ASC Report

Optimising Networked Multiplayer Games by Transferring Computation to Clients

Harry Slatyer · u4671691
Malcolm Gill · u4669875

Supervisor:
Dr Eric McCreath

Advanced Studies Course (6 units)

November 16, 2011

Abstract

Networked multiplayer video games have become enormously popular over the past two decades. Recently the capabilities of traditionally-designed game servers have begun to be outstripped by the demands of vast numbers of players, motivating re-thinking of the fundamental server design.

This project proposes a novel architecture for highly scalable networked multiplayer games, and examines the potential for its practical application. The proposed architecture differs from traditional models in its distributed nature, being designed to harness the collective processing power of all connected clients rather than creating a performance bottleneck by consigning all game state calculations to one central server.

A game server is implemented using this architecture, and its performance measured. The results indicate that the proposed distributed architecture can yield orders of magnitude performance improvements over a centralised architecture, with only minor additional networking overhead required. Such an architecture could therefore be utilised to improve the scalability of real-world games which use old-fashioned centralised designs.

Contents

Acknowledgements

First and foremost we would like to thank Dr Eric McCreath, our supervisor, for his suggestions and feedback throughout the development of this project. Also thanks to Thomas Shafron, without whose common sense and bug-catching skills much of our data collection could have been flawed, and finally thanks to Nicholas Gurewitch, for his sense of humour.

Acronyms

AI artificial intelligence. 17, 18, 23

FPS frames per second. 6, 22–28, 30

LAN local area network. 21–23, 25–28, 30

MMOG massively multiplayer online game. 3, 6, 8, 10, 11, 24, 28, 30

NPC non-player character. 10, 12–30

P2P peer-to-peer. 6–9

SFML the Simple and Fast Multimedia Library. 20

STL C++ Standard Template Library. 19

TCP the Transmission Control Protocol. 19

UDP the User Datagram Protocol. 19, 21, 23

1 Introduction

From its humble beginnings in the 1960s, the video game industry has grown explosively. In 2010, the industry's global revenue was over USD 60 billion[1], eclipsing both the USD 16 billion music industry[2] and the USD 10 billion film industry[3]. Video games are played by over two-thirds of people in Australia and America, by all ages and genders[4, 5].

During the 1990s, increasingly widespread Internet access sparked commercial development of networked multiplayer video games. As Internet technology has improved, faster speeds have allowed an ever greater number of players to interact simultaneously with one another online, and massively multiplayer online games (MMOGs) have consequently experienced greatly increasing popularity. Throughout the 1990s and 2000s, the number of active MMOG players increased exponentially, with estimates for the worldwide 2011 figure at over eleven million[6].

As increasingly many people have taken up MMOGs, game designers have begun to face technical challenges. While it is possible for a single game server to handle a relatively small player base, this becomes rapidly less viable as the player count increases. For larger numbers of players, a single game server simply cannot keep up with the processing and networking demands of all of its connected players, leading to significantly less enjoyable gameplay for all involved. This is a bad thing.

To address this problem, some game designers have embraced design architectures different from the traditional server-client model[7]. In this project a model is implemented where one game server distributes work amongst all of its connected clients, harnessing their collective processing power to reduce server load and increase scalability, while retaining certain desirable features of the traditional architecture. A simple proof-of-concept game is implemented using this model, and tested to gauge its potential.

This work sits alongside that of those who have presented other novel, highly scalable designs for networked multiplayer game servers[7, 8, 9]. Our hope is that our architecture may be used to improve the performance of real-world games.

The rest of this report is structured as follows. Section 2 gives a formal definition of a game and describes several of the network architectures commonly used in contemporary game design. Section 3 lists the specific goals of our project, along with some of the industry standards against which our game will be compared. Section 4 details the design of each component of the game, explaining the important algorithms used as well as describing the overarching code structure. Section 5 focuses on implementation details, and lists the external libraries used. Section 6 presents the results of load-testing the game server, and discusses their implications, the merits of our chosen architecture compared to others, and the relative performance of our game compared to commercial products.

Section 7 concludes by summarising the results of the project and stating the relevance of our work to current research.

2 Background

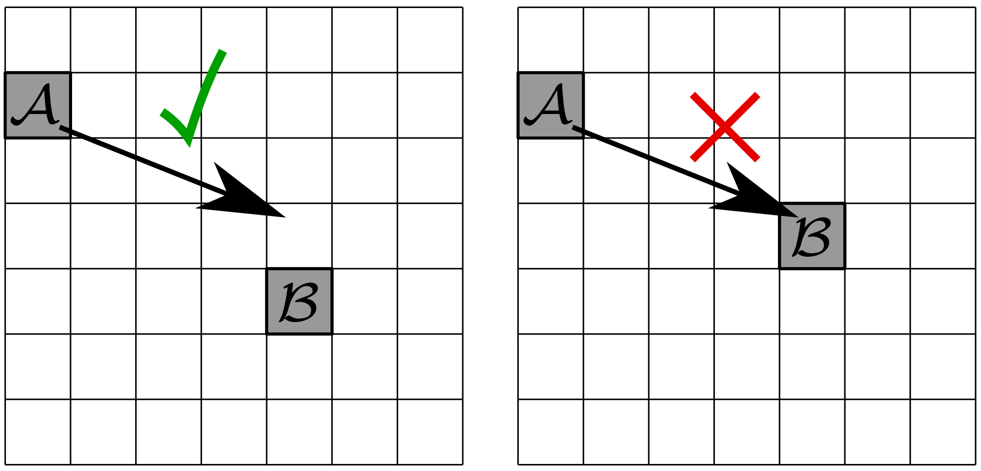
2.1 Games

For our purposes, a ‘game’ consists of a state (a collection of variables) and a set of rules (functions) which, when applied to a state, yield a new state. Users of the game are able to modify the state by applying rules or requesting that rules be applied.

For example, a very simple single-player game might have its state given by a single integer x describing the position of a character, and one rule that x may be incremented or decremented, according to the user’s choice.

In a single-player game, the state is simply maintained by the user and updated according to the rules whenever necessary. In a multiplayer game, however, each user’s state must agree with that of all other users to avoid inconsistencies when users attempt to interact.

For example, consider a simple game in which each user controls the position of a character, and no two characters are allowed to occupy the same position. Suppose there are two users, A and B, controlling players $\mathcal{A}$ and $\mathcal{B}$. Suppose further that the position of $\mathcal{A}$ according to both A and B is $P_{\mathcal{A}}$, while the position of $\mathcal{B}$ is $P_{\mathcal{B}}$ according to A, but $P'_{\mathcal{B}} \neq P_{\mathcal{B}}$ according to B. If A were to attempt to move $\mathcal{A}$ to position $P'_{\mathcal{B}}$, this would be allowed by A, but not by B: the inconsistent game states give rise to undefined behaviour. Figure 1 shows a diagram of this situation.



(a) State according to A permits movement. (b) State according to B forbids it.

Figure 1: How differences in state according to two users can cause inconsistencies.

The most straightforward way to retain consistency of the game states in a multiplayer game is by running a server which maintains the official game state, synchronising every client's individual game state with the server's own. This is a centralised design. More recently, in an effort to achieve greater scalability while reducing running costs, some game developers have instead used designs with no central server[9, 10], sharing the processing and network load between all the clients instead. This is a distributed design.

2.2 Centralised design

Traditionally, a multiplayer game has consisted of a central server connected to every client. Until the mid-2000s, almost every commercial multiplayer game used this architecture[11]. The server stores and updates the game state, sends information about the state to each of the clients, and handles requests from the clients to update the state. Each user controls a client, which provides an interface between the user and the server by drawing the game state to the screen and processing user input, sending appropriate requests to the server to apply game rules. This design approach is shown schematically in Figure 2.

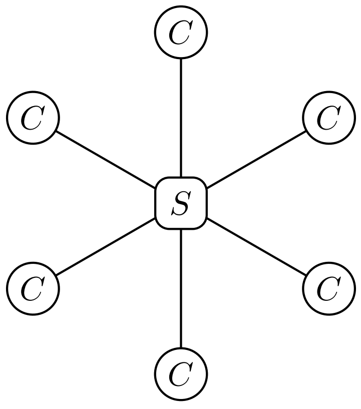


Figure 2: The server S maintains the game state, sending it to each client C.

This design has two strong points. First, if there is only one official game state, then it is easy to avoid inconsistencies between clients' individual states. The second advantage of this method is its security. Clients can only manipulate the game state by sending requests to the server, and these requests can be validated by the server, to ensure that users are not trying to cheat, or to ignore them if they are.

One problem with this centralised design is that all game state processing must be done on the server, creating a performance bottleneck. Consider the simple game described previously, which allows each user to move a character around a map such that no two characters can occupy the same position. If there are n users, then each frame the server must ensure that none of the n characters collides with any other. This can be determined in $O(\log n)$ time with a binary search, and must be done for every character, so collision detection alone requires $O(n \log n)$ time every frame. If the server is to run at, for example, 100 frames per second (FPS), even a powerful computer would only be able handle on the order of 100 000 users of this trivial game[1]. In practice, a real game requires orders of magnitude more processing to be performed than the simple game described here, so this optimistic estimate of 100 000 would shrink correspondingly, becoming a serious limitation.

Network bandwidth requirements pose a similar restriction on the centralised server design. Each user would need to be sent the positions of other characters around five times per second to ensure smooth gameplay. Each character only needs to be sent the positions of nearby (visible) characters; assume there are m of these on average. The position of a character would fit into approximately ten bytes, so each user would need to be sent $5 \times m \times 10 = 50m$ bytes per second. For all characters, then, the server would need to send about $50mn$ bytes per second. Assuming at most about 1% of the characters in the game are visible to any individual character at any time gives $m = 0.01n$, so the server must send $0.5n^2$ bytes per second. Even with a gigabit network connection, n could be at most around 15 000. Thus even a powerful server with a fast Internet connection could only handle on the order of 10 000 simultaneously connected users of this extremely simple game.

It is clear that while a centralised design is simple and secure, it is both difficult and expensive to create a scalable server using it. For modern MMOGs, which are expected to cope with thousands of simultaneously connected users, this poses a significant problem.

2.3 Distributed design

The general idea behind distributed design is to spread processing across several machines, rather than relying on a single server. This reduces the workload on each individual computer, and can dramatically improve the scalability of the system.

2.3.1 Peer-to-peer

In a peer-to-peer (P2P) system there is no dedicated server, and instead each peer maintains its own version of the game state. This design is pictured in Figure 3. Each user controls a peer, which responds to user input by updating its own game state and notifying other peers of this change. The main advantage of this is that each peer need only

[1]Assuming $n \log_2 n$ comparisons are performed and each of these takes *roughly* ten clock cycles (an average value, accounting for memory access time), running at 100 FPS would require $1000n \log_2 n$ cycles per second. For $n = 100\,000$ this equates to around 1.7 GHz. Thus this figure is a reasonable order of magnitude estimate for the maximum number of users.

perform processing and send messages associated with its own user's character.

