Bournemouth University

Faculty of Science and Technology

BSc (Hons) Games Programming

May 2019

Persistent Data Synchronisation in Peer-To-Peer Network Architectures

S4917575

Charlie Long

DISSERTATION DECLARATION

This Dissertation/Project Report is submitted in partial fulfilment of the requirements for an honour's degree at Bournemouth University. I declare that this Dissertation/ Project Report is my own work and that it does not contravene any academic offence as specified in the University's regulations.

Retention

I agree that, should the University wish to retain it for reference purposes, a copy of my Dissertation/Project Report may be held by Bournemouth University normally for a period of 3 academic years. I understand that my Dissertation/Project Report may be destroyed once the retention period has expired. I am also aware that the University does not guarantee to retain this Dissertation/Project Report for any length of time (if at all) and that I have been advised to retain a copy for my future reference.

Confidentiality

I confirm that this Dissertation/Project Report does not contain information of a commercial or confidential nature or include personal information other than that which would normally be in the public domain unless the relevant permissions have been obtained. In particular, any information which identifies a particular individual's religious or political beliefs, information relating to their health, ethnicity, criminal history or personal life has been anonymised unless permission for its publication has been granted from the person to whom it relates.

Copyright

The copyright for this dissertation remains with me.

Requests for Information

I agree that this Dissertation/Project Report may be made available as the result of a request for information under the Freedom of Information Act.

Signed:

Name: Charlie Long

Date: 16th May 2019

Programme: BSc GP

contents

chapter 1 : abstract

1.1 scenario

One of the primary costs associated with developing multi-player online video games is that of dedicated server hosting. Ranging from short-lived competitive or co-op game instances to massive multi-instanced virtual worlds, these servers require both large amounts of computational power and a high data throughput. The onus of providing these servers as a service is often on the developers or publishers, resulting in a lack of player control over the lifetime of the game.

1.2 proposition

This project aims to develop and test a method for enabling the distributed processing and hosting of virtual game worlds in which all clients act as equal nodes in the network. Eliminating as many aspects of the centralised network architecture is important; complete player independence from any third party hosted tools is a major factor in the development of this project. The result will be compared to a more traditional network design and the viability of the network architecture will be evaluated. An attempt will also be made to introduce data persistence and load balancing to the resulting partial mesh network. The deliverable will be a module written in C++, integrated loosely with a simple, custom, 3D capable game engine to display the library's capabilities.

chapter 2 : introduction, aims & objectives

2.1 background

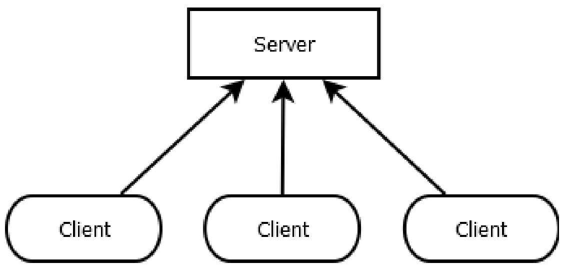
Not long after the introduction of the very first commercially available computer games came the internet. This was not the invention of multiplayer video game interaction, which happened early in the arcade era, with two or more players constrained to a single machine with multiple inputs. This was a progression from local multiplayer to a more connected audience. Soon came ARPANET, or the Advanced Research Projects Agency Network, and the introduction of TCP/IP, a communications model that set the standards for local network data transmission. The internet as we know it was born.

Since then it has expanded massively. According to the Internet World Stats, March 2019 [1], 56.1% of the world's population has access to the internet, including up to 81% of the developed world. With such a large audience accessing the internet on devices ranging from mobile phones to desktop computers the potential market value is incomprehensible. The internet has become a tool for communication, research, media consumption and entertainment. A number of these uses are relatively latency agnostic, but for those that aren't, and with such large distances separating many users of the internet, many different solutions for reducing the impact of high latency have been suggested.

Multiplayer games are one area in which time is an important factor. Improving latency and data throughput without sacrificing reliability of information transfer is a difficult task but is essential for the development of complex real time virtual worlds. Not only must all interactions occur as instantaneously as possible, but there must be little, or no, chance of data loss for all users to share a common view of the world. Multiple different network architectures have been developed to perform this task, each with differing use cases.

2.2 client-server architecture

Traditional video game networks are single-server many-client constructs. A central server is hosted which can be connected to by any number of clients. This architecture has the advantage that a single authority orders events, resolves conflicts in the simulation, acts as a central repository for data, and is easy to secure, GauthierDickey, C, 2004 [2]. This server is often distributed and run as a separate tool to the client software. Not only does this result in more work for developers, to keep both server and client programs up to date and compatible, but also the server instance has to be hosted temporarily, or in some cases indefinitely, by either the developers of the game or by the players themselves. This latter option is often referred to as a community hosted server.



When it comes to large numbers of players in the hundreds or even thousands in one game instance a single server may no longer cut it. For this situation several solutions involving linked server machines and virtual spatial partitioning have been developed to allocate devices to sections of the game. Load balancing in this scenario involves assigning more cores to certain segments of the world with higher numbers of entities. The virtual location-based partitioning has a better scalability in that each server only needs to manage a portion of the whole world, Ta Nguyen, et al, 2003 [3].

2.2.1 dedicated versus community

Many developers do not opt to offer these server programs to the community, such as in Grand Theft Auto Five, because the design of the game requires more control over how the game is interacted with. Widely distributed server software that allows any user to host a game often is quickly reverse-engineered or modified to influence the gameplay or virtual world. In situations in which advantages can be gained through micro-transactions such modifications can negate the need to pay, resulting in a loss of profit, but server software modification by users can also be advantageous. In the case of games such as Minecraft, especially the Java based client, user modification is a major selling point offering many times more hours of gameplay than the vanilla client can. The distribution of community focused server software can also positively impact the game's lifetime, even past the lifetime of the company that produced it.

2.2.2 server authoritative

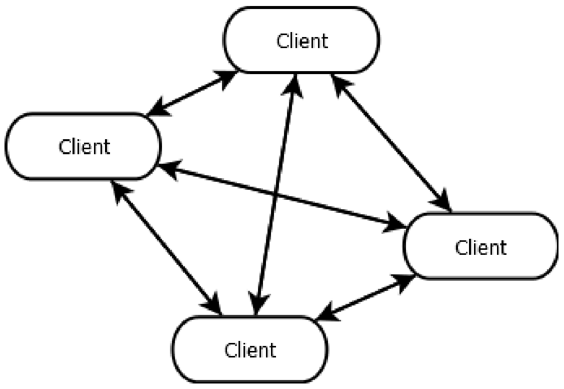
Whether developer hosted or community hosted, in this form of network the server is nearly always authoritative. This means that information received from a client is not regarded as truth but checked against peer inputs and other factors. If a client is found to be sending inputs which do not seem legitimate and might give the player an unfair advantage over other players, they can be singled out and either reprimanded or removed from the network. The server is much less likely to be modified in order to give false information than a client is, which improves the security of this network architecture significantly over alternatives.

Methods still exist of "tricking" the game server into accepting errant behaviour as legitimate inputs. A modified client might flush a number of messages indicating that it had moved to a certain position inside the virtual world in order to gain an advantage, and this sudden influx of messages might be interpreted by the server as delayed packets due to a low-quality network connection. Because it is possible for a valid client on a poor network connection to generate indistinguishable behaviour, this cheat is not in the class that most verifiers detect, states Bethea, D, et al, 2011 [4].

