Abstract

The thesis describes the development of a networking library to be used in games created for several different platforms, such as Windows, Sony Playstation 2 or Xbox. The networking library has both an abstraction of the UDP/IP protocol and packet management on the basic level, but also a generic message transport layer, providing message reliability and coalescence. A sample implementation is also given, that can be used when porting to, for example, the Playstation 2 or the Xbox. This generic layer is then used in an even higher level, to handle object synchronisation and event passing between clients and servers.

During the development, focus has primarily been on network communication efficiency, to allow large amounts of information to be passed with as little actual data as possible, e.g., packing boolean values into single bits, and packing floating point numbers into fewer bits and restricted representations. Other areas that are examined are dead reckoning algorithms and clustering algorithms and how to make these as generic as possible.

The resulting library is general enough to be useful in a variety of different game types, but also flexible and extensible enough not to become obsolete immidiately after the implementation is done. This is ensured with the use of modern C++ components such as templates and the STL together with a well planned and well modularised API.

Contents

Appendix

Chapter 1

Introduction

Multiplayer gaming across computer networks is a relatively new idea, and the concept of being able to interact in a virtual world together with thousands of other players would have seemed like science fiction only a couple of years ago. Today, Counterstrike is a competition sport much like any other, and multiplayer games are common. Network programming is becoming nearly as central as graphics programming to the game programming process, and network efficiency is more and more important.

The demands on interactive simulations to not only be as realistic as possible but also more realistic than the simulations that preceded them are ever high. The need to remain perched on the frontier of technological innovation remains. Games of today are vastly different from the games made ten years ago, and a game network library from 1995 is as inappropriate for modern use as a graphics rendering library from the same year.

Generally, network libraries and APIs have come in one of two varieties. Either in a collection of low-level routines designed to simply provide an abstraction of the platform-specific sockets implementation and providing some more advanced services on top of UDP/IP (or some other transport of choice), or in the form of a complete framework with hulking class hierarchies and severe limitations on flexibility.

A middle ground between these two extremes should be possible to achieve through the use of modern C++ design features and a non-hierarchal structure, focusing on layers of service delivery rather than rooted structures of increasing complexity. By keeping the connectivity between layers one-way-only and as minimal as possible, the level of abstraction needed by the user does not conflict with the level of abstraction provided by the library. The user can choose which layer to use directly.

The notion of a game network library is not entirely intuitive, and the concept has different meaning to different people. In one sense, it could be used to describe the simple act of collecting data from the user (the application) and transmitting this data to other members of a network. In this case, the actual structuring of client and server, object state transmission and other possible features are all left entirely to the user of the library to implement.

An opposing view might be that a game network library should not only provide these features, but also external object database interfacing, transmission of secure data

for account purposes, inter-server communications and other advanced services used by large scale simulations. In this thesis, the word library is meant to encompass both the low-level socket functionality as well as higher level client/server abstractions.

1.1 Objective

The goal of this thesis is the design and basic implementation of a network library, Gamenet, to be used in current and future products at Coldwood Interactive. The library should be portable enough to enable use on platforms such as PC, PS2 and XBOX, and flexible enough to handle several different types of games.

A set of assumptions needs to be made about how the game works and what kind of information will need to be shared among players. The topology of the network also affects the structure of the library. The goal is not to have a finished and complete library ready for creating a commercial multiplayer game. Rather, the goal is to be able to use it as the starting point of implementing multiplayer functionality.

In addition to the actual library, a sample implementation will be done in collaboration with Coldwood Interactive[1].

1.2 Conventions and assumptions

It is assumed that the reader of this thesis is familiar with the language of C++ and fairly familiar with networks (although experience programming them should not be a requisite). Some basic knowledge of how multiplayer games work is assumed. A list of books on C++ and template programming are given as references, but I have unfortunately not found any good or relevant literature on game networking.

In the protocol descriptions, different bracket types are used to show whether a variable is optional or mandatory. The mandatory items are enclosed in angular brackets, <like this>, while the optional ones are given in square brackets, such as [these]. Other than that, bold fonts and italics are sometimes used for emphasis.

Source code is written in the font used in the following example:

```
void main(int argc, char* argv[])
```

The source code examples use the same font. The reason I have chosen not to use a fixed width font is partially from aesthetic preference, and also because I feel it improves the reading of source as part of a text rather than in a programming environment.

1.3 Outline of the Chapters

The thesis is divided into a number of chapters, each chapter detailing a different stage of the process toward a completed network library. A final appendix shows the results of using the library.

[1] http://www.coldwood.com/

Chapter 2 discusses the choices made in designing Gamenet, and also talks briefly about previous implementations of network libraries made for games, and other inspirations for Gamenet.

Chapter 3 describes the underlying theory and algorithms of the network library, and discusses how these algorithms and theories can be used in the context of the network library.

Chapter 4 describes, briefly, the structure of Gamenet, and how the library works. It also talks about some examples of using the library in an actual application.

Chapter 5 contains the conclusions drawn from implementing the library, and also some final thoughts and ideas of what can be improved on in the future.

Appendix A describes the sample implementation that was made while creating Gamenet, demonstrating use of the library. It also talks briefly about the games using Gamenet, made by Coldwood Interactive.

1.4 Acknowledgements

I would like to thank everyone at Coldwood for their support, especially Jakob Marklund, who functioned as my supervisor at Coldwood. I would also like to thank Pedher Johansson, my supervisor at Umeå University, Anders Backman, for additional references, and Arvid Norberg, for comments.

Chapter 2

Design background

The aim with the design choices made, when beginning work on the library, was to achieve unintrusiveness, the desire to avoid complicated inheritance structures and complex base classes to be inherited by the user of the library. Often, the rigidness of object-oriented designs leads to huge rooted trees, where the classes that are actually useful depend on a myriad of other classes to function properly.

Gamenet will, hopefully, not only manage to avoid this need for assimilation, but also be flexible and self-contained enough to allow for quick insertion into any game without any significant effort. Although, in some cases, the best design is truly achieved with forced inheritance, most of the time a nicer way of building a library can be found, that does not dictate the basic design of the whole application.

2.1 Current implementations

There are many network libraries designed for use in games available. Much can be learned and many mistakes avoided in the basic design of the network communication by looking at other network libraries whose code has been released to the public. Most of these previous implementations are heavily coupled to existing games, and are often very specialised in what they do. Thus, a few of the optimisations that these implementations make are either hard to generalise, or simply unsuitable for use in a general library. However, the basic idea of what is to be done, the sharing of a simulation across two or more computers, is the same in most games, and thus the network code share many similarities.