This makes the load on any individual peer much less than that on a centralised server. For example, the collision detection described in Section 2.2 becomes an $O(\log n)$ operation for each peer, and each peer would only need to send around fifty bytes per second (ten bytes to store the character's position, sent five times per second) to each nearby peer in order to inform these peers of the character's position. Thus roughly the same amount of processing power and bandwidth are required for P2P and client-server systems, but a well-implemented P2P system spreads these loads uniformly across all peers, while a centralised system places great load on the server, and very little on the clients. It follows that as the number of users increases, and the load on the system consequently rises, the processing power and bandwidth available increase. For this reason, a P2P design is inherently more scalable than the client-server model.

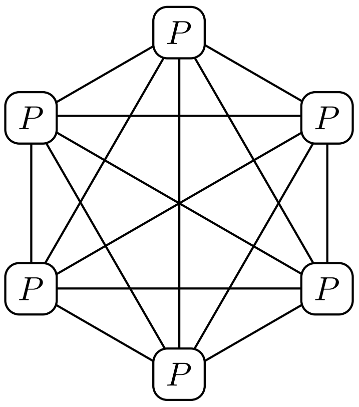


Figure 3: Each peer P maintains its own game state and synchronises with the others.

Another important practical advantage of the P2P design is the relatively inexpensive maintenance of the system. A client-server system requires a powerful server to be maintained, along with a fast Internet connection, while a game utilising the P2P design requires no such infrastructure. Moreover, a server in the client-server architecture must be equipped to handle peak load, but for most of the time will not be under such stress: the centralised design makes inefficient use of its expensive resources.

Because not all traffic must go via a central server, a P2P design typically gives better network latencies for clients. For information to propagate between any two clients A and B in a centralised architecture, the information must travel the extra distance between A and the server, and then the server and B; with a P2P network, however, the information travels as directly as possible from A to B. Thus the average latency between clients can be reduced.

One significant drawback of the P2P design is its relative insecurity. Having no centralised server maintaining an official game state makes it easy for one client to give false information about the game state to others. In order to prevent cheating, then, games using this architecture must include code to try to detect and punish peers on the network who are not playing by the rules. Naturally, this adds a good deal of complexity to the game code, and in practice it is hard to imagine that a P2P game can ever be as well-protected against cheaters as a centralised game.

Another potential drawback of the P2P design is the possibility of reduced network stability. The server in a centralised system is expected to run reliably and with good performance, but when no such server exists each peer is at the mercy of the others. If one peer has a low-quality network connection then it can impact directly on all other peers, whereas under the centralised architecture, clients do not interact directly with one another, so this is not a problem.

The P2P design has been used in a number of real-world games. *MiMaze* was an early proof-of-concept, the first 3D multiplayer game to be designed with a distributed architecture[12]. *Age of Empires* and its sequel are two commercially successful examples[10]. The low latencies and costs of this architecture make it particularly appealing for popular first-person shooter games, where network lag is very important and providing servers for a large number of players can be very expensive; as a result of this, some recent first-person shooter games have abandoned the traditional centralised model entirely, and are fully P2P. *Call of Duty: Modern Warfare 2* is one such title[13].

2.3.2 Mirrored-server

The mirrored-server architecture lies in some ways between the P2P and centralised designs. Rather than having one central server, this approach employs several, all communicating with one another to keep their individual game states synchronised. Each client connects to whichever server offers the lowest latency, and because of this, the average network response time can be improved for all clients. Figure 4 represents the mirrored-server design.

This approach is of particular use when the game server becomes limited by its network connection, rather than by processing power. In this situation, additional mirrored servers can be created, increasing the overall client capacity. This may come at the cost of increased latency between clients on different servers, if server-server game state synchronisation is performed less frequently than server-client synchronisation. Several commercial MMOGs have had success following this design pattern, such as Sony's *PlanetSide* and Lidden Lab's *Second Life*[11], and the first-person shooter *Quake* was also successfully adapted to use this network topology[8].

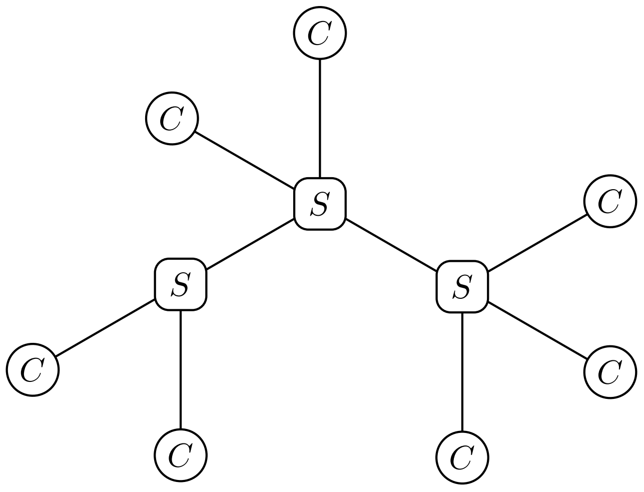


Figure 4: Seven clients C distributed between three mirrored servers S.

2.3.3 Server-to-peers

The server-to-peers architecture is another compromise between a centralised design and the fully distributed P2P architecture. A central server still exists, but every client is capable of running the game simulation as well. The server can take advantage of this fact by requesting that clients apply game rules, calculating changes to the game state and sending the results back to the server. Figure 5 shows this network design.

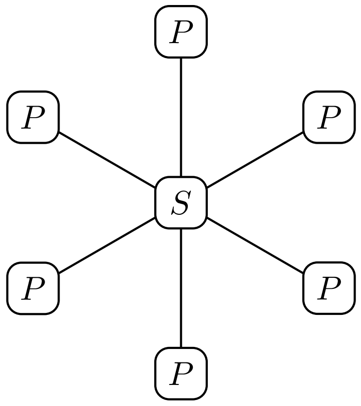


Figure 5: The server S and all peers P are capable of running the simulation, but the official game state is maintained by the central server.

With a prudent choice of which tasks to assign to connected clients, this can significantly reduce processing requirements on the server. Clearly the greatest advantage is gained from such a system when the tasks given to clients are computationally intensive but require minimal information to be transmitted over the network. One such task is pathfinding: the server need only send origin and destination coordinates to a client, and the client can calculate the best path between the points, then send back a list of coordinates specifying the path for the server.

This approach does allow for the possibility of clients returning deliberately false information to the server. For example, if in controlling a non-player character (NPC) the server requested paths from clients, one client could try to return paths leading the NPC toward a particular destination, to gain some advantage. The server can easily determine whether a returned path is valid, but not necessarily optimal, so this problem of clients giving false information is thus not entirely avoidable in general. However, the potential harm caused can be minimised by randomising which clients are asked for which information, so that even if one client gives bad data, the mistake will be corrected within a short time when another client is queried.

2.4 Measuring performance

The canonical measure of the performance of an MMOG is the number of users who can simultaneously play the game before lag effects reduce the gameplay quality to an unacceptable level. This number will be referred to as the 'capacity' of the game. The relatively subjective nature of this definition, and the difficulty of finding sufficiently many people (potentially numbering in the thousands) to agree to play the game at a particular time, mean that it is almost impossible to determine this number solely through real-world testing. For server-client, mirrored-server and server-to-peers designs, the performance bottlenecks are almost guaranteed to occur at the server, because the minimum CPU and bandwidth requirements of the clients are typically low. This means that the framerate and network activity of the server can be monitored for relatively low numbers of users, and the data extrapolated to determine the number of users required to cause the framerate or network activity to reach unacceptable levels.

3 Requirements

3.1 Objective

The overarching goal of this project was to develop an MMOG which used the server-to-peers architecture, and demonstrate that this can provide a significant performance advantage over the same game implemented with a server-client model. More precisely, we aimed to implement server and client programs satisfying the following three conditions.

- A user should be able to connect to the server via the client program and interact with the game world, including other users.

- The server should be able to handle a large number of simultaneously connected clients while maintaining a specified framerate and without exceeding its bandwidth limitations. This number of clients should be comparable to the capacities of commercial MMOG servers.

- The server and client should subscribe to the server-to-peers architecture. That is, the client should be capable, at least partially, of updating the game state, and the server should be able to make use of this to maximise its capacity.

3.2 Industry standards

A simple way to grade our server's performance is to compare it against that of commercial MMOGs. Although the mechanics of our gameplay are significantly more simple than those of real-world examples, the underlying concepts should largely be the same: once the foundations of player movement, collision detection and range searching (the operation of determining the set of characters within a given area) have been laid, any further gameplay details can be built from these without introducing additional algorithmic complexity to the game loop. Moreover, the (relatively) small performance overhead incurred from the added complexities commercial games contain could well be offset by the large amounts of time and money that are invested in their development, and in the hardware they use. For these reasons, we believe that a rough performance comparison between commercial game servers and our own can provide meaningful results.

The server capacities of two well-known commercial MMOGs are given below.

3.2.1 EVE Online

One of the most popular MMOGs for which these performance statistics are freely available to the public is *EVE Online*. Game logic and processing for *EVE Online* are performed by around 100 separate physical servers, with each handling different areas of the game universe[14]. In 2008, one server, powered by a server-grade 3.0 GHz Intel Woodcrest processor, handled up to 1000 active players simultaneously before the server's framerate became unacceptable[14].

The *EVE Online* server updates its game state once per second[15].

3.2.2 World of Warcraft

Undoubtedly the most popular MMOG at present is Blizzard's *World of Warcraft*[6]. While Blizzard refuse to release official performance statistics, unofficial sources suggest that a *World of Warcraft* server can handle roughly a few thousand clients[7, 16, 17].

4 Design

The implementation of this project consisted of three components: a server, a client and a bot. The server is responsible for maintaining the official game state, and communicates with clients over the network, receiving instructions from and sending updated game state information to each client. The client provides a simple visual representation of the game environment based on data received from the server, and accepts user input, transforms it to instructions and sends them to the server. The bot is essentially an automated client, programmed to move randomly around the game world and designed for the purpose of load-testing the server.

An object-oriented design was used for each of these components. This paradigm has a strong tradition in game engine architecture, with the majority of past commercial games having been developed this way[18]; for this reason, it seemed a natural choice for this project. Each of the three components contains a single hierarchy of objects, with each object responsible for updating and maintaining its child objects. The components are all single-threaded (though the server spawns a second process to handle console input for administrative control). The straightforward nature of this object-oriented, single-threaded design means that communication between the various parts within each component can be done simply via the object hierarchy, with no need for inter-thread communication or a complicated inter-object message-passing system. It is likely that this approach would be too simplistic for practical application in a more complicated real-world game, but the added complexities only require additional bookkeeping, rather than introducing new algorithmic challenges, so were not necessary for consideration in this project.

The following three subsections describe the designs of the server, client and bot.

