 2^{i+1} distance away from this node. The lists are sorted by time last seen, such that the most recently seen node is at the tail of the list. This is useful later when evicting stale nodes from the k-bucket. Note that k is a parameter of the network, the replication parameter.

In order to populate these k-buckets, whenever a node receivers a message from another, it looks for the appropriate k-bucket and, if the sender is already in the k-bucket then it is moved to the tail of the list, otherwise it is appended to the tail of the list. If the k-bucket is full then we send a `PING` RPC to the least recently seen node, if it fails to reply then we evict it and put the new node in instead, else we discard the new node[2].

2.3.3 RPCs

The Kademlia protocol has four RPCs: `PING`, `FIND_NODE`, `FIND_VALUE` and `STORE`. All other operations are built up from these four RPCs. Table 2.1 details the function of each RPC. My implementation will deviate from this specification as detailed in § 3.1.2.

2.3.4 Node Lookup

The lookup procedure is used to locate the k closest nodes to a supplied identifier. The lookup procedure has one parameter, the concurrency factor α. It proceeds as follows:

[1] This is not strictly true, actually the k closest nodes all store values for that key, where k is a parameter of the network.

[2] In my implementation, the new node is added to a queue to join the k-bucket.

`PING`	Used to check whether a node is online, upon receiving a `PING` RPC a node will reply with its ID.
`FIND_NODE`	Takes a 160bit integer as argument (and identifier). When a node receives a `FIND_NODE` RPC it returns <IP address, UDP port, node ID> triples from the $k$nearest nodes to the argument identifier that it knows of.
`FIND_VALUE`	Behaves in the same was as `FIND_NODE` but will return a value if it possesses one for the supplied ID.
`STORE`	Takes a <key, value> pair which the receiving node stores.

Table 2.1: The four *Kademlia* RPCs.

1. Find α closest nodes from own k-buckets.
2. Send `FIND_NODE` RPCs to these α nodes searching for supplied identifier.
3. Then we recursively send `FIND_NODE` requests nodes it learned of from the results of previous steps.
4. When an iteration of RPCs gives us no new nodes better than the current closest, we send RPCs to all of the k closest nodes we have not yet queried.
5. The procedure terminates when we have received a response from all of the k nearest nodes.

The k nearest nodes are returned from this procedure.

2.3.5 Value Lookup

The procedure for retrieving a value from the DHT is similar to the node lookup procedure above, replacing the `FIND_NODE` RPCs in the above description with `FIND_VALUE` RPCs. Instead of returning the k nearest nodes it will return the value found, or some notional `NULL` value if none exists.

2.3.6 Value Storage

The store value procedure consists of performing a lookup node procedure as above with the identifier being the key of the <key, value> pair to be stored. Then `STORE` RPCs with the <key, value> pair are sent to the k nodes returned from the lookup.

2.3.7 Bootstrap

Bootstrapping is the process by which a node joins the network. Because *Kademlia* routing information is implicitly learned through network activity

we do not need an explicit `JOIN` method, we can simply use existing RPCs to join a network. All that we need is the IP, port and ID of any existing node in the network, this is the bootstrap node.

The joining node, n, inserts the bootstrap node, m, into the appropriate k-bucket and then performs a node lookup for its own ID. Finally it refreshes all its buckets which are further away than its closest neighbour. Refreshing a k-bucket simple means picking a random ID from that bucket's range and performing a node lookup for that ID. This operation is performed automatically by each node periodically on all buckets which have not been touched in a certain amount of time[3]. By performing a lookup of itself and by refreshing those k-buckets we have ensured that this node has been inserted into the routing tables of a number of other nodes.

2.4 Game Server

The second major part of my project is the game server, for this I will use an architecture similar to that used by *Minecraft* and by *Valve*'s *Source* engine [6]. An instance of a game server will be the authoritative dedicated host that runs the computations for a given set of chunks of the game world, a client will connect to a number of servers in order to receive the current world state and display it to the user graphically. This section of the system is purely client server, clients do not communicate among one another, instead doing so via the server(s).

The server will use an approach used in both *Minecraft* and *Source* where the game world is simulated in discrete time steps known as 'ticks'. During a tick we process any incoming packets and update the state of the world, then we send any packets to clients in order to update the world state. In these examples world state is transferred to clients using *delta compression*, where, after the initial sending of the game state, we only send changes that happened since the last tick, this reduces network load.

A number of further approaches could be employed by my implementation, such as compensating for latency and interpolating between ticks. However, these are beyond the scope of my investigation and are thus not a requirement for my project.

2.5 Client

The third major part of my project is the client, which will be used to connect to and interact with, the world. This section of the project will require some 3D graphics, thus it will draw on material from the two graphics courses in Part

[3] Usually 1 hour.

IA and Part IB. I will also need to implement the algorithm for locating and loading the relevant chunks into the world so that the chunks surrounding the player's current location are always loaded.

For this section of the project I decided to use *Unity*, rather than *LWJGL*, because the graphical element was simpler and as graphics is not the focus of my project this felt appropriate.

2.5.1 *Unity*

Unity is a 3D game engine which is widely used. It is fairly easy to learn yet quite powerful and expressive making it a popular tool in the modern games industry. A simple overview is all that is needed for this project. *Unity* provides us with a 3D world populated with game objects. These game objects can have *components* attached to them, such as a mesh renderer (to render a 3D mesh) or light source for example. However, most importantly you are able to assign scripts as components which you can write yourself, these scripts have a number of built-in methods which *Unity* calls at particular times, most importantly we have the `Start()` method, which is called upon creating the component, the `Update()` method which is called each frame and the `FixedUpdate()` method which is called each time the physics engine updates. These scripts have access to a powerful API allowing us to influence the game world.

2.6 World & Terrain

The game world will be analagous to that of *Minecraft*, in that it will consist of voxels (i.e. blocks) arranged in a 3D grid. An example of *Minecraft*'s terrain can be seen in figure 2.1. The *Minecraft* world is broken into 'chunks' each $16 \times 16 \times 256$ blocks, then each chunk is simply a 3D array of block data.

The terrain in *Minecraft* is generated procedurally, allowing for infinite worlds to be created on the fly. A common approach in procedurally generated video games is to use some form of coherent noise[4] to generate a height map[5]. I plan to use Perlin noise [7] (or its successor, Simplex noise [8]) to generate a heightmap for my world. Then I will use simple rules to assign blocks at different heights different values (i.e. grass on top, followed by dirt, followed by stone) in order to procude a *Minecraft*-like world. The structure of the world

[4]Coherent noise simply means smooth pseudorandom noise which obeys the following properties:

1. The same input always gives the same output.
2. A small change in the input will produce a large change in the output.
3. A large change in the input will produce a random change in the output.

[5]Simply a 2D function or array where the value at any given point is the height of the terrain at any given point.

Figure 2.1: A screenshot of terrain from the game *Minecraft*.

into chunks allows for easy segmentation across servers as each chunk can reside on a different server, additionally, by having data represented within a chunk as a 3D array this makes editing the world simple.

2.7 Professional Practice

2.7.1 Ethical Implications

One ethical and legal concern is that my project would give users access to a canvas within which they could, theoretically, encode any data. This could give rise to legal issues if, for example, illegal material were encoded in world data, then the server owner on who's server that data is stored could technically be in breach of the law.

2.7.2 Methodology

The project was broken up into discrete features, with a timeline planning to complete each in approximately $2-3$ weeks. Thus I followed the *Agile* software development workflow. Each $2-3$ week sprint had a deliverable which could be tested independently and demonstrated. My sprint timetable outlined in the proposal was adapted as the projected moved forward and some parts of the project took more, or less, time than anticipated.