2.1.1 Quake II

The source code of the game Quake II [iS97] by Id Software has been released publically under a GPL license. This means that it is available as a reference point, but the code itself cannot be included in a library with a license that may differ from the GPL license (such as Gamenet). Thus, Quake II has and will be a valuable insight into how an actual

production implementation might look, but to actually base the Gamenet library on it is not an option.

Quake II is written in ANSI C, and lacks many of the more advanced features a network protocol might have, such as bit-packing. Compression in the Quake II network protocol is achieved through delta compression. This means that there are two different types of object state frames, key frames and delta frames. The data sent in a delta frame is a set of values relative to the data sent in a previous key frame. This is basically how MPEG compression works [vdMea03].

One problem this could cause is if we are unlucky and specifically the key frames happen to be dropped. We would then lose not only the key frame itself but also every delta frame that is based on it. This could be solved by sending keyframes as reliable information, which is the solution used in Quake II. Also, keyframes are sent often enough that dropping one does not matter much, as the error is corrected soon enough. Each delta frame also contains information about which keyframe it is a delta frame to.

2.1.2 DirectPlay

DirectPlay [Mic04] is Microsofts own network library provided as a part of DirectX on the MS Windows and XBox platforms. If a game is to be marketed solely for the PC or XBox markets, relying on the more advanced features of DirectPlay might be an option. Most games of today are primarily targetted for release on Playstation 2, so for DirectPlay to be of use, the programmer would have to duplicate much of its features on that platform manually.

Since DirectPlay is not an option for Playstation 2 games, it has not been considered as a possible base for Gamenet. However, to be compatible with the XBox platform, DirectPlay support on the socket layer would need to be added into Gamenet. Since the part of DirectPlay used directly in simulations is very similar if not identical to UDP/IP, this should hopefully not be too difficult.

The main use of DirectPlay in the context of XBox Live is as a game arbitrer, a central server keeping track of current games in progress and potential participants. It provides chat functionality for those looking for games, and services such as high score charts. The use of these services should not be limited by Gamenet. It should be perfectly possible to create a game that uses all of these services, coupled with use of Gamenet for the actual simulation.

2.1.3 Boost

A main source of inspiration for the design of Gamenet is the Boost libraries [Com04]. Boost was created to complement the Standard Template Library, and includes libraries that not only improve and expand upon existing STL functionality, but also provides things like portable file handling and multithreading, to name a few. The style and structure of Boost will be directly influential in the design of Gamenet, and many of the more advanced uses of templates were learned from studying the Boost libraries.

2.2 Previous Work

Gamenet will primarily be based on the experiences gained when making a simple network game about a year ago. It was extremely simple and there wasn't really much of a network protocol to speak of, but it worked well for the game created then. Making that game resulted in a number of ideas coming to life, and an attempt will be made to realise at least some of these when making Gamenet.

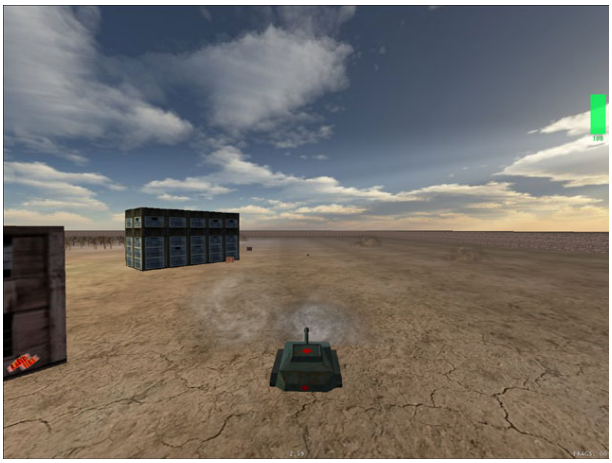

Figure 2.1: An image from the game Panzer, featuring internet multiplayer support.

2.2.1 Panzer

The game Panzer [KGJJ03] was created, by the author and four others, about a year before work started on Gamenet. It is a simple third-person multiplayer game, in which the players control tanks, trying to blow each other up. There are no advanced networking features such as dead reckoning or much compression, but it works fairly well for its purpose, at least on local networks.

Panzer was written to be multi-platform, running on both Linux systems and windows, and hopefully it compiles on other unix variants as well. This was very helpful in seeing what parts of the socket APIs differed between platforms, and where problems might arise in porting the library to a console.

Since the network code was so simple, the only major features that were really interesting was the use of something we called implicit object creation and a simple form of floating number compression. Mainly, this served as an early lesson on how game networking could be done. The implicit object creation meant that no explicit messages were sent to the clients telling them to create or remove objects. They only received a steady stream of state updates on various objects from the server, and if the state of an object that was previously unknown to the client, it was created when its state arrived. This meant that the object type had to be sent along with the state, so that the object was

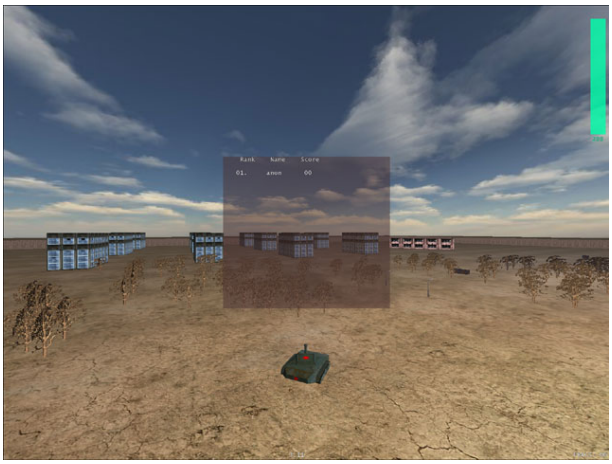

Figure 2.2: Panzer was a fully functional multiplayer game, keeping score and time across machines. However, it did not have a very extensible or flexible network protocol.

correctly constructed. However, it completely removed the need for reliable communication in Panzer, and ensured that if any object existed on the client, its state was the latest to arrive from the server. In contrast, if an object needs a reliable creation message to be created, it might be that several state updates need to be discarded since the object they are for is unknown to the client.

2.2.2 Autobahn

Coldwood Interactive, the company that the library is being developed for, had previously been working on a similar library but abandoned it, due to the programmer responsible leaving the company, and only some pieces of documentation remained. That documentation is a source for some of the concepts in this library, such as the division of information into messages and events, but in general the Gamenet library will not be based upon Autobahn.