4.1 Server

There were several important decisions to be made in what was otherwise a relatively routine design process for the server. These were associated with how the entire game world could be distributed amongst several servers, how characters were to be stored in the server, and which methods should be used for collision detection and pathfinding These decisions are discussed below.

The virtual environment of the game is split into several distinct 'worlds'. Players and NPCs in one world cannot interact with those in another, but they can move between worlds by performing certain actions (walking to the edge of their current world, for example). Each server process can handle several worlds, but all worlds need not run on the same server. This means that some servers can be dedicated to running single worlds expected to have relatively high numbers of players, while other servers can run several less popular worlds. A diagram of this situation is shown in Figure 6.

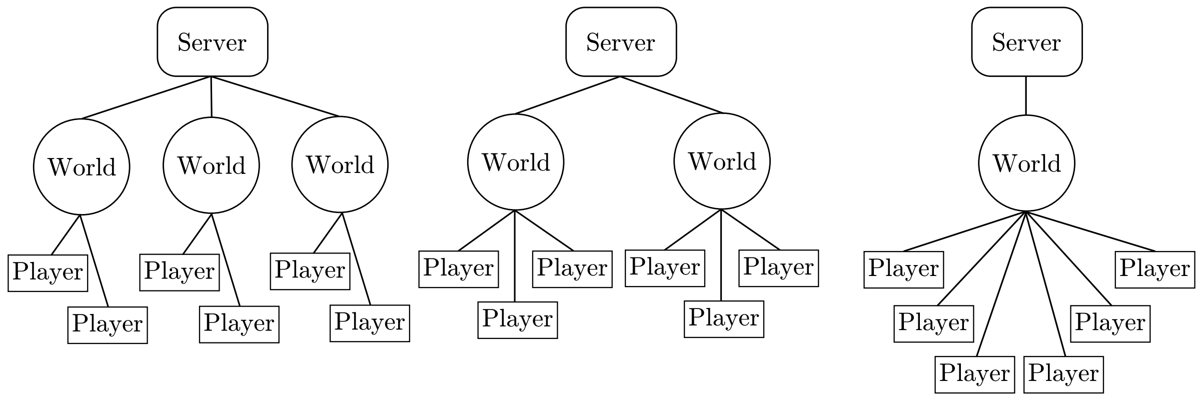


Figure 6: An assignment of six worlds of different popularity to three servers, spreading players evenly across the servers.

Within each world, any entity (player, NPC or object) can only be affected by other entities within a certain distance of itself. Operations such as collision detection, which may naïvely be implemented by checking each player against every other player, can thus be optimised by considering only entities close to one other. Similarly, each player need only be sent information about nearby, visible entities, significantly reducing the network bandwidth requirements of the server.

In order to benefit from these facts, the server must be able to find all characters within a given rectangle efficiently. This is operation is known as 'range searching'; a discussion of various data structures and algorithms for range searching can be found in [19]. For collision detection, the rectangle being range-searched need only be slightly larger than a single character, while for determining what information should be sent to each client, the region should be the player's field of view.

Each character's position is updated each frame according to the current direction of travel, and the server is notified by the appropriate client of any changes to this direction. In order to reduce the load on the server, collision detection simply determines whether or not a player is able to move in the desired direction, and it is the client's responsibility to adjust the player's direction to avoid any obstacles. For example, if a player is moving in a diagonal direction and encounters a horizontal wall, the server will completely stop the player, rather than automatically changing the movement direction to be parallel to the wall. Making the client perform its own collision detection and corresponding direction changes in this way reduces the workload on the server.

NPC movement is implemented with a system where each NPC has a target coordinate, its 'waypoint', to which it tries to move. Currently NPCs have a hard-coded list of waypoints which they attempt to visit in sequence, but it would be simple to implement a more complicated system where certain in-game events determine an NPC's waypoint (for example, if a player moved within a certain distance of a hostile NPC, the NPC's waypoint could be set to the player's position). Actually determining a path from an NPC's current position to its waypoint is a non-trivial and potentially time-consuming operation, because such a path must not pass through any solid objects or characters. For this reason, the server can query a randomly-selected nearby client for a path between two specified points. The queried client then calculates such a path and sends it back to the server. It is then trivial for the server to ensure that the NPC moves along this path to the target waypoint.

4.1.1 Class list

There were five important classes in the code for the server, pictured in Figure 7. The main class was Server, which contained objects of type World, each of which contained a Map and collections of Players and NPCs. Details are given below.

Server is the main server class. It maintains a list of connected clients (along with the connections to these clients) and Worlds. Every frame the Server object starts by handling any incoming network activity, such as connections, disconnections and information packets from clients (these packets are usually passed on to the player's World). The Server object then updates each of its Worlds, and finally fills a collection of outgoing packets with information provided by the Worlds, sending these to the appropriate clients. Requiring all network activity to pass through the Server object (as opposed to allowing each World to send information over the network directly) allows the server to control the relative bandwidth requirements of the different Worlds easily.

World is the class representing each world controlled by a server. It maintains a list of Players and NPCs contained in the world, along with information about the map where the characters move and interact. When the World is updated, first it updates the positions of each character, based on the current direction of movement, and checks for resultant collisions. It then determines which clients are due to be sent information about the positions of nearby characters, and constructs the appropriate packets to be sent over the network (these will be sent when the Server next queries the World for outgoing packets). Finally, the World object updates each NPC, determines whether any NPCs require new paths to be found, and constructs packets requesting any such paths, to be sent to randomly-selected nearby

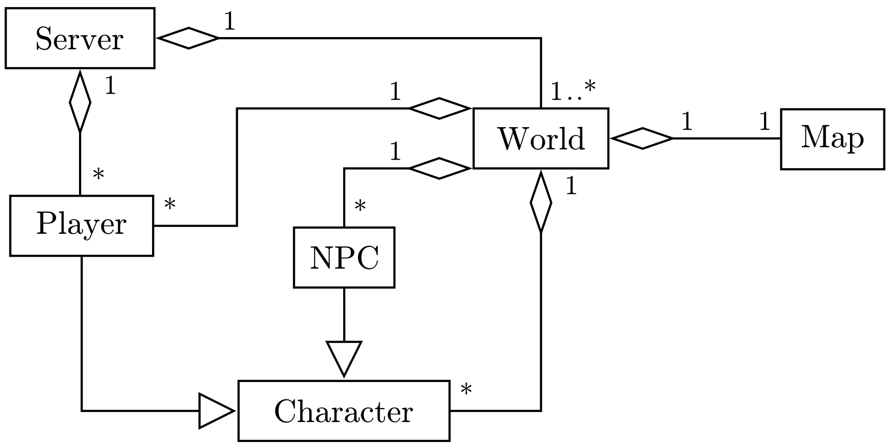


Figure 7: Relationship diagram of the important Server classes.

clients. The World is also notified by the Server of any incoming packets from its players, and handles these appropriately.

Player stores the network identification, name, current World, World position, current direction of movement, and any packets which are to be sent to the client of a player. This class is also used by the World for sending packets to players.

NPC represents a non-player character in the game. Each NPC object contains the NPC's name, position and direction of movement, as well as information about the NPC's route (the list of target waypoints) and path between the current position and the next waypoint. When an NPC object is updated by a World object, it calculates the direction to move based on its current position in its path.

Map stores the map for each World, containing information about the dimensions of the space available and the location of any solid objects through which characters cannot walk (walls, for example).

4.2 Client

As with the server, much of the client design was fairly standard. The most important features of note are its 'state' system and the techniques used for NPC pathfinding, both of which are discussed below.

The client's state system maintains a stack of States, using the top State to determine the program's behaviour at any given time. Each State describes a different stage of the program's execution; for example, initially the stack contains a single StartState, which causes the program to display an interface for the user to enter the player's name and the IP address of a server to join. When the user has entered these details, a new GameState is pushed on top of the initial State. This creates a connection to the server and displays the game world, and allows the user to play the game until he or she presses the Escape key, at which point the GameState is popped from the stack and the StartState becomes the top element again. This displays the interface for entering the player's name and the server address, so the user can either join a different server (causing the GameState to be pushed back on top of the stack) or quit the program. This system allows extra features to be added to the client with relative ease: an 'inventory state', for example, could be pushed on to the stack from the GameState if the user wanted to see the items in the player's inventory.

For the purpose of NPC pathfinding, the client must be able to find a valid path (one which does not pass through any solid objects or characters) between any two points. We chose to model the world as a graph, where vertices represent unoccupied positions and edges join vertices to represent adjacent positions on the map. We then used a simple breadth-first search to find the shortest path between the vertices corresponding to the start and end positions. While this is a fairly slow operation (the time complexity is $O(V+E)$, where V is the number of vertices in the graph and E is the number edges[19]), each client should need to perform it quite infrequently.

4.2.1 Class list

The client consisted of seven important classes. The main Client class ultimately contained two State subclasses, each of which determined program behaviour at a particular time. StartState was used when launching the program and connecting to a server, using the Input helper class for user interaction. GameState handled drawing the Map and World, and using the PathFinder, during normal gameplay. More information about these classes is given in the following list.

Client is the main client class. It initialises the graphics subsystem and maintains the stack of States, and keeps an Input object to control all keyboard input. Each frame, the Client object updates the State on top of the stack, and, based on the return value of the State's 'update' method, either leaves the State on top of the stack, pops the State off the stack, or pushes the State's child to the top of the stack. The Client then updates its Input object to check for any keyboard events, handling these appropriately.

Input detects key presses and key releases, and notifies the State on top of the stack of these events.

StartState is the initial State on the stack. It displays an interface allowing the user to enter a player name and a server IP address. When this information has been entered, the StartState object requests (via the return value of its 'update' method) that its child, namely a GameState object, be pushed to the top of the stack.

GameState is the main State in the client program. It provides an interface for the player to move around and interact with the game world. When a GameState object is pushed on to the stack it attempts to connect to the server specified by the user. If no connection can be established, the GameState immediately requests to be popped from the stack. Otherwise the GameState sends the player's name to the server, while waiting for a packet from the server containing information about the World which the player is about to join. When this information is received, the GameState initialises a World object appropriately. From this point on the GameState object communicates with the server to allow the user to control the player with the keyboard, chat with other users and change into different Worlds. The GameState object also passes information about other characters (such as their names and positions) to the World object, and uses a PathFinder object to perform NPC pathfinding at the server's request.

World stores information about the game world, namely a list of characters in the world, a sublist containing those characters which are visible to the user, and a Map object. World also handles drawing the game to the screen, and can be used, for pathfinding, to determine whether a particular position in the world is occupied by either a character or a solid object on the map.

Map loads a map from a local file and stores it in memory. A Map object can draw its map to the screen, and detect whether a character at a given position is colliding with any solid objects on the map.

PathFinder is the class used to find paths between any two points in a given World. The object first performs a breadth-first search to find the shortest path between the two points not passing through any solid map objects or characters, and then converts this path (which is initially a sequence of adjacent positions) into a list of waypoints so that a character can move in a straight horizontal or vertical line from each waypoint to the next without colliding with another character or object. An example of a path, with the corresponding waypoints, is shown in Figure 8.