2.7.3 Tooling

I used the *PyCharm* IDE for the development of the *Kademlia* implementation and my game server as these were both written in Python using version 3.8 due to improvements made to the *asyncio* library in Python 3.8. For the client I used *Unity* with *Microsoft Visual Studio 2017* for editing the *C#* scripts. *Git* was used for version control, with code pushed to *GitHub* regularly and further backed-up daily to both the SRCF[6] and the MCS using a *cron* job.

2.7.4 Documentation

Pending...

[6]Student-Run Computing Facility.

Chapter 3

Implementation

My project consists of three parts: the bespoke *Kademlia* implementation, the game server and the client. The system works by having the client query the *Kademlia* implementation to locate the appropriate servers for a particular part of the world, then connecting to that server and 'joining' the world via that server. This is visualised in figure 3.1.

The client needs to connect to a single node from the *VoxelPopuli* network, which it will use as a 'stepping stone' to access the whole network via. It uses this stepping stone node to query the DHT to find the <IP, port> of the game servers responsible for the chunks it needs. It then initiates connections with each of these game servers in parallel, registers the player in that chunk and downloads the world data.

The *VoxelPopuli* server consists of two distinct parts, the *Kademlia* node and the game server. Thus for each node in the network there are two *virtual* nodes. The *Kademlia* nodes are not visible to a client. In order to query the DHT, clients connect to any game server and initiate a special type of session specifically for DHT access. The game server then performs queries to the DHT on behalf of the client. Further details of this special session are in § 3.2.4. Note that for simplicity's sake the *Kademlia* node and game server bind to the same IP address with the game server port being the *Kademlia* port incremented by 1.

3.1 Kademlia

This section introduces my bespoke implementation of the *Kademlia* specification. A custom implementation of *Kademlia* was necessary for two main reasons:

- The project necessitated not only `PUT` and `GET` procedures but also a

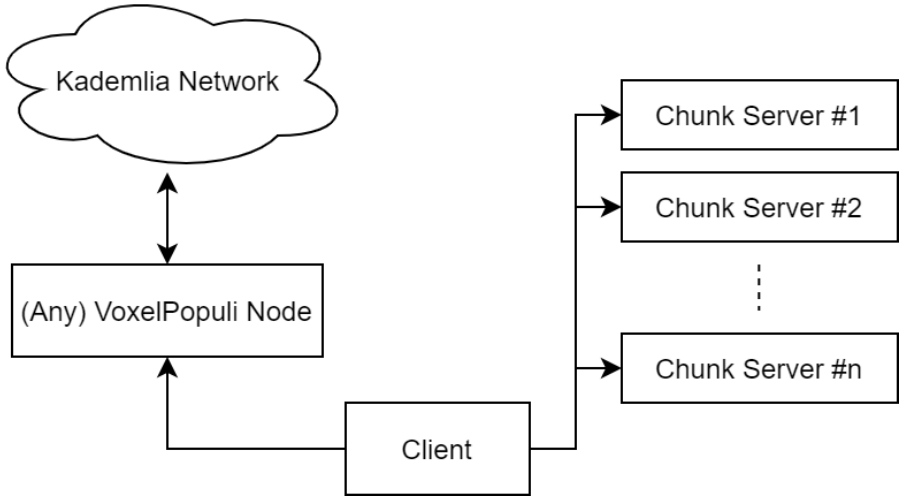


Figure 3.1: Diagram giving an overview of *VoxelPopuli* architecture.

`GENERATE` procedure for locating the appropriate server for a particular chunk and then instructing it to generate said chunk.

- Two distinct types of data needed to be stored in the network. The network needed to store chunk location information (IP and port of appropriate server) and player state information in distinct areas so as not to conflate them.

As such, a custom implementation was devised with additional RPCs and a bespoke high-level interface with the required `PUT`, `GET` and `GENERATE` procedures.

3.1.1 RPC Framework

In order for *Kademlia* nodes to communicate with each other we need a method of issuing RPCs to remote machines and retrieving the results. In order to do this I used *asyncio*'s `DatagramProtocol` class which is a base class for implementing protocols over UDP. This class provides overrideable methods such as `datagram_received()` which is called when the underlying socket receives a UDP datagram. I wrote a general-purpose RPC system because at the time I was unsure how many RPCs I would need and wanted to be able to add and remove them on the fly. This proved useful when revising my *Kademlia* implementation to include the separate `STORE` and `FIND_VALUE` RPCs.

In order to implement the RPC framework I designed a *JSON* format for RPC calls and responses (see table 3.1 for details). I created two function decorators in Python: `@rpc` and `@stub`. `@rpc` adds no special behaviour and simply marks that this method may be called remotely; `@stub` replaces the supplied method with a method which takes the same arguments and computes

`id`	32-bit random number to uniquely identify this RPC call.
`node`	ID of the sending node.
`call`	Boolean representing whether this is an RPC call or a response.
`rpc`	The name of the remote procedure to be called.
`args`	The list of arguments (in order) to be supplied to the remote procedure. Not present on responses.
`ret`	The return value of a remote call. Not present on calls.

Table 3.1: JSON RPC specification.

the JSON RPC string to be sent to the other machine, then sends this message and awaits a reply before returning the result (or `None` on a timeout), this process is outlined in algorithm 1.

Upon receiving a datagram the RPC framework decodes it as a JSON string and checks whether it is a function call or a response. In the case that it is a call, the relevant function is checked to determine if it has the `@rpc` decorator (i.e. it has been marked for remote calling) and then executes it, packaging the result up as a JSON message and returning it to the sender. If it is a response, it checks if there is a pending RPC with that ID, if so it will supply the result to that RPC call, otherwise the message is discarded. This process is detailed in algorithm 2.

Algorithm 1 RPC framework `@stub` decorator algorithm.

```
function RPC_STUB(func, to_node, args)
    id ← randombits(32)
    json ← {"id": id, "node": this_node_id, "call": true, "rpc": func, "args": args}
    store pending rpc in table
    send_udp(json, to)
    schedule timeout
    await response OR timeout
    if timed out then
        return None
    else
        response ← get_response(id)
        return response
    end if
end function
```

This framework allows for the *Kademlia* specification to be implemented as described in the following sections. It allows methods to be tagged as remotely callable and allows for the creation of *stub* methods which allow calling of RPCs on remote machines. This is necessary for the implementation of the *Kademlia* RPCs.

Algorithm 2 Datagram handling in my JSON RPC framework.

```
function DATAGRAM(data, from)
    msg ← json_decode(data)
    if msg.get("call") then
        func ← get_function(msg.get("RPC"))
        if func exists AND func has decorator @rpc then
            res ← func(msg.get("args"))
            json ← {"id": msg.get("id"), "node": this_node_id, "call": false,
"rpc": func.name, "ret": res}
            send_udp(json, from)
        end if
    else
        if rpc with id msg.get("id") is pending then
            send msg.get("ret") to pending RPC calls
        end if
    end if
end function
```

3.1.2 Custom RPC Specification

In table 2.1 I outlined the RPCs in the default *Kademlia* specification. In my implementation I have separate `STORE_PLAYER` and `STORE_CHUNK` RPCs as well as equivalent variants of the `FIND_VALUE` RPCs. These replace the default `STORE` and `FIND_VALUE` RPCs meaning we have a final specification consisting of 6 RPCs: `PING`, `FIND_NODE`, `STORE_PLAYER`, `STORE_CHUNK`, `FIND_PLAYER` and `FIND_CHUNK`.

In order to accommodate these new RPCs, the lookup procedure was made polymorphic, taking the appropriate RPC as an argument. Additionally, each node now has two separate storage tables, one for player data and one for chunk data. This implementation was preferable to the alternative of running two distinct *Kademlia* networks, as this would have significant additional overhead as two separate node states would need to be maintained for each node.