2.3 Design choices

Gamenet will be divided into a set of layers, each layer functioning as a wrapper around the previous one. By doing this, low-level functionality is kept in the lower layers, rather than the tendency for each part of a library to use some functionality of every other part. It also makes it easy for a user of the library to cut in at any level, ignoring the overlaying layers if there is no need for the functionality they provide. A traditional object-oriented approach would be to create a single-rooted tree of all classes in the library, but this often leads to heavy intra-dependencies across classes, which needs to be avoided as much as

possible if the library is to be flexible [Str97].

Here follows a list of the basic layers of Gamenet. Each layer may consist of one or more classes, or functions, and provides a specific service to the library or application.

Socket layer: Sockets for UDP and TCP communication, implementation depending on platform.

Stream layer: Provides connection/disconnection protocol and keeps track of current connections.

Message layer: Adds a generic message transmission system, on top of the stream layer.

Node layer: The concept of connected nodes sharing a simulation.

Client - Server layer: Aids in adding ownership and differences between server and client code.

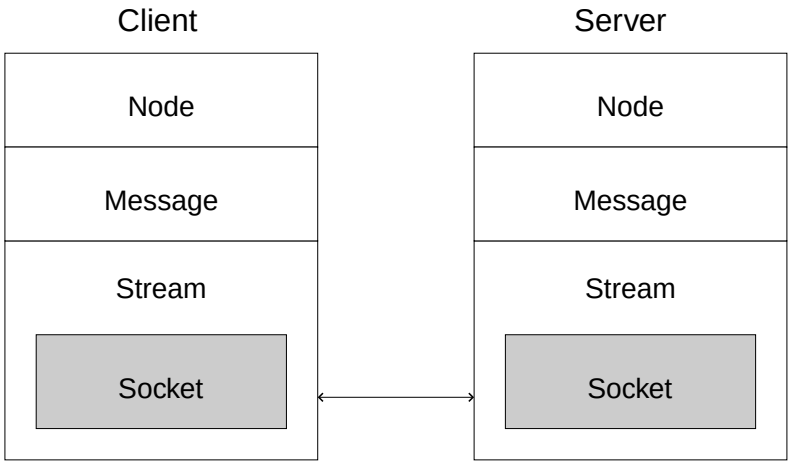

Figure 2.3: Illustration of the separation of Gamenet into distinct layers. Connection and packet transmission is handled in the bottom two layers, and message transmission is done through the Message layer, enabling message coalescence.

This structure can be further expanded toward more specific genres, such as, for example, in the case of a massively multiplayer game [Ent03], facilities for external databases and intra-server communications, or in the case of a peer-to-peer game, an alternative to the *client - server layer*. The most interesting layers for now are the *stream layer*, the *message layer* and the *node layer*.

The stream layer serves to extend the *socket layer* by providing a connection protocol, packet transmission, traffic statistics and other such low-level features. The ability to transmit, retransmit and receive generic messages over the network connection is provided by the message layer. Finally, the node layer introduces the notion of a simulation shared across a network, where each vertex in the network graph is known as a node.

It deals with object serialisation and event transmission/reception. The conceptual link between the node layer and a client/server structure is demonstrated in Figure 2.3. Main focus in the library, at this stage, is placed on the lower four layers, though a very simple client/server distinction is made on top of the node layer.

Chapter 3

Theory and Algorithms

This chapter describes the basic theories of networking in general and game networking in particular, and also introduces the concepts used in the Gamenet library. Many of the algorithms and concepts described in this chapter are implemented in Chapter 4.

A major part of developing a game simulation across a network is dealing with the problem of network latency. Latency is the time it takes for a packet of information to travel from one computer, across the network through any number of other computers, and arrive at another computer.

If the game we are making is slow moving or turn-based such as chess or most turn-based strategy games, latency is not much of a problem. However, if the game is based on quick reflexes and real-time interaction between players, latency can become a huge issue. Most of what you will read in this chapter are different techniques for dealing with latency.

3.1 Networking basics

This section serves as a short introduction to the basic concepts and standards of the IP protocol and related technologies. Gamenet will in principle only utilise UDP/IP, a connectionless transport layer. However, since the library will need to implement some of the features of the TCP/IP protocol, it is useful as a basis of reference to know a little about it as well.

3.1.1 The Transport Layer

In network communication terms, the Transport Layer is the layer of abstraction directly under the application itself. On the Internet today, this layer consists almost solely of the so-called TCP/IP family of protocols, TCP/IP [Pos80a] and UDP/IP [Pos80b].

TCP/IP is a connection-oriented and reliable byte-stream delivery protocol, providing services such as retransmission of dropped packets, ordering and built-in checksums. It is intended for use in applications where the integrity of the communication is more important than the shortest possible time of delivery.

Most internet based multiplayer games of today use the UDP/IP protocol for network communication rather than TCP/IP. UDP/IP is a connectionless and unreliable byte packet delivery protocol, designed to be a low-overhead companion to the bulkier and more feature-complete TCP/IP.

The reason for this is that many of the features of TCP/IP are not needed or even unwanted in the case of real-time communication. If a packet is lost in TCP/IP, retransmission will be carried out automatically, regardless of what the packet contained. Since ordering of packets is enforced, this could mean long stalls in the communication while the receiving end waits for the missing packet to be retransmitted.

In a real-time simulation, such a delay is totally unacceptable. We need to be able to keep sending new data even though some has been lost, and ideally, it should be possible to lose any one bit of information as long as the bulk of the data flow continues.

In practice, some things need to be sent reliably, such as the creation of objects or level change information, but much can be made to be unreliable. For example, instead of sending score updates as difference values when it changes, we send the exact score each time. Then, it does not matter much if one update is lost as long as the next arrives, because the error is quickly corrected.

Datagram structure

In UDP, each packet sent at the client level is also sent, unmodified, as a packet (datagram) by the sockets implementation. This is different from TCP/IP, where packets are conceptually separate from the constant stream of data given at the interface level. The proper size of an UDP packet varies depending on a number of factors, but a common number given as a good maximum size is 1400 bytes, since that is somewhat lower than an IP MTU (Maximum Transmission Unit). However, for internet and long distance use, it might be advisable to keep packet sizes below 512 bytes to avoid IP fragmentation on intermediate routers. Lowering the maximum packet size might also be a good idea for slower connections such as modems, since the actual transmission of a larger packet takes time.

Packet overhead

Since UDP/IP lacks a lot of the reliability features of TCP/IP, it offers a much lower overhead (8 bytes over 40 for TCP) per packet. This might not seem like much, but consider that this is 32 bytes less to transmit every frame. Given a transmit rate of 20 Hz, 32 bytes yields 1.28 kBps given a two-way communication. On a modem connection where the maximum achievable transmit rate might be somewhere around 4-5 kBps, this is quite a reduction in the amount of transmitted data. However, the actual saving will not be quite that large, since we ourselves need to provide some of the services TCP/IP otherwise would have given us. Also, TCP/IP on slower connections often uses header compression, in practice reducing the packet overhead to about 6 bytes. This is, however, usually not possible over the internet but if the game is designed for LAN play only

where the risk of packet loss and stalls is virtually nonexistent, TCP/IP may be a viable alternative.