4.3 Bot

The bot is based heavily upon the client, but does not require a stack of States, because only the GameState is used. The bot uses the same algorithm as the client for NPC pathfinding. The only other notable feature of the bot is its artificial intelligence (AI).

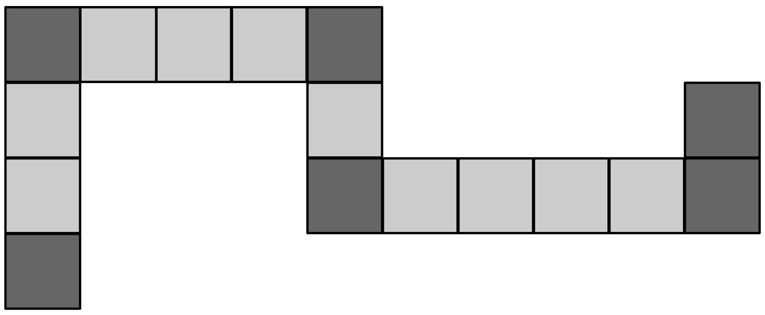


Figure 8: An example of a path with waypoints shaded.

The bot uses a simple AI system, choosing a random direction each second and walking in that direction until the next second. While this is not particularly realistic, it ensures that the bot induces a fairly similar strain on the server to a human client, by frequently requesting direction changes and causing collisions.

4.3.1 Class list

The bot reuses the GameState, World, Map and PathFinder classes from the client, and introduces a new class, Bot. While World, Map and PathFinder have exactly the same functionality as in the client program, GameState is used differently. These differences, along with the Bot class, are detailed in the list below.

Bot is the main bot class. It starts by initialising a GameState object with a player name and server IP address specified by commandline arguments, and then updates this object until disconnected from the server.

GameState is identical to the class in the client program, but is used in a slightly different manner. In particular, it is never pushed on to a stack of states (because no such stack exists in the bot program), its 'draw' method is never called (because the bot does not need to display any graphics), and the methods for handling key events are called by the Bot object (rather than an Input object) to move the bot's character in random directions. However, all other functions of GameState (communicating with the server, storing character information in a World object, performing NPC pathfinding with a PathFinder object, and so on) are used by the bot program, which ensures that the bot and client programs are indistinguishable from the server's point of view.

5 Implementation

The server, client and bot described in the preceding section were all implemented in C++, making extensive use of the C++ Standard Template Library (STL). Several other external libraries were also included, and their uses are described in the following subsections.

5.1 Server

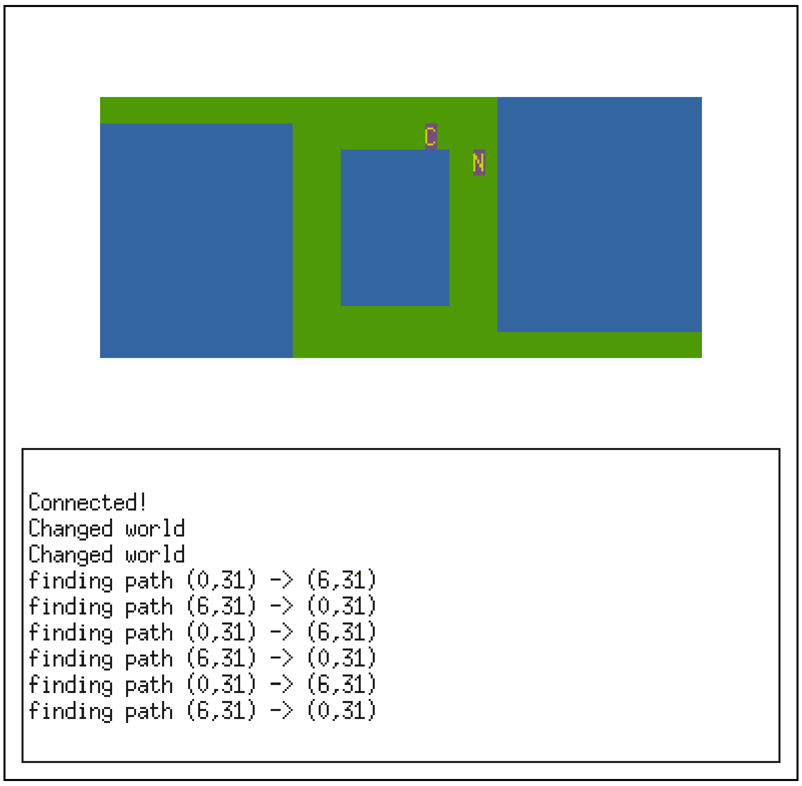
The server and client use the ENet library for networking, which provides a wrapper around the User Datagram Protocol (UDP) supporting optional reliable packets (that is, packets which are guaranteed to reach their destination)[20]. We chose not to use the Transmission Control Protocol (TCP), which automatically ensures reliability of packets at the expense of speed, because some information sent between the server and client need not be reliable, rendering the speed sacrifice unnecessary. For example, position updates, sent from the server to each ten times per second, are so frequent that it does not matter if a small number of these packets are lost, because the client can simply extrapolate the positions of nearby characters, guessing their locations based on previous positions and directions (which will be from at most around 0.5 seconds in the past).

We chose to store players and NPCs in a k-d tree (with $k = 2$) provided by the libkdtree++ library[21]. This allows range searches to be performed in $O(\sqrt{n} + m)$ time, where n is the total number of characters in the world and m is the number of characters in the rectangle being searched, and characters to be inserted into or removed from the tree in $O(\log n)$ time[19, 21].

5.2 Client

The client required some form of graphical interface to aid in debugging and confirming that the server was correctly performing tasks such as collision detection and NPC pathfinding, but we did not want to waste time making the client particularly aesthetically pleasing, when the focus of the project was on creating a scalable server. For this reason, we chose to have the client run in a terminal and use the ncurses library for graphics[22], which allows simple text-based graphics to be set up and drawn easily. A screenshot of the client running is shown in Figure 9.

The ncurses library also provides functionality for handling keyboard input, but the client required slightly more flexibility than that offered by ncurses. In particular, ncurses cannot be used to detect specific 'key pressed' and 'key released' events, but instead detects when a key has been 'typed' (pressed and then released). We wanted the user



```
Connected!
Changed world
Changed world
finding path (0,31) -> (6,31)
finding path (6,31) -> (0,31)
finding path (0,31) -> (6,31)
finding path (6,31) -> (0,31)
finding path (0,31) -> (6,31)
finding path (6,31) -> (0,31)
```

Figure 9: A screenshot of the client running, showing a player ('C') and NPC ('N') in a small world.

to be able to move the player by holding a particular direction key, so we chose to use the Xlib library for keyboard input[23]. This library provides the necessary low-level keyboard support, while not requiring the creation of a separate X window (unlike other libraries such as the Simple and Fast Multimedia Library (SFML)[24]).

5.3 Bot

The bot reuses most of the client code, but instead of reading input from the keyboard with Xlib, it generates its own key press and release events to move the player appropriately. This allows it to control a player without any input from the user. The bot also does not require graphics of any form, so does not use the ncurses library.

6 Testing and results

To determine the performance of our game relative to that of commercial games, and to investigate the difference in performance between the server-to-peers and server-client architectures, extensive testing of the game was performed. In this section, the hardware used to run the server is listed, and these system specifications used to predict where the performance bottlenecks will occur. The results of the testing are then given, and their interpretations and implications discussed.

6.1 Server specifications

All testing of the game was performed with the server running on *compasaur*, a mid-range desktop computer running Debian GNU/Linux 6.0. Unfortunately we did not have access to server-grade hardware, and while this will have a detrimental effect on the absolute performance of the game, it should not greatly influence the relative performance of the server-to-peers and server-client architectures. The CPU used was an AMD Phenom II X4 945 processor, with four cores (only two of which were used by server processes) each clocked at 3 GHz. Memory available comprised a total of 64 KiB L1 cache per core, 2 MiB unified L2 cache, 24 MiB unified L3 cache, and 4 GiB DDR3 system RAM, clocked at 1600 MHz.

Two types of network were used when testing the server. First the server was run over the Internet, with an upstream connection speed measured at $100\,\mathrm{kB\,s^{-1}}$ (and $950\,\mathrm{kB\,s^{-1}}$ downstream). Later the server was run over a local area network (LAN), with symmetric speed of approximately $11\,\mathrm{MB\,s^{-1}}$.

6.2 Performance estimates

With this information in mind, some predictions can be made as to where the performance bottlenecks will occur in the server. Suppose there are n characters in the server, and n' of these are clients (so there are $n - n'$ NPCs). Furthermore, assume that about 1% of the world's population is visible to any character at any one time – this corresponds to $0.01n$ characters. This estimate should be accurate because the map used for testing has dimension 200×200, while each character can see others within the 20×20 square centred at their position.

The server sends each player the positions of all visible characters ten times per second. Storing the position of a single character requires six bytes (two bytes to identify the character, two bytes to store the row coordinate and two bytes to store the column coordinate), so storing the positions of all nearby characters requires around $6 \times 0.01n = 0.06n$ bytes. Each of these packets also contains ENet, UDP and IPv4 headers, the combined sizes of which are around thirty bytes[25, 26]. Therefore each character will be sent position updates of approximately $10 \times (0.06n + 30) = 0.6n + 300$ bytes per second. Thus the total bandwidth required to send position updates to all players is around $0.6nn' + 300n'$ bytes per second. Requests for NPC paths to be found are sent once per 1.5 seconds per NPC, and each of these requires ten bytes plus the thirty bytes of headers, so about $\frac{40(n-n')}{1.5} \approx 27(n-n')$ bytes are sent per second. Some basic algebra shows that with 400 NPCs, the $100\,\mathrm{kB\,s^{-1}}$ upload connection will be saturated when there are around 140 connected clients. If the server is to be run over the LAN with $11\,\mathrm{MB\,s^{-1}}$ upstream connection speed, this number could be as high as 4000.

The server download requirements do not pose any serious limitation. Only two types of packet are regularly received from clients: namely, direction changes and NPC paths. Assuming each path contains on average ten waypoints, it requires about twenty bytes to store an NPC path, plus thirty bytes of headers. These paths should be received about once per 1.5 s per NPC, which corresponds to a download rate of about $\frac{50\times(n-n')}{1.5} \approx 33(n-n')$ bytes per second. Direction changes are received approximately once per second per client, and each of these packets requires only one byte plus thirty bytes of headers, yielding a download rate of around $31n'$ bytes per second. Therefore the total download rate of the server should be approximately $33n - 2n'$ bytes per second. With 400 NPCs, the download speed of $950\,\mathrm{kB\,s^{-1}}$ should not become a limiting factor until about 30 000 clients have joined. With the $11\,\mathrm{MB\,s^{-1}}$ LAN connection this number will be even higher – close to 350 000.