3.1.3 Generate Procedure

When a chunk is found not to exist in the network, it must be generated. In order to do this we need to first locate the server it should be generated on, then we need to check that server is up. Once we have confirmation that the server is running we send a request to the game server on that VoxelPopuli server to generate and initialise the node ready for players. If a node is not up we move to the next nearest node. Once the chunk is generated we must store the <IP address, port> of the node it was generated on in the network so that in future when we look up this chunk we will find the correct server. Algorithm 3 gives pseudocode of the generate procedure.

Algorithm 3 Generate Procedure Pseudocode.

```
function GENERATE(ChunkCoordinate)
    key ← sha1(ChunkCoordinates)
    nodes ← lookup(key)
    for n ∈ nodes do
        Send generate request to game server at < n.ip, n.port + 1 > ▷ Game server address is that of respective Kademlia node with port incremented.
        if Successful then
            Call RPC STORE_CHUNK(key, < n.ip, n.port + 1 >)
            return Success
        end if
    end for
    return Failure
end function
```

Block Type	Air	Stone	Grass	Dirt
Integer Value	0	1	2	3

Table 3.2: *VoxelPopuli* block types.

3.2 Game Server

The game server is responsible for performing the computation and maintaining the state for a number of chunks concurrently. It is also responsible for providing clients with a method of querying the DHT to locate chunks in the *VoxelPopuli* network and to retrieve player data.

3.2.1 Server State

The game server state consists of a set of active client connections (and two queues for each connection, one for receiving and one for sending data) and the state for each of the chunks it is responsible for. Chunk state consists of the world data for that chunk, represented as a 3D array[1], of integers ranging between $0 - 3$ to represent different voxel types (see table 3.2); the set of players currently active in this chunk and their locations; a list of clients who are subscribed to updates on this chunk and (x, y) – the coordinates of this chunk in the world. Note here that a client may be connected to a single game server multiple times because it is connected to multiple of the chunks this server is responsible for.

Chunks are either loaded or unloaded, unloaded chunks have no connected clients and their computation (i.e. game loop) is not currently being executed. A loaded chunk has a dedicated thread for performing the game computations

[1]The current size of a chunk is $32 \times 32 \times 32$ blocks.

of the chunk, a chunk is only loaded if it has a non-zero number of connected clients and is unloaded as soon as the last client disconnects.

The game server's main thread runs continuously a loop it checks if there are any new connections, in which case it will perform the handshake procedure in § 3.2.2.1. Furthermore, it uses `select` to get a list of sockets ready for reading or writing. Then, for each readable socket it reads all data available and adds it to the socket's buffer, if it encounters a newline character (denoting the end of a packet) it puts the packet into the receive queue associated with this socket and clears the buffer. For each writeable socket it checks if there is queued data to send and if so, attempts to send it, removing the successfully sent data from the send queue associated with that socket.

3.2.2 Protocol

3.2.2.1 Handshake

When the server receives a new connection it waits for a JSON message from the new client, this message is decoded and it should contain a *type* field. This *type* field can take four different values, depending on which the server will take different actions. The possible values are:

- **"connect"** – this means that the client wishes to connect to a chunk resident on this *VoxelPopuli* server. There will be a field called *chunk* which contains the chunk coordinates of the chunk to be connected to. The server will then perform the following:
  1. Check if the relevant chunk is indeed resident and generated on this server. If not it will send a failure message to the client and return.
  2. Check whether the chunk is loaded and if not load it by starting a new chunk processing thread for this chunk (see § 3.2.3).
  3. Add this client to the chunk processing thread's client list.
- **"generate"** – this means that the client (in fact another *VoxelPopuli* server in this case) is requesting that a supplied chunk be generated. Again the *chunk* field will be present, containing the coordinates of the chunk to be generated. The server will simply generate the chunk and add it to its set of chunks, however, it will not load it.
- **"dht"** – this means that the client does not wish to use this node as a game server but instead as an access point to the DHT. In this case we launch a new, dedicated, thread to handle DHT queries, details of this thread can be found in § 3.2.4.
- **"ping"** – used to query whether a game server is alive, returns the `UTF-8` encoding of the string *"pong"*.

Type (`int`)	Arguments (`list of floats`)	Player (`string`)

Table 3.3: Message format for the game server protocol.

Once the action has been completed successfully the server sends an acknowledgement to the client, informing it that the operation succeeded. Unless the connection is to be kept live (i.e. in the case of connecting to the game server ("connect" packet) or DHT interfacing ("dht" packet)) then the connection is closed.

3.2.2.2 Game

There are a number of messages which need to be exchanged between the client and game server. These allow the client(s) to inform the server that it has performed an action updating the world's state and allow the server to inform the client(s) of these, and other, state changes. Again these are exchanged as JSON messages of the format shown in table 3.3.

The game server protocol consists of 6 types of packet which are exchanged between clients and servers. The details of each packet are in table 3.4. In order to signal the end of a packet each packet has a newline character appended to the end of its JSON representation. Packets are encoded as bytes using UTF-8 and then sent over a TCP socket between client and server.

3.2.3 Chunk Thread

The chunk thread performs the computation for one chunk of the world. In theory, the network should contain no more than one chunk thread for any given chunk. Meaning that clients connecting to the same chunk will always find the same server.

It is important here to make a distinction between a 'client' and a 'player', a client is any instance of the software described in § 3.3 and has an associated player with a position in the world. This client may be connected to multiple chunk threads across the network, however, the player is only resident in one of these chunks, in all the other chunks the client is simply listening to updates from this server in order to render this chunk to the user.

The computation runs in a continuous loop, it pops any packets from its receive queue and performs the appropriate action as explained in table 3.4, this may involve updating the game state and hence, sending packets to some of the clients connected. It then steps the game time and sends all players in this chunk a packet informing them of the time change. Next it checks if any player has not been heard of for a certain period of time and removes it, assuming it has gone offline.

Packet Type	ID	Purpose	Arguments
PLAYER_REGISTER	1	Used by clients to inform a server that a player is entering a particular chunk. Relayed to clients to inform them they should start rendering this player.	Player location (x, y, z).
PLAYER_DEREGISTER	2	Used by clients to inform a server that a player has left a particular chunk. Also triggers server to save player data to DHT. Relayed by server to clients to inform they should stop rendering this player.	None.
PLAYER_MOVE	3	Used by clients to update inform the server of a change in a player's location. Relayed by server to clients so they can update where the player is being displayed.	New player position (x, y, z, θ), θ = yaw.
CHUNK_DATA	5	Packs entire chunk block array into packet and sends to player so they can load and display a chunk.	Flattened chunk data array.
TIME	6	Informs the client that the in-game time has changed, sent every tick.	Current time as minutes from midnight (accurate to second).
BLOCK_CHANGE	7	Used by a client to inform the server that the world has changed, server updates state and sends updated CHUNK_DATA packet to all clients.	Location of block to change and new block type.

Table 3.4: List of packets types exchanged between clients and game servers.

When there are no active clients for this chunk thread it will unload, saving the state and stopping the thread so as not to waste resources. This ensures that the number or chunk threads active on a given node at a particular time is kept to a minimum.

3.2.4 DHT Interface

As previously explained, in order for clients to query the DHT (and thus retrieve their saved player data and the locations of chunks in the network) they must have some method to query the DHT. In order to do this they connect to a game server and perform the DHT handshake as described in § 3.2.2.1. This connection is then given a dedicated thread to respond to queries to the DHT. The current protocol allows a client to either get the <IP, port> for a particular chunk (the chunk is generated if does not exist in the network) or download player save data (location primarily).