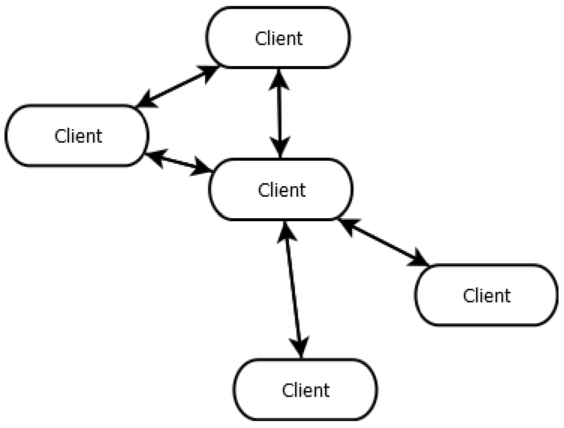
2.3 mesh-network architecture

An alternative network structure is that of a mesh network. In a mesh network there is no central host; all peers are connected directly to each other or to another peer that is connected to the network. Peer-to-peer architectures, due to their distributed and collaborative nature, have low infrastructure costs and can achieve high scalability. They can also achieve fast response times by creating direct connections between players, Yahyavi, A, 2013 [5].

In a full mesh network each and every peer is connected to each other, creating a large number of redundant connections, but ensuring connection at all times to every other member of the network. In a partial mesh network, clients may not be connected to all other peers, but connected to a single peer which forwards packets to it. Although this latter option reduces the number of connections that each peer must handle at any one time, the amount of time processing the data transmitted is increased as packets have to be received, parsed, and forwarded multiple times. A single dropped connection can isolate one or more peers if no redundant connections are made. With the right data a reconnect attempt can be made but this can still result in missed packets during the downtime.



Full Mesh Network (Above) Vs Partial Mesh Network (Below)



2.3.1 mesh-network potential pitfalls

The first pitfall the one might encounter with this architecture is the issue of authority. As there is no one central device that is not a user there is no viable peer that can be selected as an authoritative member of the network. Every peer has the same level of trust, and so the network is open to exploitation. Peer-to-peer game architectures provide better scaling, but open the game to additional cheating, since players are responsible for distributing events and storing state, Jardine, J, et al, 2008 [6]. One potential solution to this problem is to provide versions of the client which do not act as in-game entities but continue to interact with the network as is normal. These clients will be able to compare and analyse inputs from all peers connected to the network and will be able to single out potential malicious devices. Other solutions include industry standard methods of anti-tampering, such as hashing game files and comparing results with expected hashes or those of peers in the network. Unfortunately, a number of these methods are no longer applicable with no central authority.

A second hurdle which is that of a problem with handling connections between devices on Local Area Network (LAN) versus devices on Wide Area Network (WAN). Any users that wish to be accessible from outside of their local network must port-forward the connection on their router. The only way to test if this port forward is active is to connect to the device from outside of the network, which complicates the process considerably. Listing device accessibility in the network image is a must, as this information is required when picking a new target for reconnecting.

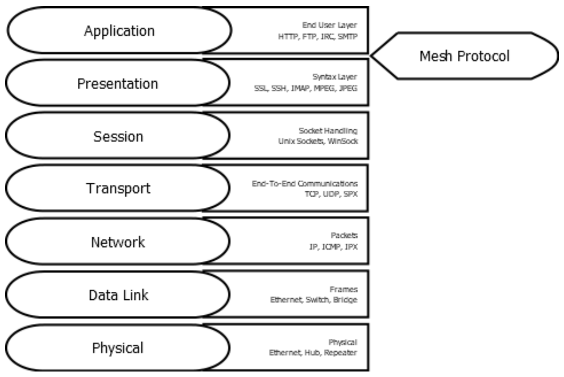
Another major problem occurs when the root, or first, node disconnects from the network.

As this is the first node to join the network it should be at the top of the hierarchy. If it has multiple children, a race condition of sorts will be created, as each of the children search for a new host. In particularly unfortunate circumstances each peer might attempt to connect to a node that is lower in the hierarchy of another, creating a loop in which messages might get caught indefinitely. The first solution that comes to mind is for each host to elect a child to take its position should it disconnect, or to rank children by the order in which position inheritance should occur.

2.4 project goals & deliverable

The target of this project is to construct a partial mesh network without excessive redundant connections, instead including network imaging that allows for reconnect attempts to be made. This will reduce the number of connections each peer has to handle but might exacerbate unreliable aspects of the network. As this is only a demonstration the aim is not to be completely reliable, but to prove that this structure is fast enough, yet light enough, to run on user machines without impacting performance unacceptably.

This network imaging will require the development of a protocol that will run on top of TCP or UDP that can differentiate game data and network packets. These network packets will be utilised in the construction of a network tree on each peer that will provide the required information for message targeting and autonomous reconnect handling. This network tree will have to be persistent across the network, meaning that new connections must be able to receive the serialised network image, parse it, and then add themselves to the network, broadcasting this event to the rest of the network. This is a basic form of data persistence in a mesh network; a concept which will also be employed in the game state syncing itself. This network image could potentially also be used to reduce the amount of data bouncing around the network. If each packet includes target and source identification it can be forwarded only to peers which are in the direct path to the target peer. In this way an advantage can be gained over traditional network structures.



Using this constructed network image, the library will handle reconnect and load balancing autonomously. Information on an active peer's address, port forward status, and current load can be used to pick an appropriate target to attempt reconnect. Any children that the disconnected peer owns will have to be serialised and sent to the new host if connection is regained, eliminating the need for all disconnected peers in the hierarchy to reconnect. The amount of time that this operation takes is important, as it will determine the number of packets missed. It might instead be possible for disconnected peers to drop persistent data and request a new game state image upon reconnect to the network. This will negate the potential issues that reconnect presents but will increase the load upon the new host and the network as a whole as more data is transmitted. Another approach might be to connect to multiple hosts and pick a single host to communicate through, switching if that host is lost. This approach will reduce the chance of complete disconnections, but not eliminate them entirely.

This project aims to provide a "plug & play" solution to handling dynamic mesh networks for communicating game state data. In order to provide a library that is as accessible as possible certain aspects will be abstracted. Sockets, for example, will be hidden, and peers will instead be addressed using Unique User Identification Numbers (UUIDs) which are consistent across the network. Events will be reduced to connections and disconnections. Reconnecting will be handled as autonomously as possible, requiring little or no intervention by the user. The primary protocol that differentiates network events and game data and communicates message targets and sources will be hidden. Although this has the potential to reduce the flexibility of the library it will hopefully open use to programmers with less experience in the lower level handling of such systems.

2.4.1 unattainable targets

The time given for this project is limited and existing expertise when it comes to network handling is minimal. Because of this there are several tasks which might not be completed by the deadline. These tasks may compromise the reliability of the network but should not prevent a basic demonstration of the concept from running. Hurdles that might not be overcome include handling of the disparity between WAN and LAN networks, as this requires a larger amount of IP address handling and comparison, and the persistence of game state data itself, as this utilises the same principles as persistent network imaging does, only on a different layer of the network. Anti-cheat and exploitation methods will also very likely not be implemented, as they are not necessary for the operation of the network and involve multiple other disciplines in programming that have not been explored.

