Samuel J. Sully

Voxel Populi: 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 an individual chunk for 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

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 using Amazon Web Services.

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 *AWS EC2* Virtual Private Servers 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 **LINK TO**.

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 chunk 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 IA and Part IB. I will also need to implement the algorithm for locating and

[3] Usually 1 hour.

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

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.

2.6 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

[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.

stone) in order to procude a *Minecraft*-like world. The structure of the world 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.

3.1 Kademlia

3.2 Game Server

3.3 Client

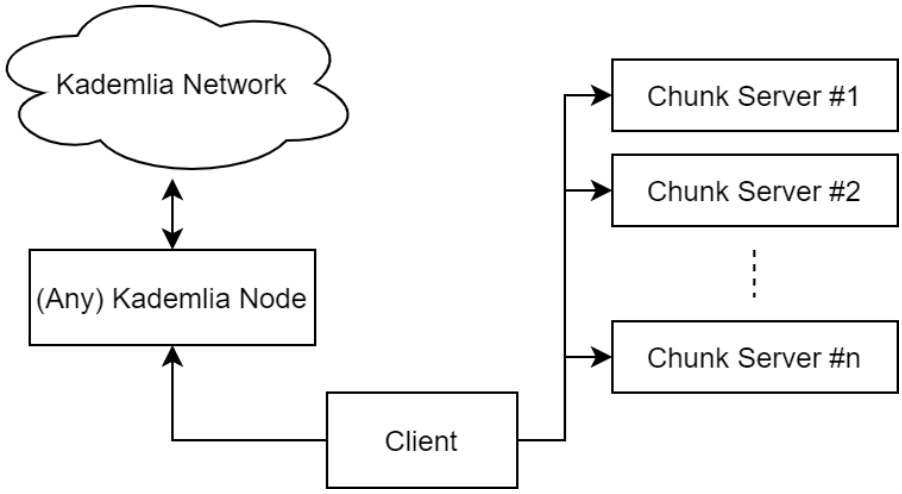

Figure 3.1: Diagram giving an overview of *Voxel Populi* architecture. Note that the client needs to know only the IP and port of any node in the Kademlia network, which it will use to access the DHT and query the IP and port pairs of the nodes containing world data it requires.

Chapter 4

Evaluation

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". 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.

Appendix A

Proposal