Blocking vs. Non-blocking

By default, most sockets implementations are blocking. This means that whenever asked to do something (sending, receiving), the call blocks until the procedure is completed. In the case of receiving, this might result in a very long wait until a new packet arrives and can be delivered.

In a game, such behaviour is not desired - the local simulation should be continuing decoupled from the arrival of network packets, and such packets should simply be processed upon arrival. The solution is setting the socket to be non-blocking. This can easily be done on all platforms, and the result is that the receiving procedure returns a soft error if no packet is waiting to be processed. Another way to work around this issue is to receive network messages in a separate thread, but this adds issues of slow context switching and concurrency, and will not be supported in the current implementation of Gamenet. For additional information on sockets, multithreading and the issues that arise in larger simulations, see for example [Ste94], [SP97], [Keg03], [MZ99] and [MZ03].

Combining TCP and UDP

One tempting idea is to combine the use of TCP/IP and UDP/IP, sending reliable data across the TCP channel and unreliable data on the UDP channel. This is however not a viable situation in practice. Because of how TCP/IP behaves under poor connections, for example the use of a 3-second or longer rate-halving retransmission algorithm, it is extremely hard to keep the two channels synchronised. A stall in the TCP pipe will stall the entire simulation, and such stalls can last for several seconds (an eternity in a gaming situation). The resulting maintainance issue of keeping the two channels synchronised is simply not worth the headache, since a game simulation often can get away with a very simplistic retransmission algorithm anyway.

Network Address Translation

Due to the limited availability of IP addresses, something known as *network address translation routers* (NAT boxes) [KE94] has become commonplace. The theory behind a NAT box is simple. It keeps a single, public IP address and multiplexes it to multiple internal computers each having a private IP address. This is done by keeping track of incoming and outgoing traffic and maintaining address translations internally. Thus the internal computers are not aware that they are not directly connected to the internet. This unfortunately has drawbacks in the context of gaming.

The first problem that one encounters is that a server sitting behind a NAT box needs to have its listening port forwarded by the NAT box. This is however not something that can be remedied by the game itself but is a general enough problem with a number

of applications that most people using NAT boxes know how to set them up to redirect specific ports to specific internal clients.

The second problem is that a node cannot associate a given IP address with a specific other node, since multiple nodes may be sharing the same IP. This is not so much a problem, since the nodes will have differing return ports, so a comparison that takes the port into consideration will not suffer from this problem.

The third problem is that some NAT boxes change the port that they are forwarding dynamically. Since conceptually UDP/IP is a connectionless protocol this should not be a problem, but since a game maintains an implied connection this can become a problem if a specific connected client is identified by its source port (which may change). The solution in this case is to assign a node ID to each node in a network simulation, and transmit this as part of the game packet. This is then used instead of source port as client identification.

A simple way to ensure these node IDs are unique is to base them on the private IP address of the node, since this is guaranteed to differ between clients sitting behind the same NAT box. Thus the identification scheme becomes public IP + private IP (often only the lower two bytes of the private IP need to be checked since the higher bytes rarely differ).

3.1.2 Network Topology

The word topology in this case refers to the graph formed by the computer network, where peers are vertices in the graph and the connections between them the edges. Every network has such a topology, and the structure of it determines how data flows through the network [Min01], [GSN94].

Each game simulation forms its own sub-network with its own connection topology and internal communication channels. This topology is not bounded by the physical connections linking the nodes in the simulation. Thus we strive to find a topology that is both easy to maintain and efficient. Which such internal topology is the most appropriate for a certain game differs depending on what kind of game it is, but in general the possible choices can be narrowed down to a couple of common alternatives.

Please note: Whenever the word network is used in the follow chapter, it is used as a synonym for sub-network, referring to the implicit network formed by the game simulation. If the word is used in its general sense or referring to a physical structure, this will be pointed out in the surrounding text.

Peer-to-peer

Peer-to-peer is normally, in the context of game programming, used to refer to a system where each node in the game network has an edge leading to every other node, ensuring the shortest possible route to each of the other nodes involved in the simulation. In a system like this, each node runs a complete simulation of the game state. See Figure 3.1.

Each node maintaining connections to every other node results in the network having

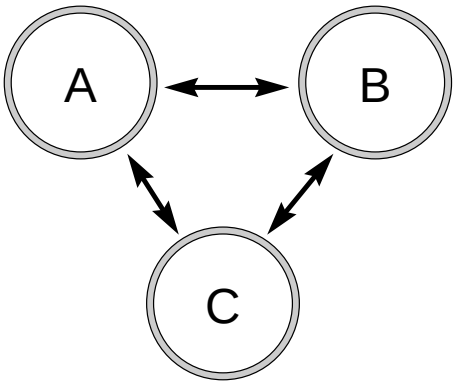

Figure 3.1: In a peer-to-peer topology, each client is connected to every other client. Also known as a *fully connected topology*.

$n^2 - n$ connections. This approach is mostly appropriate for games involving few players. These can for example be real time strategy games or Turn-based games.

Another topology that would fall under this nomenclature would be one where an optimal connection topology between nodes, based on latency, was determined before the game starts, hopefully limiting the number of connections each node needs to maintain.

Peer-to-peer as used when referring to a file sharing network is not quite the same thing as the variants described here. In such a network, each node can maintain a random number of connections to other nodes, and there is no notion of authority. To actually run a real-time game in such a network would be, if not impossible, very hard. Since the route to any other node is arbitrary, the latency within the network would be difficult to predict. Keeping the multiple simulations in synchronisation would be very tricky, and dead reckoning and other such extrapolations depending on educated guesswork would become much harder problems compared to a fully connected network.

Authority presents a problem in a non-hierarchal structure. Some decisions need to be made, not by every single node independently, but by one specific node. Under a client-server structure, there is no problem. The server makes all such decisions. One relatively simple solution would be to, at the first connection of two peers, simply assign server status to the one peer who is contacted by the other (communication is always directed). The problem then becomes, what happens when that peer drops out of the network?

Responsibility not only needs to be redistributed, but every peer in the network needs to be informed about the change in structure. Such a roundtrip decision will require a special sub-protocol. Gamenet, at the moment, lacks such roundtrip decision making support. However, nothing prevents a user of the library from adding this functionality, and thus enabling peer-to-peer topologies to be used.