To estimate the number of characters that can join before processor speed starts to limit performance, the amount of time required to perform the slowest operation in the game loop, namely collision detection, must be determined. Collision detection requires a range search to be performed in a small (4×4) rectangle around each character. It is shown in Appendix B that, when n is less than around 100 000, each of these range searches requires around $0.07\sqrt{n} \times 10^{-6}$ seconds on the server machine used. The range search must be performed for each of the n characters 100 times per second, that is, every 0.01 seconds, so the server will be at its maximum capacity when $0.01 = 0.07\sqrt{n}n \times 10^{-6}$, or $n = \left(\frac{10^4}{0.07}\right)^{2/3} \approx 2700$. Thus the server framerate should drop below 100 FPS when there are around 2700 characters in the server. With 400 NPCs, this corresponds to 2300 connected clients. Note that this is probably an overestimate of the number of clients required to overload the processor, because only the slowest operation of the game loop, collision detection, was taken into account, even though other operations such as maintaining the k-d tree are almost as slow.

If the server is forced to perform NPC pathfinding itself (without allocating this work to connected clients), this becomes the most CPU-intensive operation of the game loop even though it occurs relatively infrequently (once per NPC per 1.5 s). The breadth-first search used for finding a path of length l requires collision detection to be performed at up to l^2 different positions on the map, so this operation takes $l^2 \times 0.07\sqrt{n} \times 10^{-6}$ seconds. Hence, if l is around 100 on average, finding a single path requires approximately $0.07\sqrt{n}\times 10^{-2}$ seconds. With 400 NPCs, this operation must be performed approximately

$\frac{400}{1.5} \approx 270$ times per second, or once every 3.7×10^{-3} seconds. The server will thus be at its maximum capacity when $0.07\sqrt{n} \times 10^{-2} = 3.7 \times 10^{-3}$, or $n = \left(\frac{3.7\times10^{-3}}{0.07\times10^{-2}}\right)^2 \approx 28$. But with 400 NPCs, n must be at least 400, so this result suggests that the server will not even be able to cope with all of the NPCs, let alone any connected clients. However, this derivation assumed that paths had to be found for all NPCs, but in fact the server only calculates paths if there is a client nearby (to ensure a fair comparison with the server-to-peers implementation, which only requests paths for NPCs which are close to a connected client). Hence, despite this estimate, the server would still be able to handle a small number of clients, because there would be very few paths to find. It is clear, however, that the capacity of the server-client implementation should be far less than that of the server-to-peers.

Our server is likely to encounter memory bottlenecks long after already becoming restricted by processing and networking limitations. By far the most significant memory requirement is the allocation of 20 kB per player (used to store data packets queued to be sent to clients), which will not fill the 4 GiB available until 200 000 clients are connected.

The order in which bottlenecks will be encountered during testing depends on the manner of the testing. If the server is run over a home Internet connection, the upload speed will become the first limiting factor, after around 140 clients have joined. On the other hand, if the server is run over a LAN or other high-bandwidth network connection, the processor speed is predicted to be the first bottleneck encountered, when approximately 2300 clients have connected.

6.3 Testing

During each test, at least one human client joined the server to ensure that collision detection, NPC pathfinding and bot AI were all working correctly, and that there was no perceivable network or server lag. The first three of these could be determined objectively just from studying the movement of NPCs and bots. Detecting lag was a more subjective process, and meant determining that all entities in the game world appeared to be moving smoothly and at constant speed. If the network became saturated, packets would be lost (since UDP was used), and the client would receive position updates only sporadically, giving the appearance of characters 'warping' from one location to another, no longer moving at constant speed. If the server's CPU became overloaded, the game framerate would drop to an unacceptably low level, causing client updates to be sent infrequently, making characters move in a visibly jerky fashion.

The first test was to connect as many bots as possible over the Internet, while recording the FPS and bandwidth use of the server. For this test the server had 400 NPCs, because we thought this would be a reasonable estimate of the number of NPCs in a real MMOG. Bots were connected to the server ten at a time until either they started being disconnected, or could no longer connect at all, which indicated that the server was unable to maintain these extra players. A graph of upload and download speed against number of characters is shown in Figure 10, while a graph of FPS against number of characters is shown in Figure 11. The error bars in these figures, and all others contained herein, show the standard error in the values (for each particular character count, the server was left running long enough to collect at least ten data points; a script was written to group together all data points with the same character count, then average them and calculate the standard error).

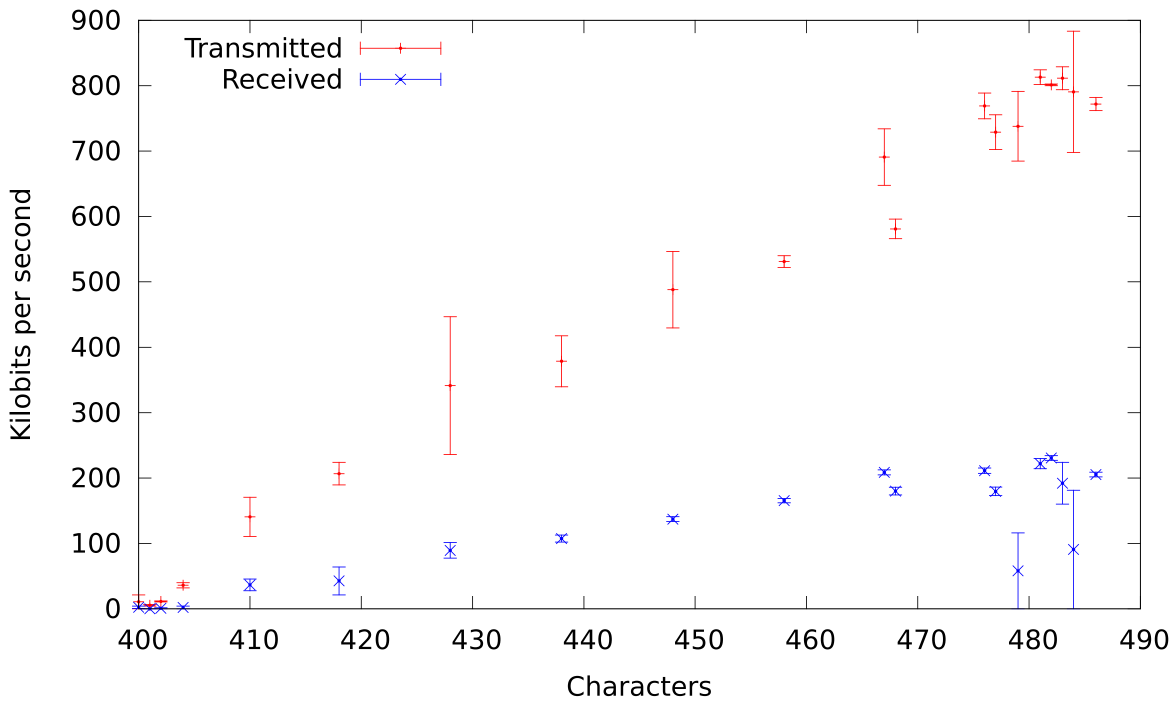


Figure 10: Graph of upload and download speed against number of characters for bots connected over the Internet with 400 NPCs.

Figure 10 shows that the 100 kB s^{-1} upload speed limit is reached when there around 480 characters, or 80 players. Note that this is lower than the estimate of 140 players made in Section 6.2, but this can be attributed to the frequent packets sent automatically between the server and each client by ENet to transfer basic information, such as latencies, and to ensure that the connection is still active. Figure 11 shows that the framerate of the server dropped as more players joined the server, until around sixty players had joined. The rise in framerate occurring beyond this point can be explained by considering the behaviour of the game as the upload connection became saturated. When this occurred, packets sent to clients requesting paths for NPCs would often be lost, so these NPCs

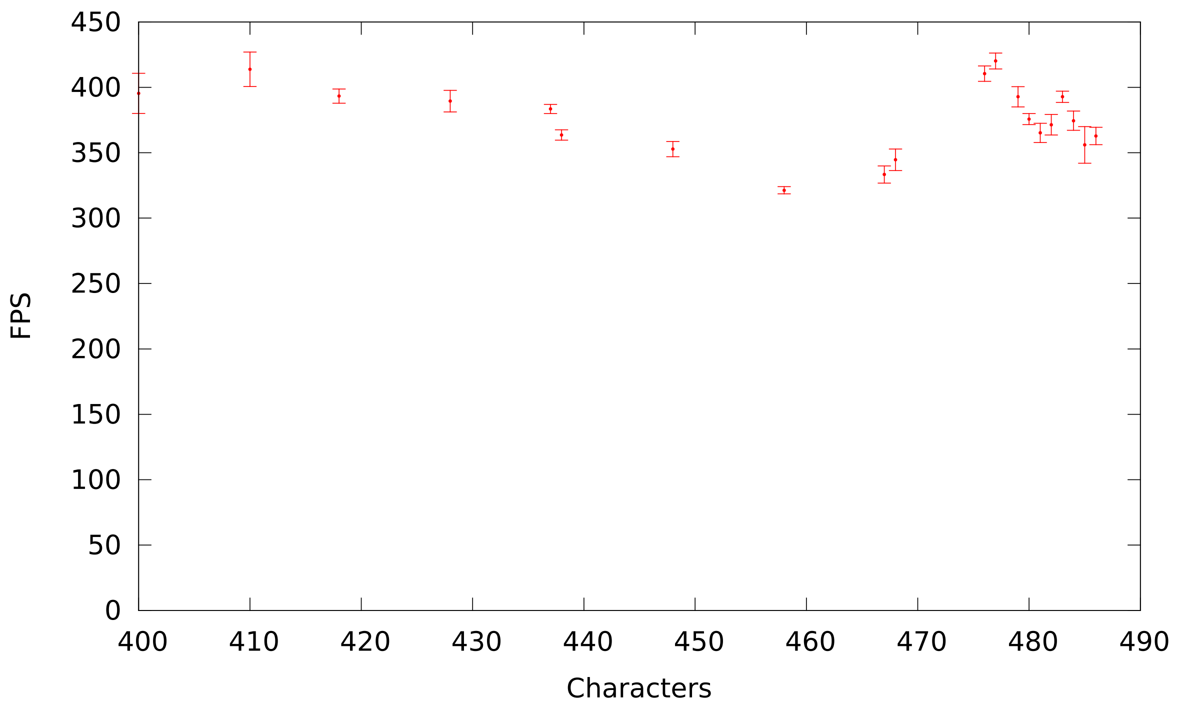


Figure 11: Graph of FPS against number of characters for bots connected over the Internet with 400 NPCs.

would not move at all (because they had no paths). This lack of movement meant that less collision detection and maintenance of the k-d tree had to be performed, so the load on the server decreased. These graphs demonstrate that the bottleneck for this test was the upload connection, as expected from the estimates in Section 6.2. Unfortunately the relatively small number of data points in the graphs, and the large fluctuations even in nearby values, make it is difficult to extrapolate these trends in order to determine when the framerate of the server would drop below 100 FPS. However, Figure 10 suggests that the download requirements of the server will always be significantly less than the upload demands, implying our game server will never be limited by the download speed of the connection.