3.2.4.1 Protocol

The thread process one request at a time and is not guaranteed to respond to requests issued while another request is pending. There are two packet types chunk query (type 0) and player query (type 1), a packet's first byte is its packet type. To query a chunk location a packet is sent consisting of a leading 0 byte (packet type) followed by the JSON encoding of the coordinates of the chunk (as a JSON list). The server replies with the JSON list `[IP address, port]`. To query player data a packet consisting of a leading 1 byte (packet type) followed by the JSON attribute-value pair: {`'name':player_name`} where `player_name` is the username of the player to query. The server responds with the JSON representation of the player's coordinates. If the player was not found then it returns $(0, 32, 0)$ (the default spawn location). Note that all the JSON strings are encoded as bytes for sending using UTF-8.

3.3 Client

The client is the user's entry point into the world, it renders the world (and other players – rendered as grey cuboids) and allows the player to move around within it; it provides a basic physics simulation for realistic movement and collisions and it allows the user to make changes to the world by breaking and placing blocks. A screenshot of the client in action can be seen in fig 3.2. The client was written in C# using the *Unity* game engine.

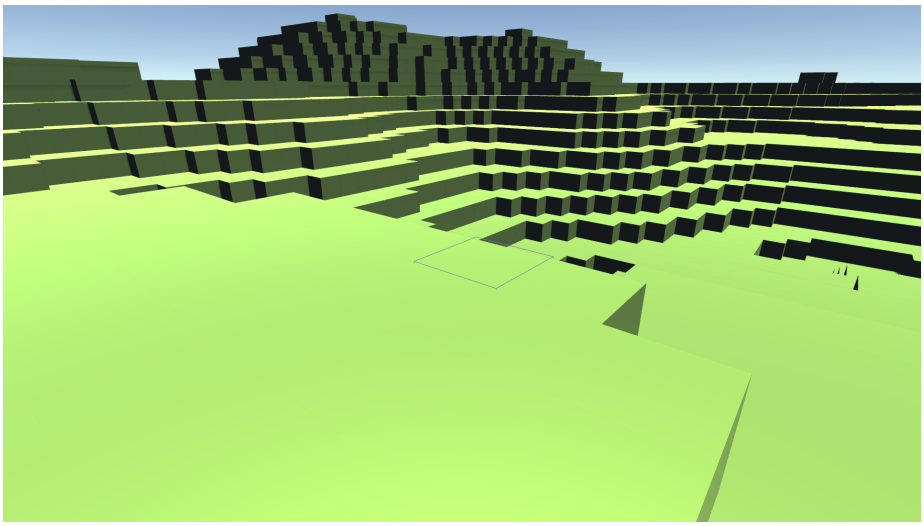


Figure 3.2: A screenshot of the *VoxelPopuli* client connected to a world.

3.3.1 Architecture

The client operates two main threads as well as a two threads (one each for receiving and sending) for each chunk it is connected to. The two main threads exist to separate long, blocking processes from the thread rendering the world, this prevents frame rate stuttering[2]. The two main threads will be referred to as the *network thread* and the *game thread*.

3.3.1.1 Network Thread

The network thread has access to send/receive queues for each chunk the client is connected to. It takes received packets from these queues and processes them ready to be sent to the game thread. Some packets require an amount of computationally intensive pre-processing before they are ready to be used by the game. For example, packets of type `CHUNK_DATA` require a large amount of processing as the array needs to be unflattened and then the chunk's mesh data needs to be generated – this can take 10s of frames. This computation is performed in the network thread which has no bearing on the client's frame rate. Once any pre-processing is performed a state update is sent to the game thread which is rendering the world.

The network thread also receives updates from the game thread, these updates can be player movement or block changes. Where appropriate, it converts these updates into a packet and queues these for sending to the relevant chunk's server.

[2]Where the frame rate drops briefly, causing the game to appear to momentarily freeze

Update Type	Purpose	Arguments
`PLAYER_MOVE`	Used by the network thread to inform the game thread that another player has moved and the object representing that player should be moved.	Player name and new location.
	Used by the game thread to inform the network thread that the player associated with this client has moved so that the network thread can relay this to the server.	
`LOAD_CHUNK`	Used by the network thread to make the game thread load in newly downloaded chunk data.	Chunk coordinates, block array and mesh data. (Encapsulated as a 'Chunk' object.)
`UNLOAD_CHUNK`	Used by the network thread to make the game thread unload a chunk, removing it from the game world.	Chunk coordinates.
`PLAYER_ADD`	Used by the network thread to inform the game thread of a player appearing in a particular chunk, the game thread will add an object to the world to represent this player.	New player's location and name.
`PLAYER_REMOVE`	Used by the network thread to inform the game thread a player has left a particular chunk, the game thread removes the player from the world.	The player's name.
`TIME`	Network thread informs the game thread a `TIME` packet has been received, the game thread will move the game's light object to reflect this.	The new time.
`BLOCK_CHANGE`	Used by the game thread to inform the network thread the player has broken or placed a block.	The position of the block to be updated and its new type.

Table 3.5: Details of the updates exchanged between the network and game threads of the client.

Finally, the network thread ensures that the correct chunks are loaded, locating and loading those which are needed and unloading those which are no longer needed. This is discussed in detail in § 3.3.3. To do this is maintains a connection to a *VoxelPopuli* node where the connection has been setup to be in DHT query mode, allowing the client to use this node to query the DHT.

3.3.1.2 Game Thread

This is the main *Unity* thread. The part that is of interest to this project is the `update()` method in our `World` script which is executed each frame. This has access to the incoming and outgoing update queues used to exchange state updates between the network thread and the game thread. The game thread simply performs the necessary steps to make state updates received from the network thread visible to the user, and it sends updates to the network thread when the user does something which updates the state of the world.

Details of the types of updates exchanged can be seen in table 3.5. Some of these are very similar to the packets from table 3.4 as expected. Within the game thread each player has an object to represent it (in this prototype they are simply a cube) and each chunk has an object to represent it (a 3D mesh with an associated 'Chunk' object containing the block array and the mesh's data – details of mesh data generation are in § 3.3.2).

3.3.2 Chunk Mesh Generation

In computer graphics, a *mesh* is a collection of vertices, edges and faces which make up a 3D object. For each chunk we must generate a mesh from the block array in order to render the chunk to the player. In order to do this we must construct 2 arrays; firstly, we need to compute the array of vertices in the mesh, then we need the array mapping these vertices to triangles. In order to do this, for each triangle we need the indices of the 3 vertices that make up that triangle, we then add these 3 indices to the triangles array.

In order to apply textures to the faces we also need to provide a texture coordinate (often referred to as a *uv* coordinate) to each vertex. This is supplied in an array too, where the indices in the vertex array correspond to the indices of the texture coordinate of that vertex in the uv array. Once we have computed our vertex, triangle and uv arrays we can supply these to unity which will be able to generate and render the mesh for us. This process is very computationally intensive, for our chunks it involves iterating over all $32^3 = 32768$ blocks and performing 6 iterations (1 per face) for each block. Pseudocode for the mesh generation algorithm is supplied in algorithm 4, this approach is based on similar methods found in various open source voxel-based games. Figure 3.3 shows how a face of the cube is produced by two triangles.

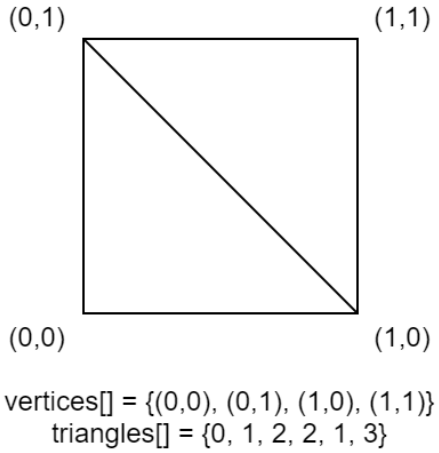


Figure 3.3: A diagram of mapping between vertices and triangles for a simplified face of a cube.