Having a real physics simulation, e.g. integrating object motion based on forces act-

ing on them, is a problem in a Peer-to-peer network. Integration on two simulations with nearly but not quite the same conditions will result in two completely different end results. The obvious solution is that only one node actually runs the simulation, in which case the strict peer-to-peer model has been abandoned. An alternative would be to use a *clustering* algorithm, i.e., detecting islands of objects interacting and simulating each island on a different peer. Each peer is then responsible for a different part of the simulation.

Another problem that arises with a peer-to-peer structure is that the game network will have big problems crossing network address translation boundaries (See 3.1.1), since each player needs to be able to connect to every other player in the network. Connecting to a computer behind a NAT box is not possible without a prior connection, so two players behind different NAT boxes can never connect to each other.

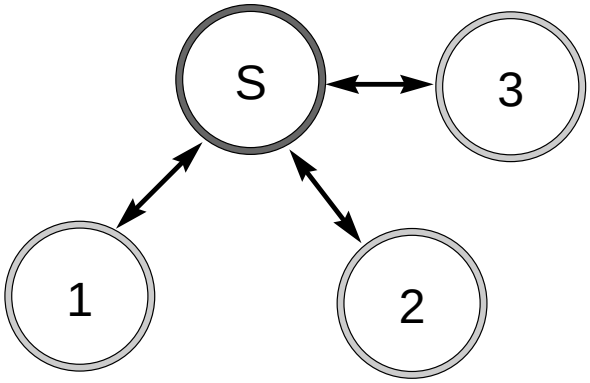

Figure 3.2: In a client-server topology, the server is connected to every client, and each client is connected only to the server. This topology is commonly referred to as a *bus topology*.

Client-Server

In a client-server system, one node is considered the master node or server. All other nodes maintain only a single connection, to the server. The only way for them to communicate is by by sending messages to be routed through the server.

The simulation that runs on the server has authority, and each client simulation is kept in synchronisation only with the server simulation. This limits the problem of synchronisation, with the drawback of having two different types of nodes instead of one. See Figure 3.2.

A client/server structure is the primary consideration of the Gamenet library, and the higher level layers make some assumptions that are founded on a client/server mindset. This does not make peer-to-peer structures strictly impossible, but the library is not geared toward that use.

Having a client/server structure means that different code needs to be written for the client and for the server. However, Gamenet aims to minimise the amount of difference in code between client and server. The synchronisation problems that occur in a peer-to-peer system are avoided in the client/server system since the server is always synchronised with itself, and so any conflict that arises due to drifting simulations is resolved through server authority.

The problem with physics simulation being fundamentally hard to distribute is not so much helped by the client/server-structure, but it simply feels better to allow the simulation to run only on the server in this case. The level designer could possibly mark certain objects as important and others as unimportant, and only the important objects would be simulated on the server. For example, simulating debree from an explosion that is too small to really effect the simulation is wasteful, and we are better off moving that simulation to the clients.

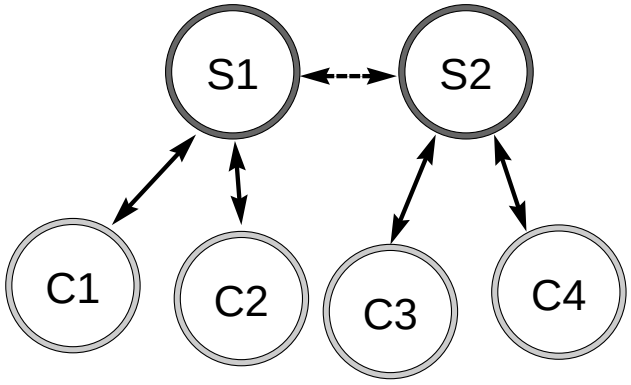

Figure 3.3: An example of a hybrid topology, where two servers both have a number of clients connected, and also share a single connection. A possible arrangement would be a peer-to-peer network of servers, each with any number of clients connected to it.

Hybrid systems

Instead of having a single server to which all clients connect, one could have an array of servers connected and kept in synchronisation in a peer-to-peer network fashion. Clients can connect to any one of these servers, and to the clients the server array can be perceived as one single server. See Figure 3.3.

An extension to such a system would be a self-modifying network where connections between peers are made and released continuously, to balance load and to keep packet travel times as low as possible. The possibilites for gain through such an hybrid system in a computer game simulation are most likely outweighed by the difficulty of maintaining it, however, it offers an attractive theoretical model.

There are many other possible topologies, e.g. the *Grid structure*, where each node

is encouraged to maintain links to four other nearby nodes, though since these are not commonly used they are not further discussed here.

Security considerations

The choice of topology plays into network security issues. In a peer-to-peer environment, there are no trusted peers (anyone can be a cheater). In contrast, in a server-client environment, there is always at least one trusted party, namely the server. Thus it is possible to work around possible cheats and manipulations by taking advantage of this fact, which makes maintaining security easier in a client-server environment.

A problem that would have to be considered would be the possibility of spoofing packet source addresses, so that a malicious user may intercept game traffic and modify it. To secure the protocol against this, a possible solution could be packet encryption using keys known only to the server and client. This session key would have to be transmitted from server to client on connection in a secure manner [oST94].

Security can be reached by a number of ways. First of all, the obvious solution is moving authority to the server. In a pure server-client based implementation, the server is fully in control of the simulation, and the clients are merely dumb terminals, rendering what the server tells them and simply relaying user input back to the server. This pure model quickly becomes cumbersome as the number of clients increases and the computational requirements on the server increases very quickly in areas such as collision detection. Thus, the common solution is keeping as little authority as possible on the server, but to transfer responsibility in areas susceptible to cheating.

A way to allow client authority is by maintaining server-side sanity controls. For example, if the maximum movement speed of a client is known, the server can maintain checks to see if a client is moving impossibly fast.

Client-side prediction

A problem that arises in server-based simulations is that since server authoritisation is needed, a delay is introduced between the occurance of an event and the actual reaction to that event. This can cause large issues in playability and can even make a game completely unplayable.

For example, it is not practical to maintain server-side authority over movement, since any delay in feedback interferes with the ability to react quickly to anything that happens, and makes it impossible to accurately time movement if needed. The common way to work around this issue is for the client to assume that the movement was allowed, act out the movement immediately, and later correct upon any errors when feedback from the server is received. This is referred to as client-side prediction.

One wants this prediction and correction process to be as smooth and invisible to the player as possible. Thus, two things are needed: An accurate prediction process, and a smooth correction process. Accurate prediction is often not a problem, but in the case of a physics simulation it would require stop-starting a local physics simulation, which requires the physics integrator to be very stable.