Further testing was performed by connecting bots over a LAN with much faster upload and download connections. This test was more realistic than the first, because a real game server would almost certainly have a significantly faster Internet connection than a home computer. As before, 400 NPCs were used, and bots were connected to the server ten at a time. For this test we were unable to saturate either the upload or download connections, or cause the framerate of the server to drop below 100 FPS, so results were only taken until 600 bots had joined (bringing the character count to 1000). These results are shown in Figures 12 and 13. Notice that Figure 12 is consistent with the result from the first test, in that the amount of data transmitted is much more than that received.

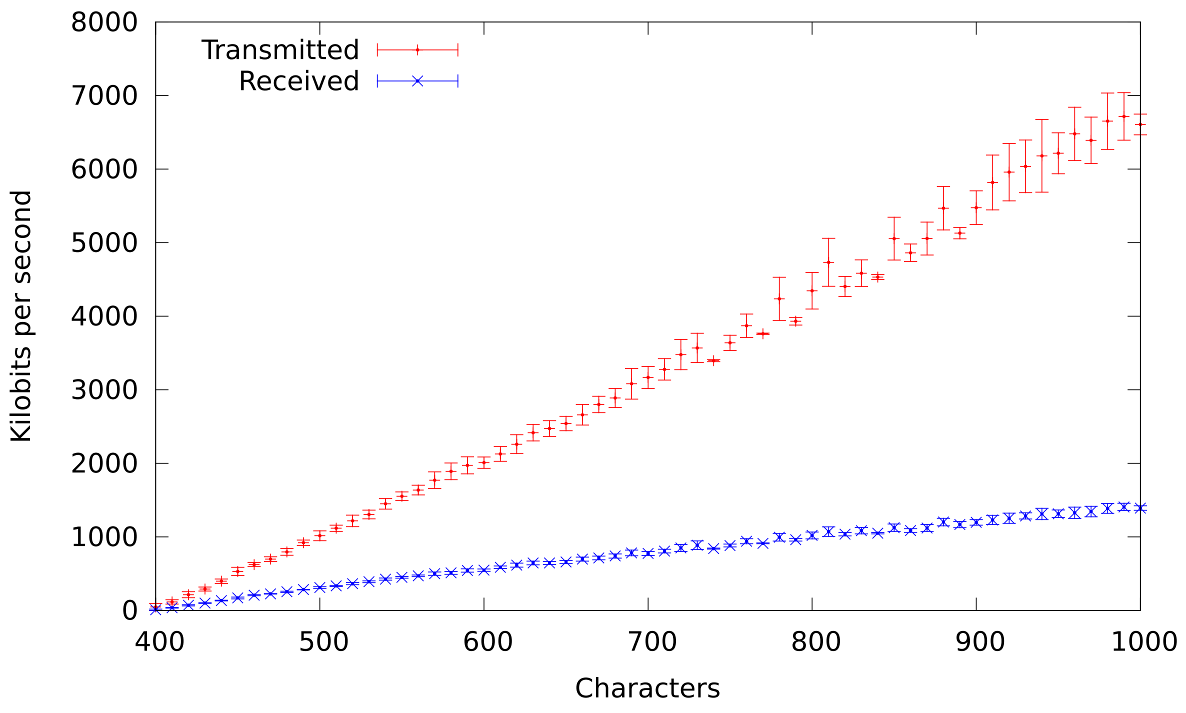


Figure 12: Graph of upload and download speed against number of characters for bots connected over a LAN with 400 NPCs.

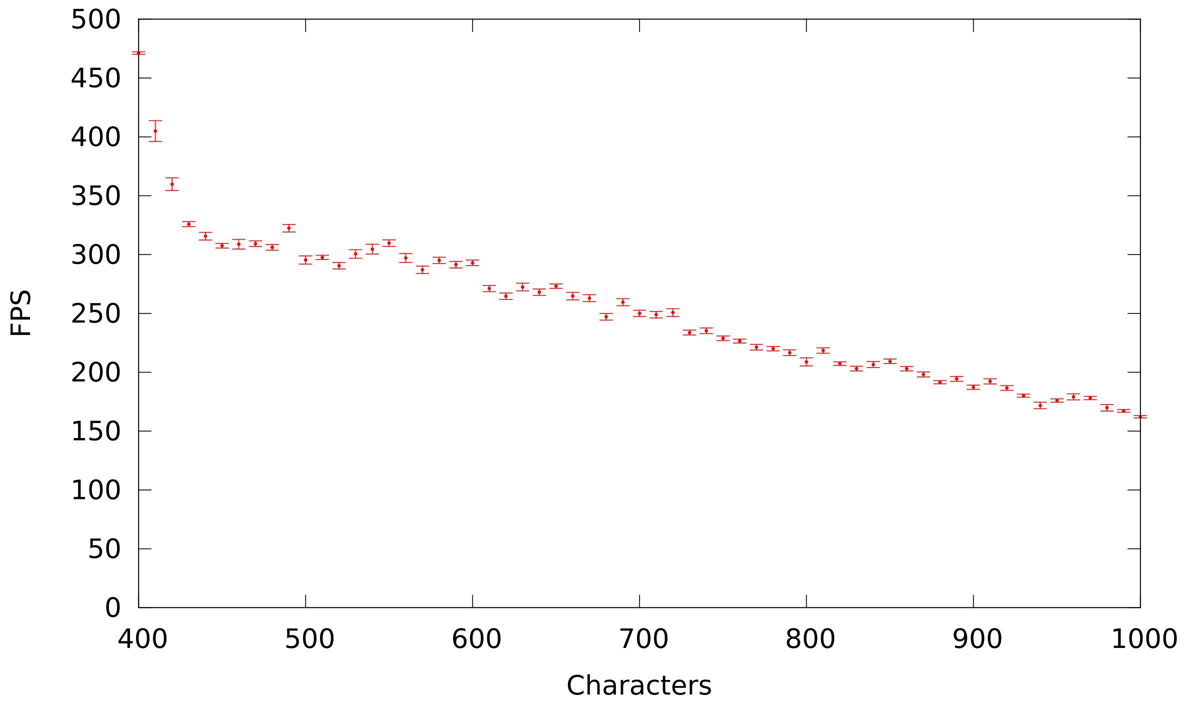


Figure 13: Graph of FPS against number of characters for bots connected over a LAN with 400 NPCs.

Quadratic and linear trendlines were fitted to the respective plots of transmitted data against number of characters and FPS against number of characters in Figures 12 and 13, as shown in Figures 14 and 15. A quadratic trendline was fitted to the plot of transmitted data against number of characters because the analysis is Section 6.2 suggests that this relationship should be parabolic. While a linear relationship was not predicted between FPS and number of characters, it is apparent from Figure 13 that the trend is fairly linear when the number of characters is greater than around 420. The data prior to this point were therefore ignored when fitting the trendline.

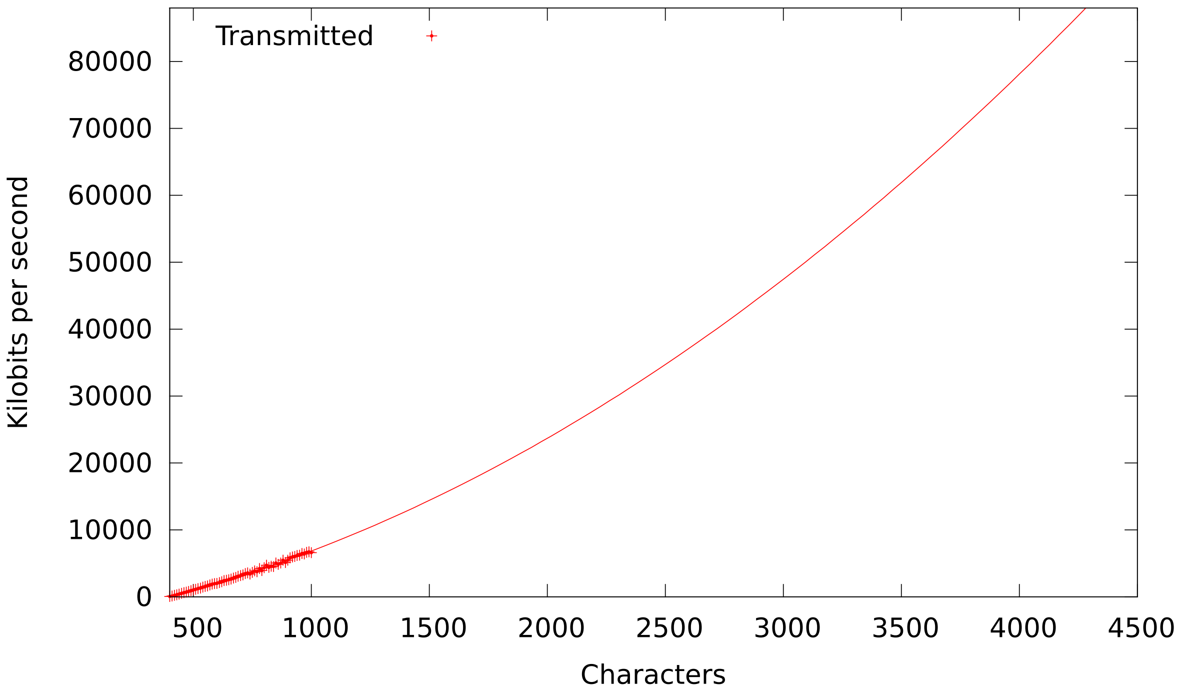


Figure 14: Quadratic trendline for plot of upload speed against number of characters for bots connected over a LAN with 400 NPCs.