The resulting library with likely work either solely on WAN or over LAN, but will not handle a combination of the two reliably. To do so would require another module to handle IP addresses in more depth, able to differentiate between a local device and a connection from outside the network, and able to determine port forward availability. In order to reduce the difficulties in testing, the network will likely only be tested across a range of local machines, although this does not provide an accurate real-world scenario as latency times will be much lower; a factor which might greatly influence the reliability of the network.

chapter 3 : theory, design, & methodology

3.1 target device architecture

The eventual goal is to write the library to be as portable as possible, but due to the difference in the way sockets behave between platforms and architectures there will always be some outliers. The most popular systems include Windows, Linux, and MacOS, but these can provide overlap with alternative, similarly structured systems. Programs that will build for Linux will often also build for MacOS and OpenBSD or FreeBSD, for example. Because of this, Linux seems to be the best target architecture that will allow for the largest number of potential devices. A choice should be made between the different CPU architectures, such as x86-x64, x86, and ARM, with the potential inclusion of newer architectures such as Risc-V. Because x86-x64 based machines often are more capable than the alternatives it might be easiest to develop solely for this architecture, especially as most development machines will also be compatible, but this does not mean that the library can't be test built for a lower-power architecture such as ARM. The addition of ARM devices decreases the cost of testing, as they often are cheap to purchase and set up, allowing for a small debugging network to be constructed easily. Windows is a lot less easily available on ARM devices than Linux or OpenBSD is, which also negatively effects the cost of development for Windows systems.

To aide in multi-platform development the language used will be C++. Unlike a lot of languages which compile to byte-code and which must be interpreted by the system or by a virtual environment, such as with C# or Java, C++ is compiled into machine code which can run anywhere (or at least anywhere it is compiled for). C++ also provides good implementations of the Posix compliant and WinSock libraries for network construction, as well as a range of tools and data structures which will be useful. As the platform chosen is Linux and the language is C++ it becomes obvious that the build system will be GCC. The construction of Makefiles can be simplified using tools such as CMake.

3.1.1 posix versus win32

Nearly all platforms aside from Windows are Posix compliant. These include MacOS, Android, FreeBSD, most flavours of Linux, and OpenBSD. The existence of the Posix standard together with high-level languages such as C or C++ meant that the programmer could write a single program which needed only be recompiled to run on any machine from a personal computer to the largest mainframe, as long as that machine was running a Posix operating system, states Korn D.G in a US patent on porting Posix to Win32, 2001 [7]. Recently Microsoft released the Windows Subsystem for Linux, or WSL, which provides a compatibility layer between Linux and Windows, allowing for Unix sockets support [8] among other tools. Due to this, and the choice of Linux as a development system, Posix is the best target for development, although some flavours of Linux and some more obscure Unix based operating systems implement aspects of socket handling in mildly different ways, complicating things a little.

3.2 required libraries

The libraries utilised must be picked to be as architecture and platform agnostic as possible. This means that they provide implementations for each of the platforms that will be targeted, and abstract away the handling of which to use. This is called "platform code" in the patent on Multi-Platform Gaming Architecture by Muir Robert Linley, 2004, and it may perform the separation and distribution of game functions for those platforms for which it is required [9]. Most of the standard libraries for string, vector, and queue data types will be used, as will the libraries for handling threads, atomic variables, mutexes, and conditional variables.

To test game state syncing, if it's even started, a graphical output would be ideal. This can be achieved using OpenGL, the most cross platform (without being absurdly verbose) graphics library, and a cross platform window manager such as GLFW or SDL. SDL provides a much larger number of features but is considered too heavy for the needs of the project. To handle OpenGL contexts an OpenGL handling library must be used, such as GLEW or GLUT. For vector and matrix mathematics GLM will be used, as it is an industry standard.

System socket, netdb, and unistd libraries will be used for the bulk of the networking, for socket handling, address structure handling, and the Posix API respectively. These allow connections to be made, ports to be bound, IP addresses to be resolved and data to be sent and received. The Posix file descriptor set will be used to store active socket numbers, as this pairs well with the select function.

3.2.1 socket handling & file descriptors

There are two main methods for handling multiple sockets or file descriptors at a time. Sockets are treated as file descriptors in Unix systems, and can be read from, written to, and can throw exceptions. The older method of the two is the select function which takes a file descriptor set and waits for one or multiple to be readable or writable, then returns. This is normally a blocking function, but it can be set to time out. This is part of the Posix specification and so is implemented in the largest number of platforms and architectures, but the implementation may not always be the same in all. For example, in some implementations the timing data structure passed to the select function is modified to return the time left on the timer, whereas in others it is not. Select will check all file descriptors in a set, even if they are not set, reducing the efficiency of networks with a large number of connections.

The second, and newer, method for handling many sockets at one time is the poll function from the Single Unix Specification and the Unix 95/98 standard. Although the poll function is less wide-spread, it provides a faster method of handling many connections at once. It is more efficient in that it scales better with higher numbers of sockets and it also does not destructively check the file descriptors in a set, unlike the select function which requires rebuilding of the set each time it is called. The poll function also allows for a higher number of file descriptors, as it is not constrained by the same rules select is. By default, the select function can handle up to 1024 connections as it utilises fixed sized bitmasks for file descriptor info.

Due to a mesh network node combining the concept of a host and a client both the sockets for the host peer and any child peers are stored in the same file descriptor set. This means that the network image must be used to differentiate data received from children from data received from the host as disconnection events from each must be handled differently. If a child were to disconnect they must be removed from the network image, along with any of their children recursively. If the host were to disconnect then the client must search for and connect to a new host.

As one of the main aims is portability, and the very design of the type of mesh network results in a small number of connections over a large number, select is the most appropriate file descriptor handling function to use. The select function is a blocking call by default and is used when handling received data from any open sockets. All sockets, including the listening socket of the bound port, are added to a file descriptor set and select is run on this, returning only when a socket has readable information. The sockets are then looped through until a set socket is found, and the received data is acted upon. If the socket set is the listening socket, a new connection attempt has been made and the connection can be accepted or rejected. Otherwise, the received data may be normal packets or a zero length, or negative length, value. A zero-length value means that a normal disconnection event has occurred, whereas a negative length means that a network error has occurred.

There exists an alternative in event-based socket handling, which has been shown to improve increasing numbers of descriptors; a task which both poll and select struggle with. These interfaces are also limited in the respect that they are unable to handle other potentially interesting activities that an application might be interested in, states Lemon, J, when discussing KQueue, a Generic & Scalable Event Notification Facility, 2001. These might include signals, file system changes, and AIO completions [10].

3.3 threading & blocking

Many of the socket handling calls can be set to either blocking or non-blocking. Blocking means that the function will pause the running thread until a certain criterion has met and data can be returned. This can be useful to prevent an unnecessary use of computing resources when an input or similar action is being waited upon but in certain situations it can be detrimental. Blocking functions often work much better in a non-primary thread, as they can be paused indefinitely without negatively effecting the performance of the program, but the addition of threading increases complexity considerably.

Transmitting and receiving data takes a long time, at least compared to simpler operations that don't require any form of input-output. As with resource managing systems in game engines, it is often best to assign tasks which handle file descriptors to a separate thread to avoid any large pauses in the main thread. In this case, when utilised by a game engine, pauses on the main thread can prevent the rendering step from occurring at regularly timed intervals, resulting in low or choppy FPS (Frames Per Second).

Threading, however, can introduce several potential problems such as race conditions and deadlocks. The first of these problems can be avoided using synchronisation techniques. Synchronisation involves indicating areas of memory which must be locked to be read from or written to, often using data types such as mutexes, resulting in only one thread being able to modify the data at a time. This eliminates the chances of two threads attempting to modify the same block of data at once, resulting in an incorrect value as only one of the modifications is made while the other is overwritten. If this is performed with multiple areas of memory it can lead to the second problem. A deadlock is when two or more threads require access to multiple blocks of data, but each thread manages to lock only a portion of the data it needs. Both threads then will wait for the other to unlock the data, but neither can progress so the data will never get released. A simpler protection method involves the use of atomic variables, meaning that to modify the variable only a single operation must be made, negating the chances of another thread attempting to access the data at the same time as it is being modified. These atomic variables are only available in very simple data types, such as booleans, integers, chars, shorts, and longs.