Algorithm 4 The mesh generation algorithm used by the client.

```
blocks[][][] ← chunk block array;
vertices[] ← new vec3 array;
triangles[] ← new int array;
uvs[] ← new vec3 array;
vertex ← 0;
for 0 ≤ x < CHUNK_SIZE do
   for 0 ≤ y < CHUNK_SIZE do
      for 0 ≤ z < CHUNK_SIZE do
         for each face do
            if blocks[x][y][z] is air OR
blocks[x][y][z] + faceNormal is not air then
               continue;
            end if
            add vertices for this face;
            add uv coordinates for correct texture;
            triangles += [vertex+0, vertex+1, vertex+2];
            triangles += [vertex+2, vertex+1, vertex+3];
            vertex ← vertex + 4;
         end for
      end for
   end for
end for
```

3.3.3 Chunk Loading & Unloading

As noted in § 3.3.1.1 the network thread is responsible for ensuring the correct chunks are loaded, it uses the DHT query connection for this purpose. The rule for determining whether a chunk should be loaded is simple: a chunk should be loaded if it is not already loaded and it is within a 3×3 grid of chunks centred on the player. Similarly, a chunk should be unloaded if it is outside of a 9×9 grid of chunks centred on the player.

Each time the network thread processes a `PLAYER_MOVE` update from the game thread it checks to see if there are any chunks to unload, if so it unloads them (i.e. ends the connection and sends a `CHUNK_UNLOAD` update to the game thread). Then it checks if there are any chunks to load, if so it queries their locations using the DHT connection, connects to them and loads them (it only sends the `CHUNK_LOAD` update when it receives the `CHUNK_DATA` packet from the server for this chunk). This process is outlined in algorithm 5.

Algorithm 5 Algorithm for maintaining the correct set of loaded chunks by the client.

```
for chunk ∈ loaded do
    if |chunk.x − player.chunk.x| ≥ 5 OR |chunk.y − player.chunk.y| ≥ 5 then
        unload(chunk);
    end if
end for
chunk_position ← (player.chunk.x, player.chunk.y);
for −1 ≤ i ≤ 1 do
    for −1 ≤ j ≤ 1 do
        if chunk at chunk_position + (i, j) is not loaded then
            s ← query_chunk_from_dht(chunk_position + (i, j));
            load chunk from server s;
        end if
    end for
end for
```

3.4 Overview

This section provides an overview of the project and the files contained within my repository (which are shown in figure 3.4 shows).

3.4.1 Client

The client takes the form of a *Unity* project. As noted, Unity allows you to assign scripts to game objects which then have methods such as `Update()` and

`Start()` which are called at the appropriate times (in these cases: each frame and when the object is created respectively).

The client has 6 files of code, 3 of which contains scripts able to be assigned to objects. `Controller.cs` is a script assigned to the player object which takes user input and moves the player, obeying physics and collisions. `WireFrame.cs` is another script applied to the player object which simply renders a wire frame cube around the block the player is looking at. `World.cs` is a script applied to the (initially) world object which performs the game thread actions described in § 3.3.1.2.

The other 3 files contain classes and data used by the aforementioned scripts. In `Chunk.cs` we have a class for chunks, these contain the chunk's block array and mesh data, and provide a number of methods for querying the contents of the chunk (such as `IsSolid(Vector3 localPos)` for querying if a block is solid at position `localPos` relative to the chunk). `Network.cs` contains the `NetworkThread` class which is used to execute the network thread as explained in § 3.3.1.1. Finally, `Constants.cs` contains a number of constants such as the tick rate, chunk size and various pieces of data needed for mesh generation.

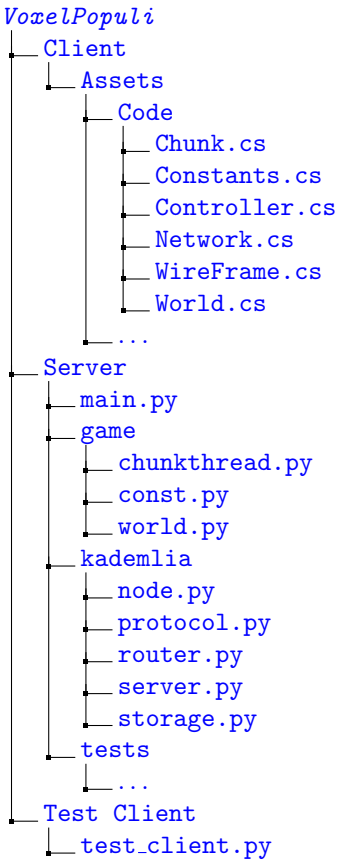


Figure 3.4: Directory overview of *VoxelPopuli*.

3.4.2 Server

The server is split into two main parts, the *Kademlia* implementation and the game server. Within the game server we have `chunkthread.py` which contains the `ChunkThread` class which is used to execute the chunk thread as discussed in § 3.2.3. We also have `world.py` which contains classes for maintaining state about the world, namely `Chunk` and `Player`. We have `const.py` which contains a number of important constants such as a `PacketType` enumeration. Finally we have `main.py` which performs the main loop of a *VoxelPopuli* node, i.e. waiting for new connections and performing the handshake procedure from § 3.2.2.1. The code for the DHT interface thread (see § 3.2.4) is here too.

Within the *Kademlia* folder we have `node.py` and `storage.py` which con-

tain simple classes to represent a *Kademlia* Node and the local storage on a node respectively. `router.py` contains the k-bucket class and routing table class. `protocol.py` contains the *Kademlia* RPC implementation as well as the RPC framework outlined in § 3.1.1. Finally, `server.py` contains the high level interface for the *Kademlia* network, allowing the following operations to be performed: `run()` (initialise a node), `get_chunk()` (lookup a chunk and generate it if it does not exist), `get_player()` and `save_player()` (retrieve or save player save data), `generate_chunk()` and `republish_chunk()`.

3.4.3 Test Client

The test client is a suite of scripst which simply initialises a configurable number of dummy players and connects them to a single server or to a network in random or configurable positions, it can then move them around randomly to simulate activity.

There are three scripts, two of which connect to a single server and one which connects to the network as a whole. First we have `test_sing_chunk.py` which connects to a single server and loads a number of players into a single chunk and then moves them around within the chunk every $\frac{1}{20}$ seconds. We also have `test_many_chunks.py` which connects to a single server and loads a number of players into a configurable number of chunks on that server, the players are evenly distributed between chunks and then moved randomly as above.

Finally we have `test_client.py` which connects to a server to query the DHT. It then spawns a number of dummy players who will connect to the network and move around the world, moving between chunks this time and connecting to the appropriate server by querying the DHT – this client most accurately simulates real players and will be used in my large scale test of the system.

Chapter 4

Evaluation

My project meets the original success criteria and further goals set out in § 2.2. It has a number of limitations which will be explored in § 4.6.

4.1 Methodology

This project aims to explore the viability of using a peer-to-peer network topology for a large online world with potentially thousands of players. Unfortunately I do not have access to enough machines (and indeed players) to accurately field test the project. I shall instead be performing two main types of testing to prove the viability of *VoxelPopuli* as a MMO engine. I will be performing local simulations using simulated players (created by my test client) to demonstrate the load that a single server experiences under different scenarios. Then I will create a small network of around 100 nodes which will be populated by simulated players to prove that the system is able to perform at scale.

I will also be testing the system by subjecting it to failure modes such as node failure, and demonstrating that it continues to operate correctly in these scenarios. I will briefly discuss the of the security of *VoxelPopuli* and its susceptibility to various attacks. Additionally, I will be investigating the adherence of the *Kademlia* implementation to the specification.

Finally some simple tests on the client will be performed and then I will consider the limitations that the peer-to-peer approach presents when compared to the more commonplace client-server model.