Figure 14 suggests that the 11 $\mathrm{MB\,s^{-1}}$ upload connection will not limit the capacity of the server until there are around 4300 characters, or 3900 connected clients. This is consistent with the estimate of 4000 clients calculated in Section 6.2, although it is difficult to be sure of the accuracy of this extrapolation without more data. Figure 15 suggests that the server framerate will drop below 100 FPS when there are about 1200 characters in the server, or 800 connected clients. This implies that, with the type of Internet connection one would expect of a real dedicated game server, the capacity of our server will be limited by the processor speed (if a framerate of 100 FPS is to be maintained), as predicted in Section 6.2. While the capacity of the server was lower than the predicted value of 2300, this is almost certainly due to the fact that only the slowest operation of the game loop, collision detection, was taken into account when estimating

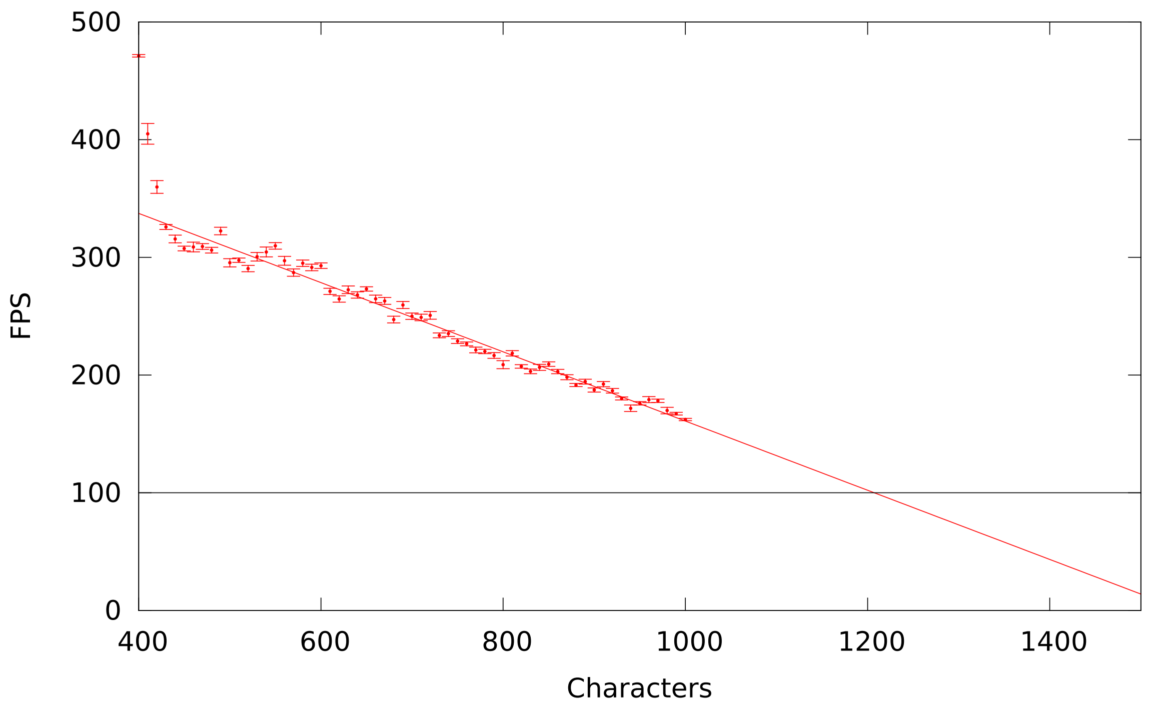


Figure 15: Linear trendline for plot of FPS against number of characters for bots connected over a LAN with 400 NPCs.

the bound placed on the capacity by the processor speed, even though other operations (such as maintaining the k-d tree) are almost as time-consuming.

Recall from Section 3 that the popular commercial games *EVE Online* and *World of Warcraft* could both handle on the order of thousands of players. Our server's capacity is marginally lower than this, but is still of a comparable order. Considering the fact that our server's target framerate was 100 FPS, which is almost certainly much higher than that of a typical MMOG[15], the performance achieved is very good.

The final step in the testing was to compare the relative performance of the server-to-peers and server-client architectures for our game. The server was modified to perform NPC pathfinding itself, thus converting the system from server-to-peers to server-client. Note that the server used the same breadth-first search as the client for this pathfinding, ensuring a fair comparison between the two architectures. With 400 NPCs, not even twenty clients could join before the server framerate dropped to under one frame per second. This is pictured in Figure 16. The framerate fluctuated a lot initially, since it was heavily dependent on how many NPCs requested paths, which depended on how many were visible to the connected bots, which, since the number of bots was so small, was essentially random. From the time that the fourth bot joined the server, the frequency of NPC path requests remained high enough to keep the server framerate below 10 FPS. By the addition of the thirteenth bot, the average server framerate stayed below 1 FPS.

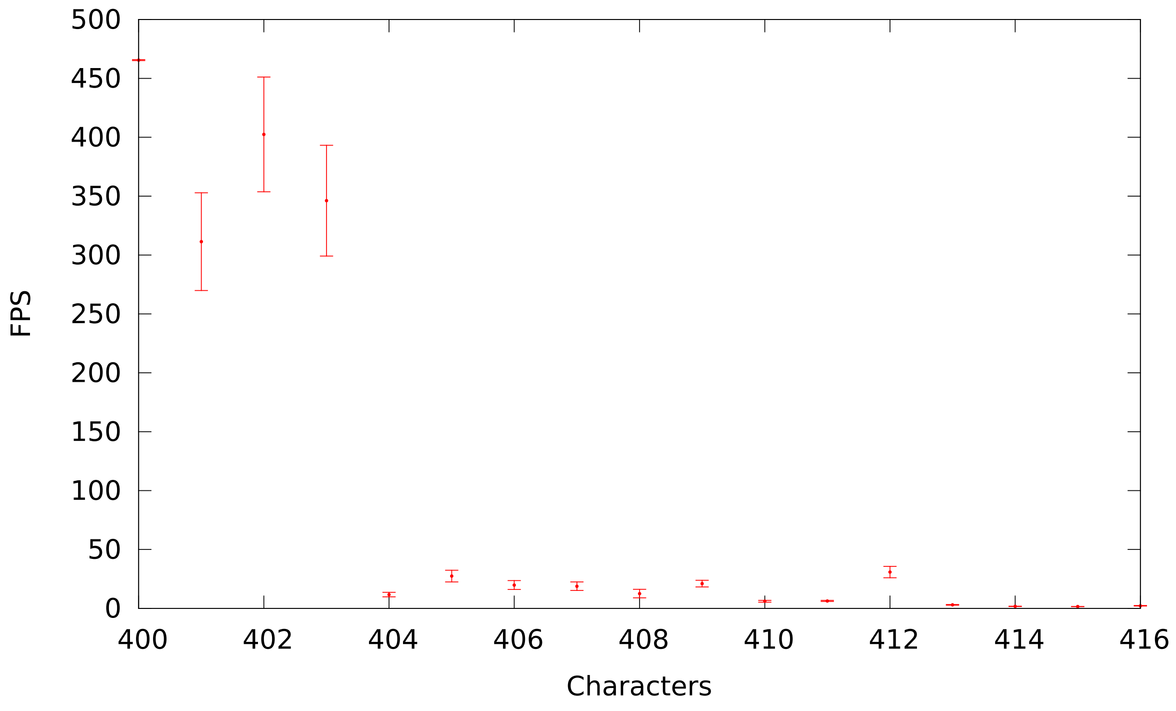


Figure 16: Framerate of a centralised server with 400 NPCs.

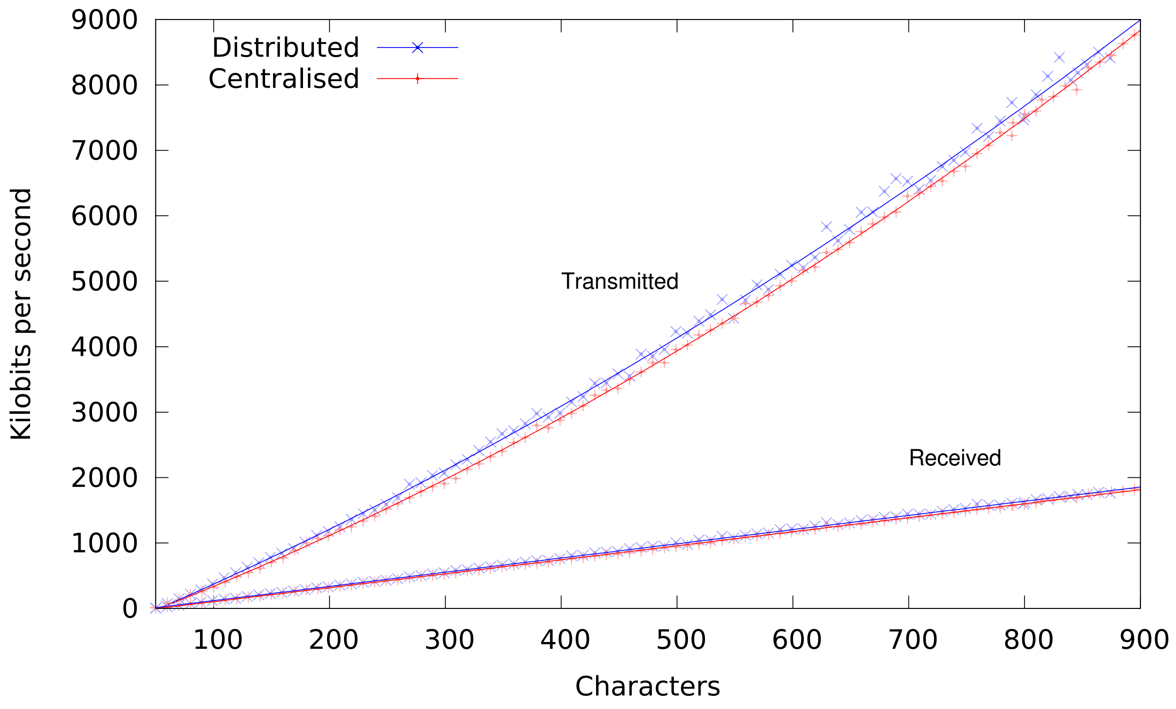


Figure 17: Data transfer rate comparison between distributed and centralised designs.

While it is clear that the use of the server-to-peers architecture provides significant processing performance benefits over server-client in terms of framerate, there are too few data points to determine the relative network performance of the two designs. The number of NPCs was lowered to 50 to ensure that several hundred bots would be able to join both the server-to-peers and server-client versions of the game server. Figure 17 compares the amounts of data transmitted and received for each of these servers. It is clear that the difference is negligible: over the range plotted, the difference in data transmission rate is no more than about 5%, and the rates of received data are almost indistinguishable.

The server-to-peers design provides great overall benefit, then. While the additional network overhead is minimal, the server load is greatly reduced. This architecture therefore vastly increases server capacity, at a negligible networking cost.

7 Conclusions

In this project we have successfully designed and implemented a simple massively multiplayer online game using a server-to-peers architecture. The performance of the game server was poor when clients connected to the server over the Internet, with only around 80 players being able to connect. However, for a dedicated game server one would expect a significantly faster Internet connection than that used for testing in this project. When clients connected to the server over a LAN, with a much faster upload connection, the server was predicted to be capable of handling around 800 players before the framerate dropped to below 100 FPS, which compares very well to the performance of popular commercial games such as *EVE Online* and *World of Warcraft*.

We have also demonstrated that the server-to-peers architecture can give significant performance gains over a server-client design, dramatically reducing the processing requirements of the server and causing only a minimal increase in network traffic. While this performance gain depends heavily on the tasks chosen to distribute to peers, we believe that server-to-peers techniques could be applied beneficially to a wide range of MMOGs which would traditionally be based on the server-client architecture. However, the applications of the server-to-peers design are not restricted to server-client games: this design also has connections with two active areas of contemporary research, the previously-mentioned mirrored-server[8] and peer-to-peers designs[7].