As the data receiving is all performed in one looping function that includes a blocking call it would be best to move this to a separate thread. The data received can be added to a queue that is protected by a mutex, so that it may be accessed from the main thread. Due to the blocking call no extra thread handling will have to be performed to reduce redundant processing usage. The same cannot be said for the sending thread, which will read from a mutex protected queue and broadcast the message to the appropriate recipients. When there are no messages to broadcast the thread must sleep, but there is no way to know how long for and so a conditional variable must be used. A conditional variable uses a mutex and a lock in order to notify a thread on when a certain criterion is met. Because locking a mutex is a blocking call, the thread will sleep without consuming large amounts of resources until otherwise notified.

This method provides multiple advantages in that the user of the library does not have to perform any actions for messages to be sent or received passively. Messages will continue to queue up in the receive queue whether they read from it or not, meaning that they can be read at any time. Messages that are received and that must be forwarded can be forwarded without waiting for the next update step in the game engine and sending and receiving messages does not affect performance in any significant fashion. This results in slow running games not effecting the overall performance of the mesh network, as only the information that they themselves are providing will be delayed or intermittent.

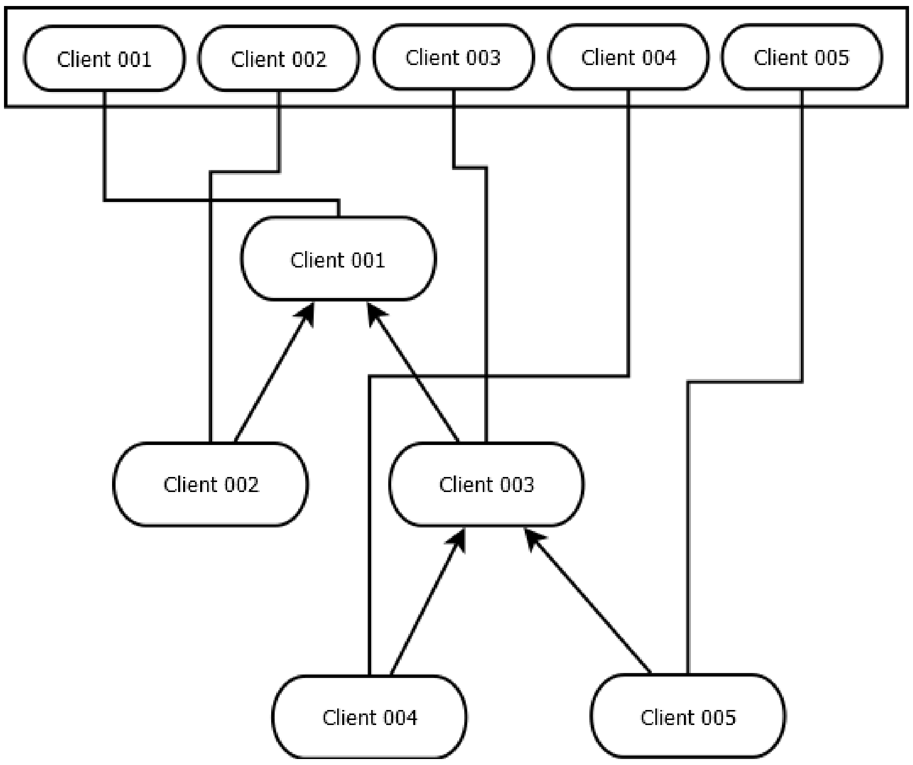
Delegating a thread each for these tasks is quite coarse granularity when it comes to multi-threading. Granularity refers to the amount of work executed per concurrent task. A fine-grained breakdown of tasks results in relatively higher levels of system overhead, according to Tulip, J, in a paper on Multi-Threaded Game Engine Design, 2006 [11]. The large granularity of these tasks means that the overhead increase is relatively insignificant as task switching will not happen too often.

3.4 network structure

The first, and most intuitive, method for storing a network image during run-time is that of a tree data structure. As each member of the network is a node with a single parent and multiple children this method seems ideal, but it does present complications when attempting to quickly navigate the structure. Although use of pointers and references in the C++ language improves memory management as, unlike with a vector data type when the structure is modified the memory does not necessarily have to be reallocated, the structure itself can be navigated bidirectionally. Unlike with a list, which would only have to be navigated in a single direction, a tree can be traversed in multiple ways, which both complicates algorithms and decreases efficiency of even simple tasks such as searching for a node with a certain identifying feature. Tree traversal methods include depth-first techniques, such as inorder, preorder and postorder.

One counter-argument to the memory management advantages that such a data structure provides is the use of linked lists, which work in a similar fashion, but reduces the number of child nodes to one, eliminating the need for complex traversal algorithms. Unfortunately, using only a list type structure does not allow for an accurate indication of network relationships, resulting in a list of peers without a discernible hierarchy.

A potential compromise comprises of a hybrid structure. A tree can be created, and each element of the tree can be also stored in a list for efficient information grepping. Duplicate data can be eliminated using C++ pointers, although this can introduce problems with dangling pointers if not handled cautiously. A dangling pointer is a pointer which no longer point to a valid object of the appropriate type, most likely because the target object has been deleted. This method does introduce complexities in that a single modification of the network would require multiple operations to be correctly reflected in the data structures.



A peer in this network structure will have to contain all the information required for identification and connection. It might also contain meta-data such as the peer name and client version number. An association must be made between the socket of the peer, if the peer is a direct connection, and the unique user identification number of the peer.

```
Peer {
    int uuid;                                   // unique user identification number
    int sock;                                   // socket of connection

    string name;                                // name of peer
    string addr;                                // IP address of peer

    Peer* host;                                 // the peer that this peer is connected to
    vector<Peer*> children;                     // the peers that are connected to this peer
};
```

3.5 mesh protocol

To keep this network structure up to date a distinction must be made between packets which contain game state data, and packets which contain network event data. This is achieved using a protocol, or a layer in which the data is wrapped with a header which describes it. Examples of protocols include Transfer Control Protocol (TCP) and User Datagram Protocol (UDP). Both protocols act upon the same layer, and each have their advantages and disadvantages.

The User Datagram Protocol is a protocol for communicating datagrams between devices. A datagram is a set of information, potentially split into multiple packets, which is broadcast at the target device. No response is expected, meaning that this protocol does not usually open and maintain a connection between two devices, and is known as a connectionless protocol. The sender does not know if the receiver has received the information, and as such, packet loss can occur. Packets can be received out-of-order, or not at all, but they will always contain the correct information. More light weight than TCP, and quicker due to the communication not requiring multiple steps, this protocol is often used when packet loss is acceptable, and speed is important. The Transfer Control Protocol is a protocol for reliable two-way communication of data between multiple devices. A connection can be made and maintained, and while this connection is open any data can be sent and received through it. No packet loss will occur, and packets will arrive in the correct order.

In this use-case it would be best to create a protocol one layer under TCP. Reliability of communication is important, as a single lost packet could result in a de-synced network image which could cause problems further into development. Speed of transmission is not so important in this demonstration, although in a full commercial product it would be a significant discussion point. Many fast paced first person shooter games have picked UDP over TCP to reduce latency of packets, such as Quake 3, Sanglard, F, 2012 [12]. An ideal solution would be to utilise UDP while giving packets priorities. Packets with a certain priority will be stored in a buffer until a response with that packet identification is received. After a certain amount of time without this response the packet will be re-broadcast. This means that with relatively unimportant packets you would still get the advantage of speed that UDP provides but with packets which must be received there is a safety net to prevent packet loss. Introduced into games such as Quake 3 is a "multicast" system in which packets can be marked as reliable, implementing a portion of the TCP call and response over UDP. As Cronin, E, et al (2001) state, reliability is achieved using a stop-and-wait protocol with a single bit reliable sequence number [13].