4.2 *Kademlia* Implementation

In order to prove that my *Kademlia* implementation conforms with the specification, as required by my success criteria, I performed two types of testing.

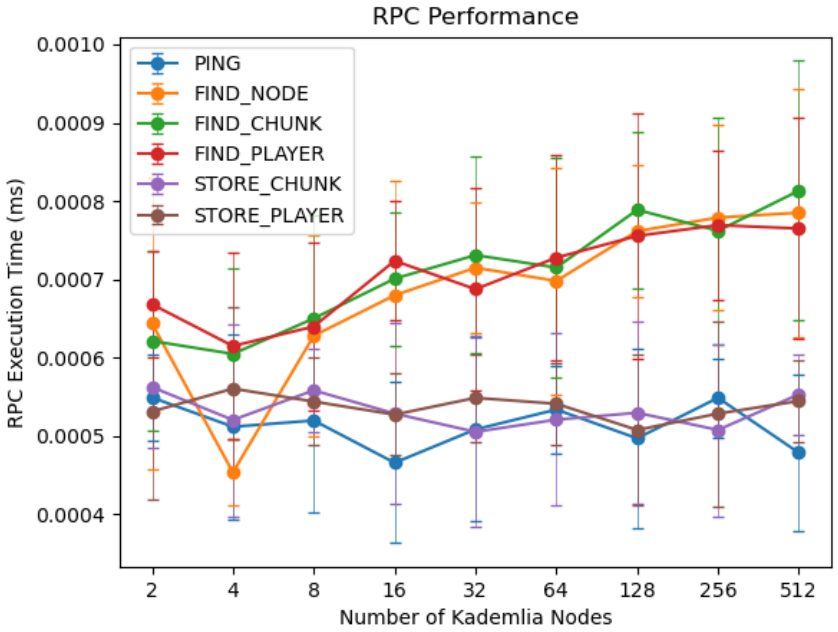


Figure 4.1: Computation time of each *Kademlia* RPC as network size increases. The *Kademlia* specification states these should all be $\mathcal{O}(1)$. Error bars represent $\pm\sigma$ over 100 trials.

4.2.1 Unit Testing

Unit testing was performed on core components of the *Kademlia* specification. These were informed by the unit tests for *OpenDHT* [9] and the Python *Kademlia* library [5]. Components tested with unit tests include the k-bucket class – ensuring that it performs correctly as nodes are added and removed, the routing table – ensuring contacts are processed correctly and that it responds correctly to queries and the storage implementation.

4.2.2 RPC Testing

4.2.2.1 Setup

The *Kademlia* specification sets out requirements for the time complexity of each RPC, I needed to prove that my implementation adheres to these requirements. In order to do this I setup a single dedicated server and instantiated a number of *Kademlia* nodes and bootstrapped them to a network to give me a network of a known size. I then performed the relevant RPC a number of times and recorded the average time and standard deviation. The plot can be seen in figure 4.1.

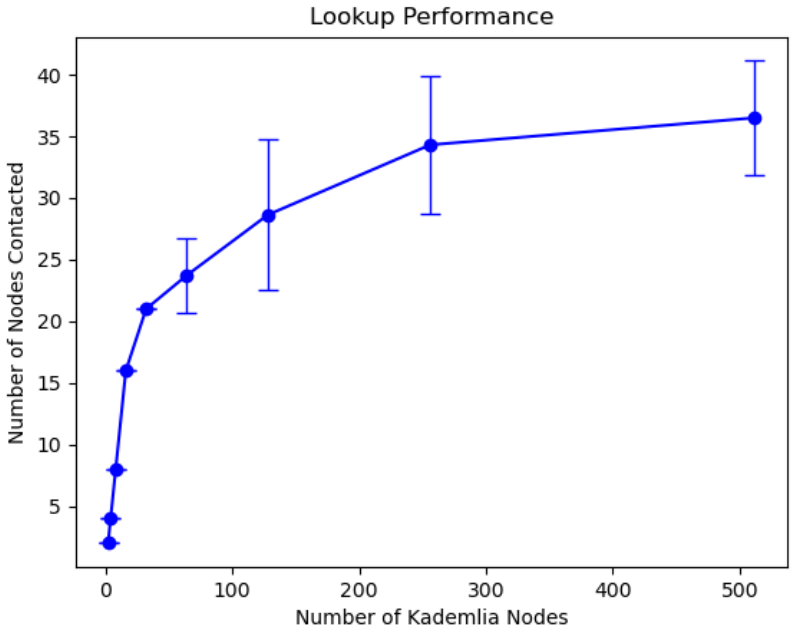


Figure 4.2: Number of nodes contacted by a `lookup` procedure as network size increases. The *Kademlia* specification states that this should grow at a rate of $\mathcal{O}(\log n)$, where n is number of nodes. Error bars represent $\pm\sigma$ over 100 trials.

4.2.2.2 Analysis

As can be seen each of `PING`, `STORE_CHUNK` and `STORE_PLAYER` are clearly $\mathcal{O}(1)$. The plots for `FIND_NODE`, `FIND_CHUNK` and `FIND_PLAYER` exhibit a slightly different shape. These RPCs still operate in $\mathcal{O}(1)$ time, however, because the find operations must sort the nodes it knowns of, this takes $\mathcal{O}(n \log(n))$ time, however, as the number of nodes in the routing table is capped at 160·k (3200 in the common case where k= 20 – used in my implementation) this is still $\mathcal{O}(1)$ as the number of nodes grows large. You can see the graph is beginning to plateau at for higher n.

4.2.3 Lookup Procedure

The *Kademlia* specification clearly states that the lookup procedure must contact $\mathcal{O}(\log(n))$ nodes. I performed a test where I recorded each node that was contacted during a lookup for increasing network sizes. Figure 4.2 shows the results. As can be clearly seen, the growth fits a logarithmic curve perfectly, demonstrating that my implementation correctly adheres to this part of the standard.

4.3 Scalability

It is important that we prove the system scales to a large number of players. In order to do this without needing to purchase a large number of servers I performed several tests on a single instance. This was to test the performance under several different types of player load to verify that a single server can cope with a representative number of players.

I then created a 100 node network hosted on my dedicated server and tested it with 1000 simulated players to demonstrate the system functions correctly when deployed as a network in a large system. In this instance communication will have been slightly slower than usual due to communication being handled locally on the server, but this is acceptable as it still gives useful a lower bound on performance. I would note that to make this function correctly I had to increase the UDP and TCP buffer sizes on my server due to the high volume of traffic being exchanged internally between nodes.

4.3.1 Local Simulation

4.3.1.1 Setup

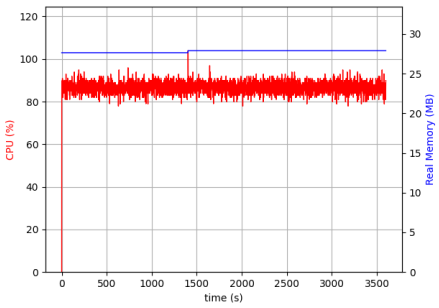
I performed 4 local simulations on a single node, each was run for 1 hour with variable player loads. I ran a single *VoxelPopuli* node on my dedicated server and connected one of my test client scripts to it to simulate player activity. The plots in figure 4.3 show the CPU and memory usage under various scenarios.

4.3.1.2 Analysis

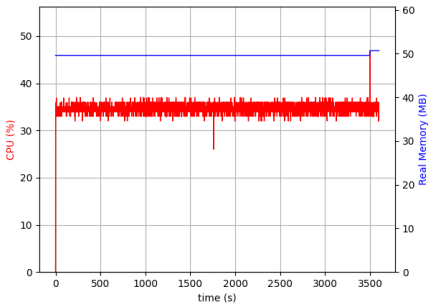
As can be seen from figure 4.3a the single thread performance is reaching 90% utilization at about 50 players in a single chunk. This is fairly typical of online games and it is at around this point that *Minecraft* servers begin to strain. This scenario is unlikely to be realised frequently in a deployed network running an MMO as this number of players tend not to occupy the same area concurrently.