The existing partitioning into Worlds (Figure 6) of our design lends itself naturally to using the mirrored-server architecture. It should therefore be a simple task to unify these two designs, sharing the advantages of both. The resulting architecture would possess the great CPU scalability of the server-to-peers design, as well as the strong network scalability and lower average latencies of the mirrored-server design (as discussed in Section 2.3.2).

There are three major ways (discussed in Section 2.3.1) that a peer-to-peer design is superior to a central server design, and our architecture can be made to share all of these advantages. First, the main advantage of peer-to-peer over centralised design is the removal of the central server bottleneck, which our results have shown occurs in the CPU long before the network. However, our design achieves the goal of greatly reducing CPU load on the server, making it possible to run a central server while retaining high scalability, matching this advantage of the peer-to-peer design. The second major advantage the peer-to-peer design has over the centralised design is decreased latency between peers. As discussed above, our design can be combined with mirrored-server design to share this advantage, too. Finally, the peer-to-peer design eliminates the large central network traffic load, but this turns out to be a largely irrelevant consideration. Research suggests that the overall network load in a peer-to-peer design is similar to that of a traditional centralised design[8], which our results indicate is only very slightly less than that of the server-to-peers layout. Our bottleneck predictions suggest that a server with a high-quality network connection will not become network-bound until many thousands[2] of users are connected simultaneously, so for practical purposes the removal of centralised network load is not an important advantage of the peer-to-peer design. (Even if it were, augmenting our server design with the mirrored-server layout would allow indefinite network scalability.)

Our server-to-peers architecture matches the strengths of the peer-to-peers design, while avoiding some of its weaknesses. The retention of a central server and official game state can greatly reduce the potential for cheating, since it is not possible for a client to change the game state however he or she sees fit. Furthermore, having a central server gives the owners of a game complete control over its state, which may be desirable for some commercial products. Overall, by combining the strengths of server-client and peer-to-peers architectures, and largely avoiding their weaknesses, our design allows great scalability and presents a compelling case for its use in modern game architecture.

[2]Quadratic extrapolation predicts over 16 000 users would saturate a gigabit connection.

A Code

The source code for the implementation of our project is available at

```
http://cs.anu.edu.au/people/Eric.McCreath/students/GillSlatyer2011/code.tgz
```

and is distributed under the terms of the zlib license:

Full documentation is available at

```
http://cs.anu.edu.au/people/Eric.McCreath/students/GillSlatyer2011/doc/classes
```

or can be generated from the code using Doxygen.

B Benchmarking libkdtree++

The time complexity of libkdtree++'s range search for a two-dimensional tree is $O(\sqrt{n} + m)$, where n is the total number of characters and m is the number of characters in the rectangle being searched[19]. For the purposes of collision detection in our game, the characters were placed in a 200×200 area and the range search required locating characters within a 4×4 square. Assuming characters are somewhat uniformly distributed throughout the 200×200 area, it follows that $m \approx \frac{4\times4}{200\times200} n = 0.0004n$. Thus when n is less than around 100 000, $m \ll \sqrt{n}$, so the complexity of the range search is approximately

$O(\sqrt{n})$. In order to make accurate predictions of the real-world performance of this range search, however, more information is required than just the time complexity.

For this reason, a program was written to generate random two-dimensional trees and perform range searches on them in the same way that a game server might for collision detection. This program is contained in the 'kdtree' directory of the code distribution. The range searches were timed (with each particular value of n measured multiple times in an attempt to reduce error), and the data plotted in Figure 18. The results indicate that the running time is indeed $O(\sqrt{n})$ for the cases tested. Therefore a function $f(n) = b\sqrt{n}$ was fitted to the data, as pictured.

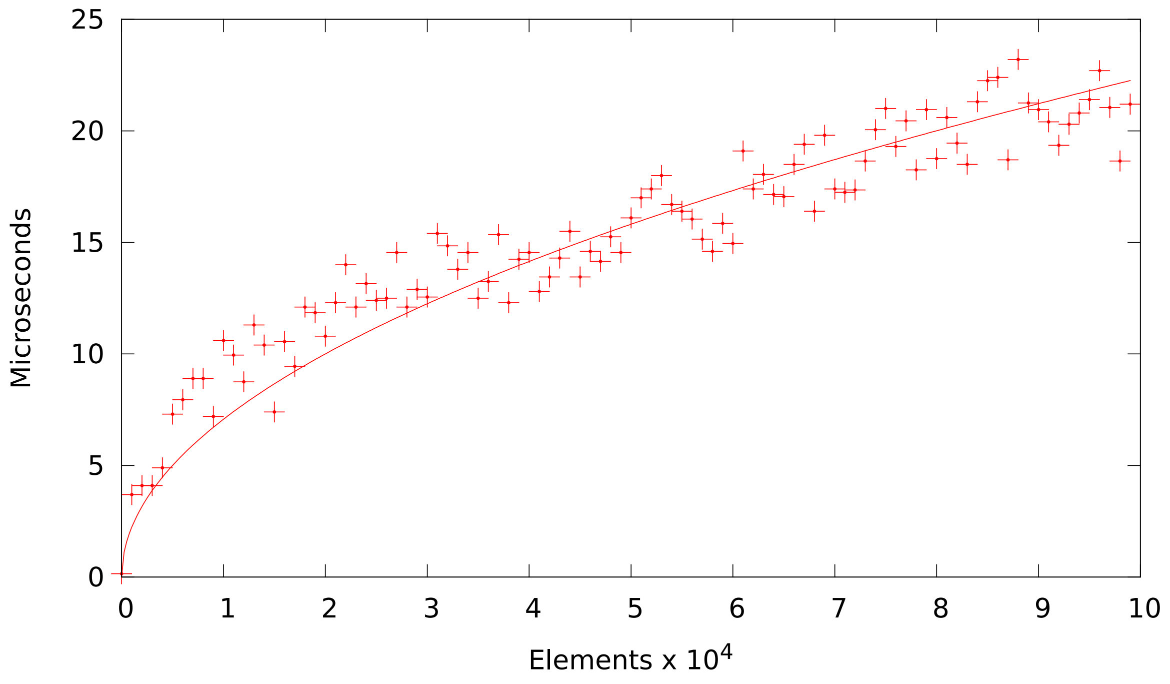


Figure 18: Dependence of range search running time on character count.

It was found that $b \approx 0.07$. That is, running on *compasaur*, each range search required $0.07\sqrt{n}$ microseconds.

Bibliography

[1] Reuters. Factbox: A look at the $65 billion video games industry. http://uk.reuters.com/article/2011/06/06/us-videogames-factbox-idUKTRE75552I20110606, 2011. [Online; accessed 11-October-2011].

[2] Lars Brandle. IFPI report: Global music trade down 8.4%. http://www.themusicnetwork.com/music-news/industry/2011/04/04/ifpi-report-global-music-trade-down-8-4/, 2011. [Online; accessed 11-October-2011].

[3] Nash Information Services, LLC. US movie market summary 1995 to 2011. http://www.the-numbers.com/market/, 2011. [Online; accessed 11-October-2011].

[4] Greg Thom. Two-thirds of Australians play video games. http://www.news.com.au/technology/most-of-us-are-gamers-new-figures-show/story-e6frfro0-1225713891208, 2009. [Online; accessed 11-October-2011].

[5] entertainment software association. 2011 essential facts about the computer and video game industry. http://www.theesa.com/facts/pdfs/ESA_EF_2011.pdf, 2011. [Online; accessed 11-October-2011].

[6] Ibe Van Geel. Subscriptions and Active Accounts with a peak between 1.000.000 and 12.000.000. http://mmodata.net/, 2011. [Online; accessed 10-November-2011].

[7] Björn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1. IEEE, 2004.

[8] Eric Cronin, Burton Filstrup, and Anthony Kurc. A distributed multiplayer game server system. In *University of Michigan*, 2001.

[9] Christophe Diot and Laurent Gautier. A distributed architecture for multiplayer interactive applications on the Internet. *IEEE Network*, 13(4):6–15, 1999.

[10] Paul Bettner and Mark Terrano. 1500 Archers on a 28.8: Network programming in Age of Empires and beyond. *Presented at GDC2001*, 2001.

[11] Ashwin Bharambe, Jeff Pang, and Srinivasan Seshan. A distributed architecture for interactive multiplayer games. Technical report, School of Computer Science, Carnegie Mellon University, 2005.

[12] Emmanuel Lety, Laurent Gautier, and Christophe Diot. Mimaze, a 3D multi-player game on the Internet. In *Proc. of 4th International Conference on VSMM (Virtual Systems and MultiMedia), Gifu, Japan.* CiteSeer, 1998.

[13] Michael Thompson. Modern Warefare 2: the case for the dedicated server. http://arstechnica.com/gaming/news/2009/10/modern-warfare-2-the-case-for-the-dedicated-server.ars, 2009. [Online; accessed 8-November-2011].

[14] EVE Insider Dev Blog. my node was equipped with the following.... http://www.eveonline.com/devblog.asp?a=blog&bid=589, 2006. [Online; accessed 11-October-2011].

[15] Erlendur Þorsteinsson. New Dev Blog: Fixing Lag: Drakes of Destiny (Part 1). http://www.eveonline.com/ingameboard.asp?a=topic&threadID=1429179&page=3, 2010. [Online; accessed 14-November-2011].

[16] MMORPG.com Discussion Forums. How many players can a full server hold? http://www.mpog.com/discussion2.cfm/post/3176164, 2009. [Online; accessed 11-October-2011].

[17] Forums – World of Warcraft. How many people does a WoW server hold? http://us.battle.net/wow/en/forum/topic/2140264926, 2011. [Online; accessed 11-October-2011].

[18] Jason Gregory. *Game Engine Architecture*. AK Peters, 2009.

[19] Steven Skiena. *The Algorithm Design Manual*. Springer, London, 2nd edition, 2008.

[20] Lee Salzman. Enet: Features and architecture. http://enet.bespin.org/Features.html, 2011. [Online; accessed 16-October-2011].

[21] Sylvain Bougerel Martin Krafft, Paul Harris. libkdtree++. http://libkdtree.alioth.debian.org/, 2010. [Online; accessed 4-November-2011].

[22] Free Software Foundation, Inc. Announcing ncurses 5.9. http://www.gnu.org/s/ncurses/, 2011. [Online; accessed 16-October-2011].

[23] Christophe Tronche. The Xlib manual. http://tronche.com/gui/x/xlib/, 2005. [Online; accessed 2-August-2011].

[24] Laurent Gomila. Simple and Fast Multimedia Library. http://www.sfml-dev.org/, 2011. [Online; accessed 16-October-2011].

[25] Network Sorcery. User datagram protocol. http://www.networksorcery.com/enp/protocol/udp.htm, 2011. [Online; accessed 11-November-2011].

[26] ENet-Discuss. Enet header size. http://lists.cubik.org/pipermail/enet-discuss/2009-January/001017.html, 2009. [Online; accessed 11-November-2011].