When it comes to developing a protocol there are a few considerations to be made when designing the header, such as whether it is human readable or not, or whether it is a dynamic or fixed length. These options can influence network reliability, extensibility, speed, and efficiency so it is important to find the correct balance.

A fixed header length simplifies the process and improves parsing speed for each packet transmitted. Less data is required to transmit the same header information, improving efficiency, but later modifications to the header structure can cause complications. If more header space is required at any point the result will no longer be compatible with older versions of the program. This header structure must be modified in every instance of code which either serialises or parses a packet, which can become time consuming.

[07][GAMEDAT][“This is some game data”]

The first two digits indicate the length of the header, which in turn describes the packet data. In this case the type of packet is that of game data, not a network event, and so is indicated using the key word “GAMEDAT”.

[10][NEWCONNECT][Client Name:Client IP Address]

This packet is a new connection broadcast, with a header length of 10 characters. The packet contains metadata on the new node in the networl.

A dynamic header, however, is easily extensible. If maintaining compatibility between versions of the program is important, as it might well be when it comes to persistent networks, then a dynamic header might be the better choice. The possibility of incompatibilities opens a discussion into the value of allowing multiple iterations of the tool to connect to the same network, which will be saved for later.

The most sensible choice for development of a protocol, and a tool which implements said protocol, appears to be that of human readable headers over the alternative. This will greatly simplify debugging, as flaws in packet transmission or handling are immediately obvious, but the decision also ties into the choice between fixed and dynamic headers. Human readable information may differ in length, making fixed headers a limitation rather than an advantage. Instead, information such as data type can be stored as enumerable values, so that they can be parsed from machine readable data to human readable values more easily. Enumerating data also greatly improves efficiency, as assigning a single number to what would otherwise consist of multiple characters reduces the space the data takes up considerably. One disadvantage of this approach is that packet header structure must be documented thoroughly so that tools which capture packets for debugging purposes are not rendered useless, or at the very least more difficult to use.

One of the main goals of this project is to reduce the cost of server hosting. The main way in which this will be accomplished is by delegating the task of hosting to the users of the network, but that does not mean the program can then be inefficient when it comes to data usage. Not only will reducing the amount of data per packet improve performance but considering the user's situation is also important. Users might be subject to data caps or limits and so reducing the burden on them as much as possible is also a priority. Because of this, a fixed header with non-human readable data will be used, as this offers the greatest efficiency and speed when it comes to transmitting and parsing packets.

3.6 network event breakdown

All network packets contain a header that is setup as follows:

TARGET_UUID:SOURCE_UUID:PACKET_TYPE:PACKET_DATA

UUIDs are always three characters long, and so they must be padded with zeros if the number is too small. If the packet is a global broadcast, then the UUID of 000 can be used. In this way packets can be associated with a source peer and a target peer and can be forwarded smartly to reduce the amount of redundant data transmitted. A global packet will always be forwarded throughout the network whereas a packet with a specific target UUID will be sent only through the correct path to the target.

The packet type can be one of the following:

GAMEDAT == 0:

This packet is just game data. It will be forwarded around the network but not parsed and acted upon.

NEWCONN == 1:

Contains a HOST_UUID that must be associated with the SOURCE_UUID in the header.

REMCONN == 2:

Contains a PEER_UUID of the peer that has lost connection to the network and should be removed.

SETSOCK == 3:

Contains no information, as the SOURCE_UUID can be obtained from the header information.

SETNAME == 4:

Contains the connection name to be associated with the SOURCE_UUID in the header.

SETIDIP == 5:

Contains the connection IP address to be associated with the SOURCE_UUID in the header.

NETSTAT == 6:

Contains a serialised network image or a portion of a serialised network image to be parsed.

Deleting and updating content is non-trivial in a peer-to-peer environment, if consistency among copies needs to be maintained, Androutsellis-Theotokis, et al, 2004 [14]. A combination of these network event packets in the correct order results in a persistent image of the network structure on each client connected.

3.6.1 connection handshake

To create a fully decentralised mesh network all members of the network must be treated the same. Each node in the network will have to be as self-sufficient as possible. There is no one device that allocates identification numbers or target nodes to new connections. There is no one device that handles user input from each peer. There is no one source of information on the network structure. Instead, these tasks will have to be accomplished collaboratively, using a range of handshaking techniques designed to communicate information in the correct order.

One consideration to make is to decide which device is the initiator when a new client joins the network. Either the host or the client could opt to inform the network, but only one can correctly assign a new, unused user identification number due to the availability of the information on which are already in use. This means that the host must make the first move: to inform the new client of the current network structure. The handshake goes like this:

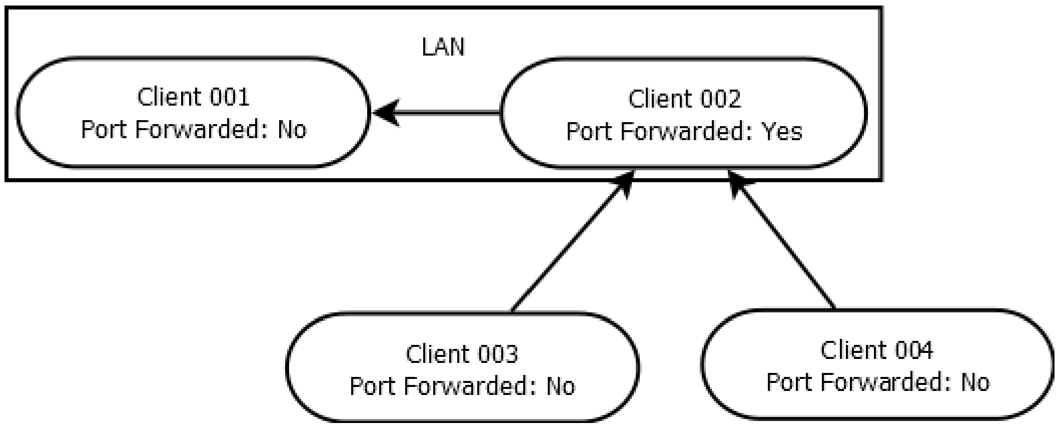
- Client attempts connection to the network
- Host accepts new connection
- Host serialises and sends the network image directly to the new client
- Client receives and parses this network image, making a note of the socket it was received from
- Client generates new UUID and broadcasts itself as a new connection
- Client sends UUID directly to host so it can associate the socket number with the UUID

In this way every user in the network has knowledge of the new connection, their UUID, and which host they are connected to. The UUID generated by the new client is valid, as they have the whole network image to base it off. Both the host and the child have associated the correct socket numbers with the correct UUIDs so that they may direct messages correctly and identify received packet sources.

3.6.2 reconnect events

One of the main uses of the network image construction is for autonomous reconnecting to the network should a host peer drop connection. Once a peer has all the information on the network structure picking an appropriate target to reconnect to is relatively simple, although it depends on a few factors. Reconnect events can promote network load balancing if done correctly as when the choice of a new host is made the current load on each peer can be considered. Other factors which are important when selecting a new host are whether there is a host that is on the same network, which would improve network latency and performance, and which peers are accessible from outside of their own networks, as not every user might have port forwarded their connection.