In figure 4.3b we have 50 clients connected to 50 separate chunks on the same server. This performs better as it is making use of both CPU cores on my dedicated server. In line with this expectation CPU utilisation is approximately half that of the single chunk case. This also represents an unrealistic scenario as we are unlikely to have players evenly distributed across the world so evenly, there will be likely more clustering. This test does demonstrate, however, that many chunk threads can operate concurrently without incurring a significant performance penalty.

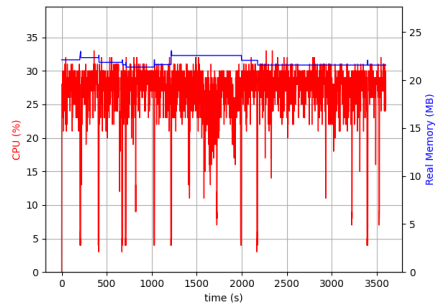
In figure 4.3c we instantiate 50 players and evenly distribute them over 10 chunks, this is a more realistic scenario and is designed to represent players playing with friends in small groups occupying the same areas. Performance



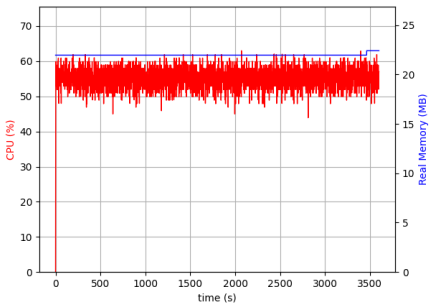
(a) Performance (50 players/1 chunk)



(b) Performance (50 players/50 chunks)



(c) Performance (50 players/10 chunks)



(d) Performance (100 players/10 chunks)

Figure 4.3: Plots showing CPU utilization and memory usage of *VoxelPopuli* nodes under different scenarios.

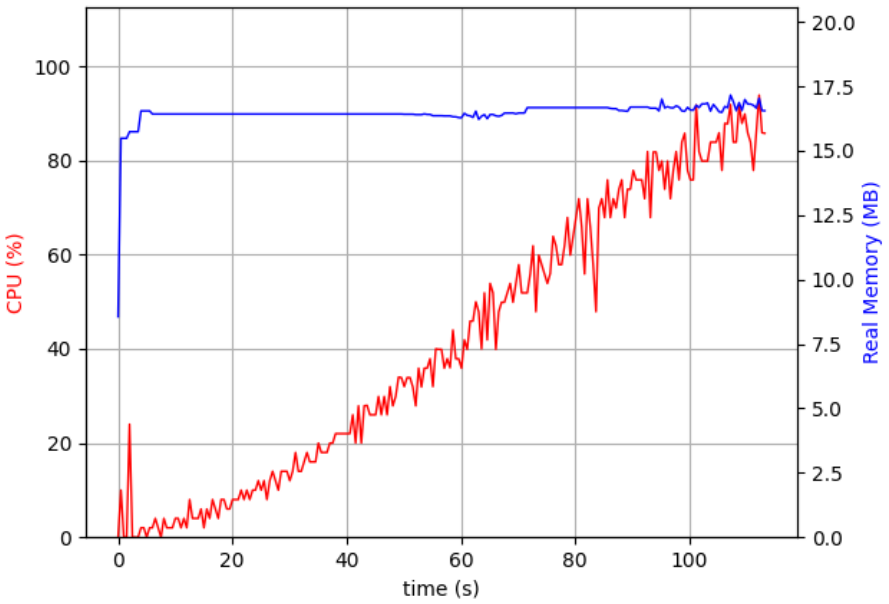


Figure 4.4: Performance as player count increases $(0 - 50)$.

here is improved over the 50 chunk case as expected, likely due to a reduction in the context-switching overhead. Averaging at around 25% utilisation the system is not struggling with this scenario, suggesting more players could be supported in a similar scenario. In order to test this I performed another test by doubling the number of players while keeping the number of chunks constant, in figure 4.3d we have 100 players across 10 chunks, and CPU utilisation averages around 55%. Suggesting a linear relationship between number of players and performance when the number of chunks is kept constant.

To verify the linearity I plotted CPU utilisation as player count was increased from one to fifty. I wrote a simple script to connect a new client every two seconds. The plot it shown in figure 4.4 and follows a roughly linear shape. This supports my theory that performance with constant number of chunks is proportional to the number of players connected.

I did a further test using the same script, instead adding a node every second and going from one to one-hundred. The resulting plot is shown in figure 4.5. The linear relationship persists up to around seventy nodes. However, we see CPU usage then plateaus at 100% and memory usage starts to rise, this is indicative that the system is struggling to handle both the garbage collector and the game server computation. Thus as a lower bound we can suggest that approximately seventy players can exist on a single chunk thread before performance issues begin to arise.

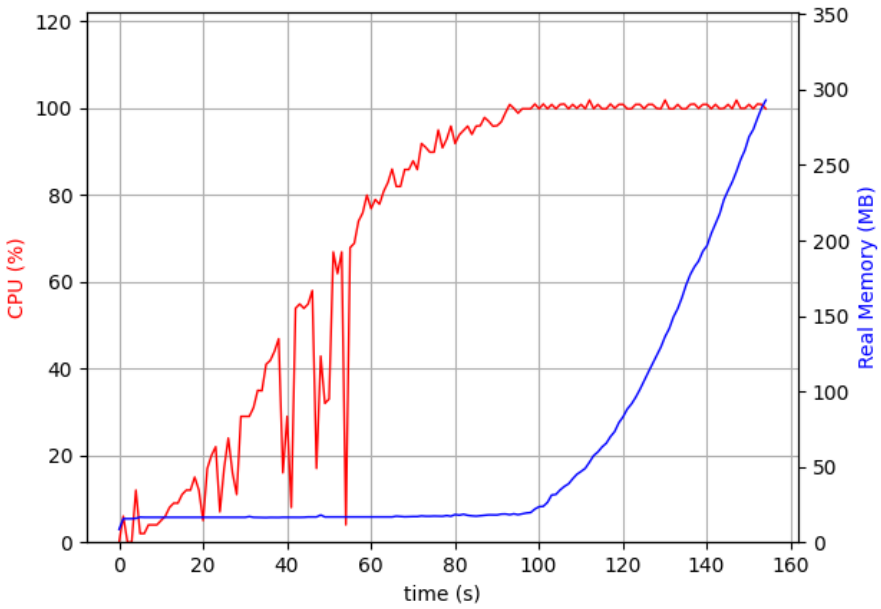


Figure 4.5: Performance as player count increases $(0 - 100)$.

Throughout this analysis the memory usage has remained approximately constant throughout the simulation, this is because memory usage is proportional to the number of chunks active and in each simulation this was constant. The memory overhead for a single chunk is below ten megabytes and consists only of the chunk data ($260KB$) and player data ($< 1KB$ per player) so this is not of major concern here. In a more complex system where there are other entities the server must keep track of such as monsters this may become a significant concern as it is for *Minecraft* servers, which can typically need hundreds of gigabytes of memory to function correctly.

4.3.2 Large Scale Simulation

4.3.2.1 Setup

To demonstrate my project functioning at scale I setup a 100 node network on my dedicated server and connected 1000 dummy players using `test_client.py` from my test client scripts. As some servers would have high load and others would have very low load depending on which chunks are loaded at any given time I found it more useful to plot the maximum CPU at any given time, as the average was very low (around half a percent). The plot is shown in figure 4.6.