Smooth correction is also complicated by proper physics simulations. One solution would be to, at the moment of correction, start a simulation process based on the server-given state, and interpolating toward that target from the locally simulated position, hopefully catching up eventually.

3.2 Minimising network traffic

There are two main ways of approaching the minimising of network traffic. The first is to remove as much extraneous data as possible algorithmically on the protocol level, compressing given data in various ways. This can make a great difference, and such approaches are taken on several of the optimisation techniques used in Gamenet.

The other way of minimising network traffic is to avoid transmitting information that does not need to be transmitted. For example, a client should only receive information about things that he can see or hear from the server, and remain unaware of changes to objects very far away until they become relevant. This approach is referred to as spatial mapping in Gamenet, and Section 3.2.7 talks more about this approach.

In this section, a number of approaches to minimising network traffic will be discussed. Most of them are used in the library, but a few of them have not been implemented. This does not mean that it is impossible to utilise these techniques in the library, only that the current implementation does not feature them.

Some of the techniques listed below are lossless, meaning that no information is lost during the encoding. This specifically refers to huffman encoding in particular. However, since most game data is temporally local, it is often safe to use lossy compression algorithms together with some form of regular correction mechanism.

Data compression schemes can also be said to be either static or dynamic. A dynamic compression scheme may change the mapping for a particular source message during the compression process.

3.2.1 Delta compression

One way of minimising network traffic is to split data into two types: key frame and delta information. This is how the MPEG codec achieves its high compression rate [vdMea03], key frames contain full frame information, the delta frames only contain information about the areas that have changed each frame.

Simple delta compression is trivial to achieve, where the user provides two variants of the data: a full key frame, and a partial/compressed intermediate. This approach can become less than trivial however if messages are unreliable and the delta frames are given as offset values from the nearest key frame. If this is the case, a dropped keyframe can result in delta information being applied to the wrong frame, causing erroneous interpolation. This, however, is corrected the next time a keyframe is received correctly, so it is probably not too severe an issue.

Given this simple delta compression scheme, the sender can decide the rate of key frame transmission independently of the client. It is somewhat susceptible to error, since

delta values can be applied to the wrong keyframe. This is however corrected as soon as a new keyframe arrives. An alternative would be to transmit the full values of only those variables that have changed since the last key frame. This approach would lessen the problem of dropped key frames. Less trivial delta compression schemes can also be devised. Common for all of them is that they can be construed as preprocessing steps on top of the Gamenet interface. They are not covered here.

3.2.2 Bit packing

Many values that an object contains do not utilise the full range of integer number, or the full precision of floating point values. Color values can often be stored as a triplet or a quadruplet of byte values rather than float values, and a concept such as health percentage can be stored within a range of 0 - 100. We can use this to make savings in how much data we actually transmit across the network, by allowing the user of the library to tell us the value range of an integer value, or for floating point values, the precision needed. This allows us to pack as many separate values as possible into as few bytes as possible.

In order to serialise an object state or an event into as few bytes as possible, data is packed below the byte barrier via bit iterators. For an example of how this is done in code, see Example 4.7. In Figure 3.4, an example of bit packing is given. Here, four variables are packed into two bytes. Compared to a system with a lowest granularity of a byte, this cuts network traffic needed to transmit these variables in half.

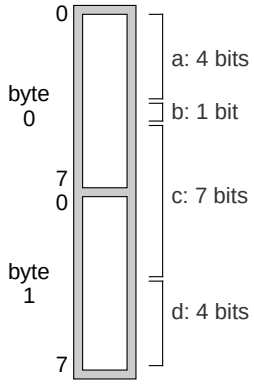

Figure 3.4: Here, four variables are packed into only two bytes, a ratio of $1 : 2$ as compared to storing each value in its own byte.

3.2.3 Arithmetic compression

Arithmetic compression [Bo03a] [Bo03b] is an even more efficient form of sub-byte level compression than bit packing. Given two numbers that can vary between 0 and 2, the

number of bits needed to compress each of them is two bits. This wastes a bit of information, since two bits can actually store numbers that vary between 0 and 3. Arithmetic compression improves upon this situation.

Arithmetic compression works by representing a number by an interval of real numbers, greater than, or equal to, zero, but less than one. As an encoded value becomes longer, the interval needed to represent it becomes smaller and smaller, and the number of bits needed to specify it increases. Arithmetic packing is not used in Gamenet, since it was deemed too complex for very little gain. However, the library permits the addition of new compression techniques on a serialiser level, if one wishes to do so.

3.2.4 Huffman encoding

Huffman encoding [Huf52] is trivial in principle. The basic algoritm works by building a tree with the possible inputs sitting in the leaves, assigning each node in the tree a bit value. Finding the mapping of a certain input involves traversing the tree from the root to the matching input leaf. Thus, it is possible to assign common inputs shorter paths, thereby reducing the amount of information needed to transmit them.

In practice, it might look like this. Given an alphabet of ABC, A being the most common, C being the least common:

A bit encoding: 0

B bit encoding: 10

C bit encoding: 110

Thus, the string AABAC becomes 00100110 encoded. With each 0 and 1 representing one bit, and given ABC as byte values, the amount of information needed to encode the string is as follows:

Uncompressed: 5 bytes

Compressed: 1 byte

Huffman encoding has not yet been fully implemented in Gamenet.

3.2.5 Interpolation

Using inter-/extrapolation, the delta information between key frames can be estimated client-side and thus is not transmitted. How accurate the estimation is depends both on the nature of the motion in question, and also on how great the latency of communication is with the server.

For example, if an object is known to always move at a linear rate, the actual movement can be performed as client-side interpolation between server-transmitted target positions. If the server can guess where the object will be in the near future, it can transmit that value to the client which can then keep up with the objects motion on the server. Interpolation is only suitable for continuous values.

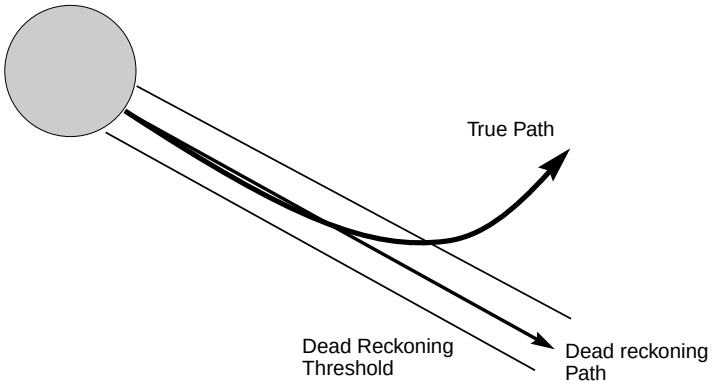

Figure 3.5: Shows dead reckoning, as performed on the server. The server keeps track of both the actual motion of the object, together with the simulated movement of the client from the last server-transmitted position. If the client position diverges too far from the actual position, another update is transmitted.