As noted above, the amount of time it takes for this reconnect process to complete is important. The longer it takes, the more packets might be missed, resulting in an incomplete network image or skipped game state data. One solution suggested in the introduction is that of requesting the complete game state or network image serialised from the new host, as if connecting to the network for the first time. This increases the amount of data transmitted but does not reduce reliability.



There are many edge cases in this network structure, and one of the most complex shows itself when managing disconnected peers. In this scenario there are three or more nodes in the network. The first node that started the mesh network is not port forwarded, yet it is connected to by another peer that is part of the same local network. This second node is port forwarded, allowing the third peer to connect from outside of the local network. If the second peer then disconnects the third peer will search for a node to reconnect to, finding the first node, and attempt to reconnect. Unfortunately, because the first peer is not accessible from outside the network the reconnect fails and the third node fails to re-join the network.

In another scenario in which a node that has multiple children disconnects there is a scramble to find a new host to connect to. If the node that has disconnected is the root node it results in a network split. None of the child nodes can find a node further up in the hierarchy to connect to, and they cannot attempt to connect to each other as this would create a circular structure which might capture and trap packets. A potential solution is to force each host that has multiple children to nominate a child peer to inherit its position in the network should it disconnect. All other children of the disconnected node would then attempt to connect to one single child, while this child becomes the new root node.

3.6.3 load balancing

Although reconnect events give the network a chance to re-balance itself somewhat, it might be necessary even when no client disconnects for several reasons. A user might connect to a heavily loaded node manually, unbalancing the network, or other aspects of the network might change. Fortunately, performing load balancing that is not triggered by a disconnect is a better position to be in as you can create the new connection before disconnecting or being disconnected from the network, resulting in no loss of packets. The solutions suggested for avoiding such a loss of data in the reconnect section are no longer needed, improving efficiency.

Load balancing offers the opportunity to not only reduce computational load on nodes in the network but to reposition nodes in a pattern that is more efficient in terms of geographical topology. The direct path between a pair of clients may have longer round-trip times than a detour path between them, because of the triangle inequality violations in the internet, according to Ly, C, in a paper on Detour Routing for Online Multiplayer Games, 2010 [15]. To rearrange nodes in a pattern that would setup certain peers as relays would be beneficial for latency but would require significant amounts of IP address and round trip time analysis in order to perform optimally.

In a way this is similar to a US patent on Minimizing Bandwidth Costs for Online Games, 2013, by Miyaki, K [16], in which they aim to provide an online game with capability to dynamically adjust the networking topology: querying a participant of the online game to determine preference and position of the participant in the networking topology of the online game. In this patent the focus is more on providing a monetary incentive to users in order to convince them to contribute to the network topology.

3.7 basic network imaging

The complexity of a mesh network is largely due to the need for all data to be synced between all peers that are connected, including data that was sent prior to a peer's connection. When a new device connects to the network it must somehow receive all the game state changes so far so that it can catch up. The same method can be used to keep all members of the network up to date on the network structure itself. The task of this network layer is the establishment and management of a peer-to-peer network containing all currently active player computers, Schiele, G, 2007 [17]. A range of methods for storing and transmitting this data present themselves, each with pros and cons.

The simplest way to handle this task is to send the whole network state or game state every frame. This is slow, and incredibly inefficient. The amount of data sent each frame will increase with the size of the network or the complexity of the game state. A much more efficient alternative is to send only the difference between frames or states. To quickly catch any new peers up the game state or network state must be serialised by the host of the new connection, sent in a single target message to the new client, and parsed correctly. This is likely the least data efficient part of the process, but it only has to happen once, every time a new peer joins or re-joins the network.

Network structure serialisation is relatively simple. The root node must be found, and then all child nodes are recursively serialised and added to a string, separated by a special character. Starting at the root node means that when the string is parsed in no node will be added to the network structure that does not have a valid host already added. This reduces potential errors during the parsing process. To parse the serialised data the string is split on the same special characters as before and then each set of node data are added to the network as if they were a new connection.

3.7.1 IP address grepping

To find a device's IP address is not simple as a device might have multiple addresses depending on the hardware available. Simply looping through all internet enabled hardware modules and picking the most likely address is not a reliable method. This method will also only return the device's local IP address at best, which is only useful if connecting from inside the local network.

The most reliable method of finding the IP address of a machine involves connecting to another device, preferably outside of the local network, and asking it what IP address the device connected from. In a normal host-clients structured network the host can be used to figure out the address, but in a mesh network there might not always be a host device.

To make each client as independent as possible it would be preferable if they could connect to a third-party service in order to determine their own IP addresses. A service such as Amazon Cloud or Google Search is very unlikely to go down anytime soon, which is good for ensuring the lifetime of the product. If longevity is the target, though, the best option might be to instead delegate IP address handling to the writer of the tool that utilises this library. If the IP address of the current device could be passed in when the server instance is created, it would allow the user to pick a more up-to-date service or alternate method for finding addresses.

It might also be useful to find an IP address before any sort of connection has occurred, in the event that a player wants to share their address with a friend or family member. If this occurs over the local network both parties might fall prey to server address reconfiguration, which can break server software, which typically binds to a (presumable immutable) address to accept incoming messages, Guttman, E, 2001 [18].

One final method that can be used is to have new peers communicate which IP address they used to connect to the network, associating the host peer with the given address. This might not be reliable as if the new peer connected over the local network with a local IP address the network will associate the local address with the host, which is not ideal. One upside to this method is that you know for sure that the address is correct in some way, as the peer managed to connect using it.

chapter 4 : testing & evaluation

4.1 data throughput

Packet analysis of capture from perspective of root node:

First Connection: 192.168.0.15 To 192.168.0.29

- 74 From	[Connection Request]
- 74 To	[Connection Request Confirmation]
- 66 From	[TCP ACK]
- 90 From	["This is a test packet;"]
- 66 To	[TCP ACK]
- 90 To	[Network Image Serialised]
- 66 From	[TCP ACK]
- 73 To	[Set Socket Host]
- 66 From	[TCP ACK]
- 72 From	[New Connection]
- 66 To	[TCP ACK]
- 113 From	[Set Name] [Set Socket] [Set Host] [Set IP Address]
- 66 To	[TCP ACK]

Totals: 547 bytes from (349 bytes without TCP ACK) | 435 bytes to (237 bytes without TCP ACK)

Second Connection: 192.168.0.16 To 192.168.0.15

- 90 From	["This is a test packet;"]
- 66 To	[TCP ACK]
- 72 From	[New Connection]
- 66 To	[TCP ACK]

- 79 From	[Set Name]
- 66 To	[TCP ACK]
- 93 From	[Set Host] [Set IP Address]
- 66 To	[TCP ACK]