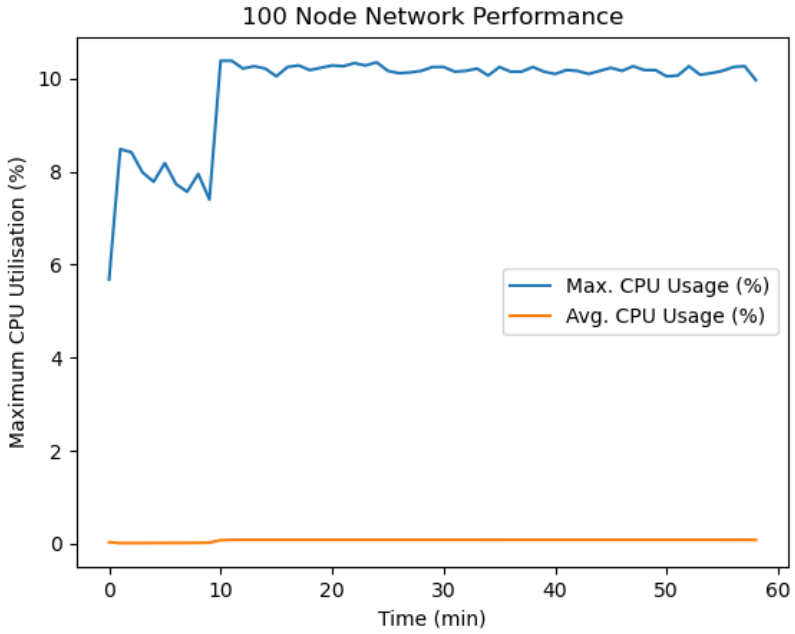


Figure 4.6: Performance of a 100 node network with 1000 players connected.

4.3.2.2 Analysis

Initially, the utilisation is not constant as nodes are still joining the network (it takes approximately ten minutes for 1000 nodes to join using the test client). Then we go into normal operation and the maximum load stays approximately at 10%, this is in line with our analysis in § 4.3.1 as there would be approximately ten nodes per server on average. The system was left running for 1 hour and experienced no problems, I was able to connect with my graphical client and explore the world, viewing the dummy players teleporting around randomly. This can be seen in figure 4.7.

4.4 Client

The client performs as required, the player is able to explore and interact with the world and the client is performant – running consistently at 60FPS+ (frames per second) on both my dedicated GPU (Nvidia GeForce GTX 1650 Max-Q) and my integrated graphics (Intel Iris Plus Graphics).

Figure 4.7: Screenshot of dummy players in the *VoxelPopuli* world.

4.5 Node Failure

The system performs correctly in the face of node failure. When the client loses connection to a node it queries the node it is using for DHT queries. This server finds the required node and discovers that it is not online. It thus performs the generate procedure (see algorithm 3) before returning the new node to the client. This takes at most a couple of seconds and produces minimal disruption to gameplay.

4.6 Current Limitations

4.6.1 Scale

I will now attempt to reason whether *VoxelPopuli* can handle a world the size of a typical MMO with a player count to match. As a baseline for player count I will be using the MMO *EVE Online* which had a peak concurrent player count $63,160$ [10] in 2011. The EVE world is sparse so I will use the *World of Warcraft* world size of approximately $200km^2$, estimates vary so I have taken the highest I could find [11]. Assuming each voxel in my game is $1m^3$, each chunk covers $1024m^2 \approx 0.001km^2$. Thus we would need $200,000$ chunks to represent this world. Assuming each server controls one hundred chunks we would need a network of 2000 nodes to store this world. This is well within the capacity of the *Kademlia* network.

Then we come to justifying that the game server component could handle this when evenly distributed. Assuming that players are evenly distributed

across the world that would put 32 players in each chunk, this is obviously going to overload the server. However, there are likely to be regions that are not currently loaded and regions which are more popular. So if we generously assume that 50% of regions are unloaded, then we have 64 players per loaded chunk and only 50 chunks loaded per server. This is still likely to exceed the performance limitations of the mid-range machine I used in my testing.

Alternatively, assume that the network is of size $20,000$, then we have ten chunks per node and still 32 players per chunk, this is more feasible. This approach might be realistic, consider that the peak active player count (this is not concurrent, but playing regularly) of *World of Warcraft* was twelve million [12]. Only 0.17% of players would need to setup a server for the system to be viable. However, there is no escaping that this is a clear limitation of my approach. The fact that each chunk operates its own thread is clearly debilitating to the performance of the system. An approach used by *SpatialOS* [4] is to have chunks grouped together spatially and computations performed on the entire region of space in one thread – this is an approach I would like to explore in my system in the future as I think it could drastically improve scalability.

4.6.2 Security

Kademlia is resistant to certain DoS attacks, the preference for old contacts ensures that flooding the system with new nodes will not cause existing nodes' state to be flushed, giving control of the network to the attacker. However, *Kademlia* remains vulnerable to various types of attack. Including but not limited to eclipse and Sybil attacks. Additionally, *Kademlia* does not prevent nodes from impersonating others when bootstrapping, creating ambiguity about who is the *true* owner of a particular ID. Ultimately, a determined attacker would be able to detrimentally influence the *VoxelPopuli* network should they so desire – and in an MMO context, where there would be motivation to cheat, this is almost guaranteed to happen.

Furthermore on this point, at present the game server trusts all requests coming from the player and does not attempt to verify the validity of their movements. This would be simple to implement – simply deny a player access to a chunk if the player's position recorded in the network is not near to it.

4.6.3 Data Loss

One significant issue with the current implementation is that when a node dies all the chunks stored within it are lost and will be regenerated by another node when they are needed. This means that anything a player has constructed within that chunk is lost. An approach I proposed as a potential extension to this project could be used to mitigate this problem. I propose that instead of having a single node responsible for each chunk, have a node and several backup

nodes, periodically the main node will multicast its state to the backup nodes, should this node die, the backup nodes elect a successor and then induct a new node into the backup set. This could be done simply using the n nearest nodes to the chunk's key.

Chapter 5

Conclusion

Bibliography

[1] Maymounkov, P. and Mazières, D. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf. Accessed: 2019-10-16.

[2] "Sharding" on Wikipedia. https://en.wikipedia.org/wiki/Shard_(database_architecture). Accessed: 2019-10-15.

[3] "Club Penguin is shutting down" – TechCrunch. https://techcrunch.com/2017/01/31/club-penguin-is-shutting-down/. Accessed: 2019-10-15.

[4] SpatialOS by Improbable. https://improbable.io/spatialos. Accessed: 2020-03-20.

[5] *Kademlia* Python Library. https://github.com/bmuller/kademlia/. Accessed: 2020-03-20.

[6] *Source* Engine Multiplayer Networking, Valve. https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking. Accessed: 25-3-2020.

[7] "Perlin Noise" on Wikipedia. https://en.wikipedia.org/wiki/Perlin_noise. Accessed: 2019-10-17.

[8] "Simplex Noise" on Wikipedia. https://en.wikipedia.org/wiki/Simplex_noise. Accessed: 2019-03-27.

[9] *OpenDHT* on GitHub. https://github.com/savoirfairelinux/opendht. Accessed: 2020-04-16.

[10] "Eve Online has over 360,000 players. 63,170 simultaneous users in January" – PC Gamer. https://www.pcgamer.com/eve-online-has-over-360000-players-63170-simultaneous-users-in-january/. Accessed: 2020-04-17.

[11] Estimation of World of Warcraft world size. http://tobolds.blogspot.com/2007/01/how-big-is-azeroth.html. Accessed: 2020-04-17.

[12] "World of Warcraft Classic: Hit game goes back to basics " – BBC News. https://www.bbc.co.uk/news/technology-49448935. Accessed: 2020-04-17.

Appendix A

Proposal