3.2.6 Dead reckoning

Dead reckoning [Aro97] is a more complete form of extrapolation (See Section 3.2.5).

The basic idea is that there is a set of extrapolation algorithms agreed upon by all nodes in the network, and also an agreement of how far the extrapolation is to be allowed to drift from reality before being corrected.

As an example of dead reckoning, consider a world simulation where a given object is owned, or controlled, by one specific node. As the object is created and distributed, both the owner of the object and all other nodes begin applying the previously agreed-upon extrapolation algorithm on its motion in the world. Thus, the object will continue to move according to the extrapolation algorithm,even if no more position values are sent out by the owner (see Figure 3.5). When the owner deems that the extrapolation is becoming too far off from the true value, it transmits a new orientation update to the other nodes, who then proceed to extrapolate based on the new information (see Figure 3.6).

Further information to aid the extrapolation can be transmitted along with the key frame information. For example, both the current position and the current velocity are usually transmitted, so that further extrapolation can use the velocity known at that point to guess at its next position.

This allows new position information to be transmitted as seldom as possible, since the server is constantly keeping track of where the client probably think the value will go.

A problem that arises with dead reckoning is if the server and the client are implemented on different platforms (for example, a server running under Windows PC and a client running on an XBox), and the different platforms have varying floating point formats. This problem can even arise on similar platforms, since the order of execution can influence final results to drift apart.

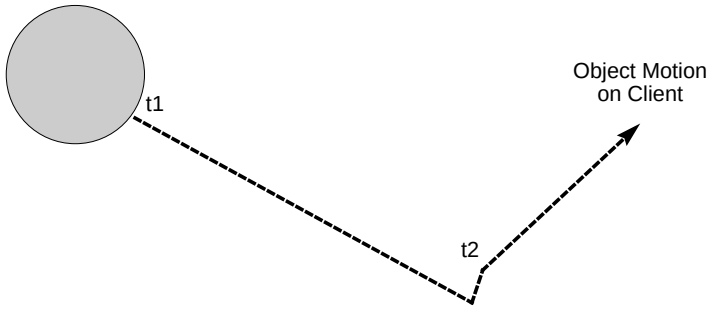

Figure 3.6: Illustrates how the object motion in the preceding example can be perceived on the client, with a simple linear extrapolation based on position, velocity and direction.

If the simulation is allowed to run for a longer period of time, for example if something is moving very linearly and closely following the extrapolation. In this case, the server extrapolation can end up diverging drastically from the client extrapolation.

To combat this, a maximum extrapolation time can be established, so that new position information is given at least at a minimal rate.

3.2.7 Spatial mapping

Given that we are trying to simulate a geographic world with positions of some dimensionality, it is possible to assign positions a priority based on their proximity to the local position. For example, a player of a first-person shooter game will be more interested in the movement of other players close to him/her rather than players far away.

In Figure 3.7, two levels of spatial mapping are used. First, the node (indicated as P) has a local bounding sphere, inside which is found object D. Also, a sector based scheme is in use, causing object C to also be considered of interest to the node. Objects A and B are completely outside the area, and are thus not transmitted. A spatial mapping scheme also helps against client-side cheating, since, if an object is not updated when not visible, making walls transparent and other similar cheats are made less efficient.

The adoption of a spatial mapping scheme has two major practical implications:

1. A spatial order is enforced upon the simulation - it has to connect in a geometrical way for a spatial mapping to make sense. Every object in the world either needs to have a position or is considered ubiquitous for spatial mapping purposes, ie. some alternative scheme is needed to limit the traffic those objects take up.

2. Every node needs an associated sphere or area that it is considered 'inside'. Just exactly what it is that the node is *in* varies with the world representation, it can refer to a Binary Space Partition or simply a vector of bounding spheres. What matters is that the node itself is considered an object in the world, and collision

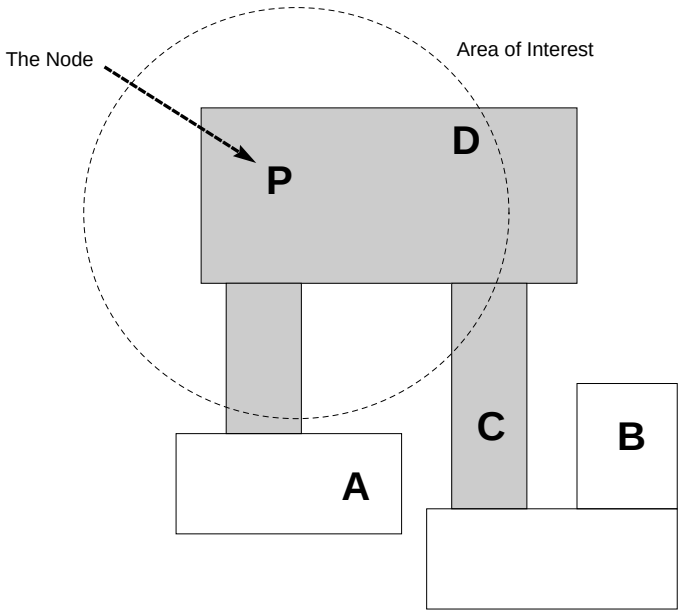

Figure 3.7: Shows two distinct levels of spatial positioning. The local node position has a radius in which objects are considered relevant, but it also keeps track of the room it is currently inside, and considers any objects residing in the current, or an adjacent, room, as interesting.

detection is performed against the node to determine whether information is to be shared with it.

In a simulation with a large number of objects, a spatial mapping scheme is essential for minimising network traffic.

Gamenet supports spatial mapping by providing a relevance callback. This is called whenever an object is to be transmitted, and should juxtaposition the target local position with the position to be transmitted, returning the resultant priority value. This value can be used to determine the set of object states to transmit each frame.

3.3 Network Protocol

A common protocol needs to be developed for Gamenet. A layered approach will be taken to the protocol, where each stage of the packet building process wraps previous data within its own. This is similar to how the IP protocol is structured.

Rather than focusing on making what is transmitted as small as possible, much can be gained from finding ways of transmitting information as seldom as possible. By attaching a frame number to each outgoing packet and returning the last received frame number when transmitting back to a remote node, each node can keep track of the latest information known to have arrived. If this information has not changed, it does not have

to be retransmitted. This is seen most typically as delta compression inside the simulation, but is also important on the lower levels of the protocol.