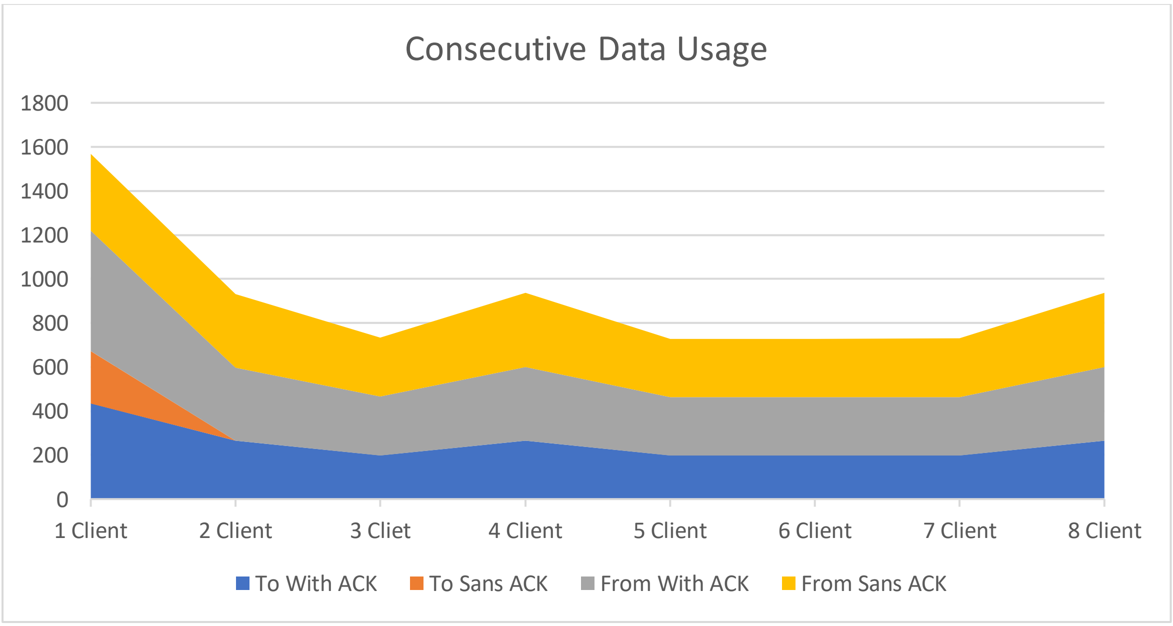
Totals: 334 bytes from (334 bytes without TCP ACK) | 264 bytes to (0 bytes without TCP ACK)

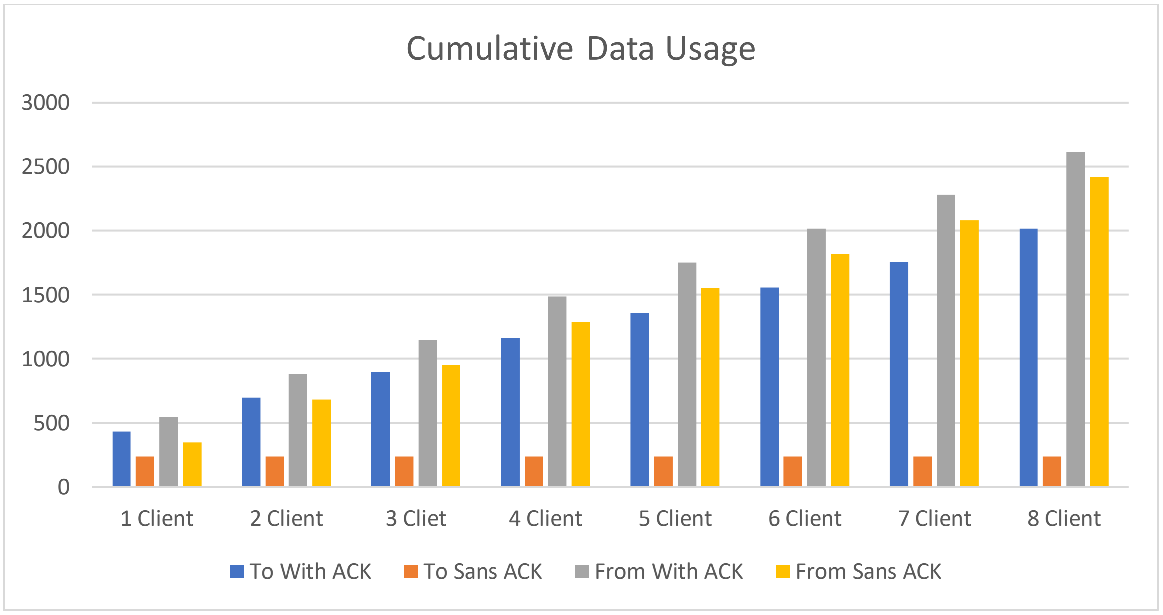
Third Connection: 192.168.0.66 To 192.168.0.15

- 90 From	["This is a test packet;"]
- 66 To	[TCP ACK]
- 72 From	[New Connection]
- 66 To	[TCP ACK]
- 106 From	[Set Name] [Set Host] [Set Ip Address]
- 66 To	[TCP ACK]

Totals: 268 bytes from (268 bytes without TCP ACK) | 198 bytes to (0 bytes without TCP ACK)

This is the captured packet history of three devices connecting to the root node. The first device connects directly, whereas the other two devices both connect to the first, so indirectly.





There will always be slight variations in data usage with new connections due to variable length data such as client names. Larger variations in data usage can be attributed to inconsistent packet merging. Packet merging is performed by the Huffman algorithm, a particular type of optimal prefix code that is commonly used for lossless data compression. Packet merging is incredibly useful for reducing the amount of unnecessary data that is transmitted, although it can introduce some latency to the system. By compressing the messages, the distributed system can save bandwidth at the cost of computational power, Smed, J, 2001 [19].

The aspect that the data showed that was not expected was just how large the headers are once you get to the TCP layer. Up to an estimated 50 bytes per packet is utilised by layers of the networking stack that you do not see. This is one reason why packet merging algorithms are so useful. Another aspect that was surprising was just how much data was taken up by TCP packet acknowledgements because of this. For packets that are not overly important, or have a short usefulness half-life, it seems a much better idea to implement UDP over TCP; eliminating the need for the call and response pattern of information transmission that TCP uses.

As expected the load on the root node might get increasingly higher in a linear fashion with each new connection, but the direct connections incur a much higher load than indirect connections. This means that were the network correctly balanced each new connection should have little effect on the network load. The main reason direct connections require more data throughput is that the host in a direct connection must communicate large volumes of data that the network has collated so far.

Data throughput is an important factor, as larger amounts of data might incur a higher cost for users of the network, but perhaps more important is the latency induced by the network structure. In fast paced, or real-time games such as first-person shooter games even small amounts of latency can prove for a large disadvantage, although there are some factors which can remedy this slightly. The delay value which can be tolerated by the participants depends on the used camera perspective... at least 50ms of additional delay can be tolerated [using certain fields of vision]. In a paper on the Impact of Delay in Real-Time Multiplayer Games, 2002, by Pantel, L, claims that for games such as first-person shooters, presentation delays of 100ms or more may be acceptable [20]. Not all player actions are equally tolerant to latency. Some actions such as shooting a sniper rifle at a moving opponent are greatly impacted by latency, while other actions such as selecting a set of troops and moving them across a battlefield tend to be less sensitive, Claypool, M, 2006 [21].

Packet merging and data compression aren't the only methods of reducing the amount of data transferred over the network. One particularly interesting method is described by A Bharambe, et al, 2008. To extrapolate the behaviour of a remote player that sends only infrequent information (because it is not in focus), we use a special replica called a doppelgänger. A doppelgänger is a bot... running on the local machine whose goal is to act in a manner that realistically approximates the behaviour of the remote player (using guided AI) [22].

4.2 socket & computational load

In a traditional network structure, the socket load for clients is very low, but for the server it increases linearly with the number of clients connected. To remove the centralised server aspect completely these clients must take up the slack, but with efficient load balancing the number of connections per client could potentially be kept between two to four, which should not incur any discernible performance penalties. Extra computational load might be attributed to the need for all packets received to be parsed before forwarding even if the packet data is not useful for that client.

Because all important network handling functions are handled in a separate thread even slow machines should not have their game performance effected in any significant way, showing that this is a viable method for eliminating a central server architecture.

4.3 debugging process

The debugging process of networking code consists largely of the same tools and practices as with other programs, with the addition of software such as Netcat and Wireshark. Netcat can be used to connect to and interact with TCP based networks to manually give certain inputs and monitor the outputs. Wireshark can be used to capture and view all packets broadcast to and from a certain application or interface, allowing for statistical analysis, packet evaluation and other such methods of network diagnosis.

Generic tools for debugging C++ code include Valgrind and Memwatch, which can hook into your program and watch for memory leaks, un-allocated variables and dangling pointers. Often the IDE in use can catch and warn these problems if the code is run through their own debugging tools.

4.4 example scenario

The setup used during the evaluation phase consisted of a mix of x86-x64 devices of differing capabilities and a handful of low power ARM devices, all running on the same network, connected over ethernet. This does not represent a real-world scenario terribly well, unless the mesh network capabilities are limited to a LAN connection, but it provides more realistic latency statistics than performing the same test on a single machine using docker or virtual machines. All devices ran the same flavour of Linux as this was the simplest way to build and run the tool for each. These devices include, ranging from most computationally capable to least:

- x86-x64 AMD based desktop computing device
- x86-x64 Intel based portable computing device
- ARM 64Bit Broadcom computing device (Raspberry Pi 3 B+)

4.5 alternative potential use cases

This mesh architecture need not be restricted to use in video games. The persistent data aspect might lend itself to many forms of chat clients or embedded networks such as in a mesh network of actual physical devices. The central idea is very transferable, even if the libraries used are not.

chapter 5 : conclusion

5.1 evaluation of progress

The progress made during the project was not as planned. The complexity of the system as a whole was hugely underestimated due to a lack of experience in network programming. A minimum viable product was not quite achieved as it was lacking major features such as autonomous reconnect handling.

However, a common network image was achieved. New nodes in the network would correctly receive and parse the current network state, and every node in the network would correctly receive connection, disconnection, and chat events. The network was relatively efficient considering the use of TCP over UDP, and the library did have some multi-platform capabilities, as it was successfully built for both x86-x64 and ARM devices.

The next task would have been to have used the information stored in the network image and set up automatic reconnecting, ensuring the network image was updated correctly on the disconnected client. Evaluation of appropriate new hosts would have come next, with a number of factors that would affect the decision indicated and discussed.

5.2 viability

What limited testing has been performed shows that this form of network at least works with small numbers of clients. As select has been used over alternatives such as poll or an event driven system there will be a hard limit on most systems of 1024 connections, but as not every client is connected to each other but connected through other nodes this limit does not affect the network. The most load will be put on the root node of the network, but as the root node can be disconnected at any time like any other peer it's likely that when the network reaches a stable state the device that is hosting the network will be capable.

Viability testing for large numbers of devices, say into the triple digits, has not been performed due to limited resources.

5.3 future work

Future work would include continued work on the library to improve speed and reliability, possibly switching out TCP for UDP and working on a multi-cast system with packet importance. Full integration with an industry standard game engine might also be a goal, as it shows the platform-agnostic aspects of the library and will enable useful visual debugging processes.

Comparison with existing products would be advantageous, especially with a module of the Mammoth networking framework called Postina, described as a self-organizing peer-to-peer network engine using tree-based broadcast, Kienzle, J, 2009 [23]; a description which fits this project very closely.

chapter 6 : references

[1] Internetworldstats.com. (2019). World Internet Users Statistics and 2019 World Population Stats. [online] Available at: https://www.internetworldstats.com/stats.htm [Accessed 2 January 2019].

[2] GauthierDickey, C., Zappala, D., Lo, V. and Marr, J., 2004, June. Low latency and cheat-proof event ordering for peer-to-peer games. In Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video (pp. 134-139). ACM.

[3] Duong, T.N.B. and Zhou, S., 2003, October. A dynamic load sharing algorithm for massively multiplayer online games. In The 11th IEEE International Conference on Networks, 2003. ICON2003. (pp. 131-136). IEEE.

[4] Bethea, D., Cochran, R.A. and Reiter, M.K., 2011. Server-side verification of client behavior in online games. ACM Transactions on Information and System Security (TISSEC), 14(4), p.32.

[5] Yahyavi, A. and Kemme, B., 2013. Peer-to-peer architectures for massively multiplayer online games: A survey. ACM Computing Surveys (CSUR), 46(1), p.9.

[6] Jardine, J. and Zappala, D., 2008, October. A hybrid architecture for massively multiplayer online games. In Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games (pp. 60-65). ACM.

[7] Korn, D.G., AT&T Corp, 2001. Porting POSIX-conforming operating systems to Win32 API-conforming operating systems. U.S. Patent 6,292,820.

[8] Windows Subsystem for Linux. (2016). WSL Networking. [online] Available at: https://blogs.msdn.microsoft.com/wsl/2016/11/08/225/ [Accessed 10 February 2019].

[9] Muir, R., Muir Robert Linley, 2004. Multi-platform gaming architecture. U.S. Patent Application 10/648,178.

[10] Lemon, J., 2001, June. Kqueue-A Generic and Scalable Event Notification Facility. In USENIX Annual Technical Conference, FREENIX Track (pp. 141-153).

[11] Tulip, J., Bekkema, J. and Nesbitt, K., 2006, December. Multi-threaded game engine design. In Proceedings of the 3rd Australasian conference on Interactive entertainment (pp. 9-14). Murdoch University.

[12] Sanglard, F., 2012, June. Quake 3 Source Code Review: Network Model. [online] Fabiensanglard.net. Available at: http://fabiensanglard.net/quake3/network.php [Accessed 9 May 2019].

[13] Cronin, E., Filstrup, B. and Kurc, A., 2001. A distributed multiplayer game server system. In University of Michigan.

[14] Androutsellis-Theotokis and D. Spinellis. A survey ofpeer-to-peer content distribution technologies.ACM Com-puting Surveys, 36(4):335–371, 2004.

[15] Ly, C., 2010. Latency reduction in online multiplayer games using detour routing (Doctoral dissertation, Applied Science: School of Computing Science).

[16] Miyaki, K., Sony Interactive Entertainment America LLC, 2013. Minimizing bandwidth costs for online games. U.S. Patent 8,369,243.

[17] Schiele, G., Suselbeck, R., Wacker, A., Hahner, J., Becker, C. and Weis, T., 2007, May. Requirements of peer-to-peer-based massively multiplayer online gaming. In Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07) (pp. 773-782). IEEE.

[18] Guttman, E., 2001. Autoconfiguration for ip networking: Enabling local communication. IEEE Internet computing, 5(3), pp.81-86.

[19] Smed, J., Kaukoranta, T. and Hakonen, H., 2002. Aspects of networking in multiplayer computer games. The Electronic Library, 20(2), pp.87-97.

[20] Pantel, L. and Wolf, L.C., 2002, May. On the impact of delay on real-time multiplayer games. In Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video (pp. 23-29). ACM.

[21] Claypool, M. and Claypool, K., 2006. On latency and player actions in online games.

[22] Bharambe, A., Douceur, J.R., Lorch, J.R., Moscibroda, T., Pang, J., Seshan, S. and Zhuang, X., 2008. Donnybrook: enabling large-scale, high-speed, peer-to-peer games. ACM SIGCOMM Computer Communication Review, 38(4), pp.389-400.

[23] Kienzle, J., Verbrugge, C., Kemme, B., Denault, A. and Hawker, M., 2009, April. Mammoth: a massively multiplayer game research framework. In Proceedings of the 4th International Conference on Foundations of Digital Games (pp. 308-315). ACM.