The data transmitted via the Gamenet protocol consists almost entirely of something called *messages*, so it is optimised towards the transmission of such messages. Some information that is stored locally in each message before transmission, is broken out and stored once per transmitted packet when sending over the network, to minimise duplication of data.

Since most of the non-message information transmitted is sent very rarely and not in bulk (such as connection messages, error messages and so on), this information can be transmitted in the form of text strings. In case the need for additional security arises, a simple move to binary information would not be sufficient in either case, and so an encryption scheme could work equally well over text-based data as binary. In any case, this is something that can be easily changed if needed. The actual network protocol developed for Gamenet is described in Section 4.4.2.

3.4 State and Events

There are mainly two types of messages that needs to be transmitted in a game simulation; the continuous change of variables such as position, age or speed of an object in the game world, and secondly the communication of events that occur, such as a client firing a rocket or a wizard exploding. The first, continuous, type of information will be referred to as object state. The second, discrete, type of information will be referred to as events.

Object state has some interesting properties that can be exploited in a simulation. First, being continuous in nature, state such as position and velocity can be interpolated or extrapolated. This allows us to transmit less data, since we only need to correct the interpolation or extrapolation algorithm of the other side when it starts to veer off target. Secondly, ordering is only relevant in the sense that more recent information overrides less recent such.

Events on the other hand, being discrete, can not be predicted or interpolated. They happen once, and thus their delivery needs to be ensured via acknowledgement algorithms.

Each type of message has its own manager. The object manager takes care of traversing the object tree, performing spatial mapping and keeping track of what objects need to be syncronised. All object creation needs to go through the object manager, so that it can keep track of appearing and disappearing objects in the simulation.

The event manager gathers events to be transmitted, and delivers events received from other network nodes.

3.4.1 Object manager

The object manager has to not only gather and compress information from the individual objects, but also has to decide what objects are important to send. It should keep track of

both the last known packet to arrive and also what data is important that it delivers often. For example, player positional information should take precedence over static object positions. Much of this comes automatically if delta compression is used, since objects that rarely change will be able to remain in the non key frame form longer. However, a separate way to give certain objects a higher priority can further help the manager.

The object manager should also keep track of what object state it has delivered earlier, so that state that arrives out-of-order can be discarded. For maximum efficiency, each object itself should keep track of the latest frame to arrive, and the latest frame it sent.

The object manager needs to be able to traverse the world scene graph, and must be able to perform searches for objects based on spatial location. It should also be able to deliver given data to specific objects, although this can be solved with a simple traversal of the object structure. Preferrably, the object manager should be able to take advantage of whatever spatial partitioning schemes the game uses for graphical partitioning of scenes, without restricting or imposing upon that structure in any way.

3.4.2 Event manager

The event manager is, in contrast to the object manager, a lot simpler. It only has to be able to send messages and deliver them when they arrive. Each message should have both a sender ID and a target ID attached, and the event manager needs to be able to map these IDs to objects locally. A message consists of a message ID, an amount of object data (unknown in quantity) and a number of transmission flags controlling things like reliability and ordering.

However, since every object also needs to be able to discern from where a given message comes, a network-independent scheme of object identification is already in place, and the event manager should be able to use this system. Thus, the event manager should be able to have a singular point of delivery, simulation-specific, which then takes care of delivering the messages to their final destinations.

3.5 Security

The protocol should be constructed with network and application safety in mind. However, since network security is an entire research area of its own, it will be glossed over for now, both in Gamenet and in this thesis.

3.5.1 Encryption

Encryption in itself serves the purpose of making it harder for someone to intercept packets and modify them, something done for example when making cheats or trying to reverse-engineer the protocol. One form of encryption is compression, since data becomes harder to interpret when compressed as much as possible. For example, a huffman encoding for strings would be an easy way to prevent immediate modifications of data en-route. For the most basic purposes, however, a simple checksum will serve.

3.5.2 Client identification

A server needs to be able to identify received packets as genuinely coming from where they are supposed to be coming. UDP/IP being connectionless makes this a bit harder, since a server needs to receive packets from anywhere and has to examine the contents of the packet to discover from where it came. To achieve an additional bit of safety, both the sender IP and the node ID should be checksummed together with the rest of the packet contents, so that if any of them changes, it can be detected by the server.

Chapter 4

The Gamenet Library

This chapter will describe in more detail the interfaces and classes designed for Gamenet. The network protocol used by Gamenet will be shown in further depth, and some examples on how to use the library are given.

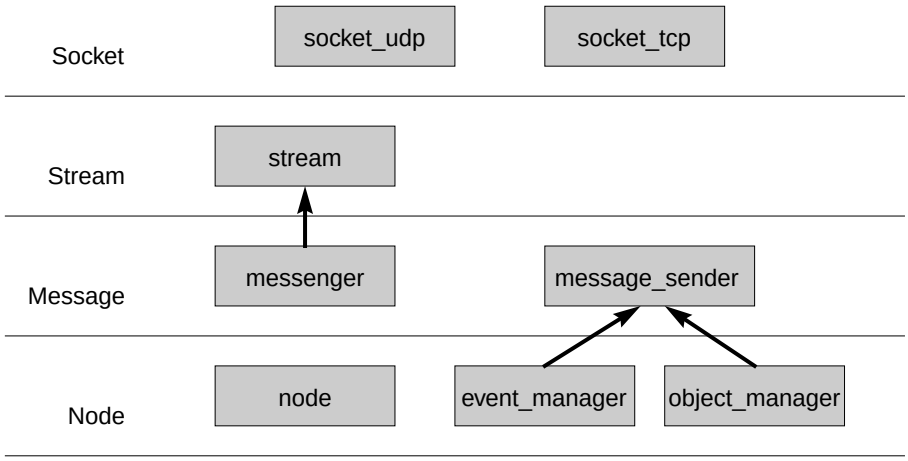

Figure 4.1: An overview of the major classes in each layer. The message_sender class is an interface that can be inherited if there is a desire to create additional/alternative managers. Other important but unconnected classes are the peer class, for information on other nodes in the network, and the serializer class, for writing data into a bit stream.

A basic overview of the major classes in Gamenet is given in Figure 4.1. The node class is the main entry point of the library, and usage of Gamenet begins by creating an instance of node. This object contains a messenger, an event_manager, an object_manager and a couple of other, less central managers. It is through the node object that connection/disconnection takes place, and it is the node that keeps track of all managers currently in use.

4.1 Layers

Just like the networking structure used on the internet today, the Gamenet library is structured into layers of abstraction. On the lowest level is the layer responsible for abstracting the basic network services of the current machine used (MS Windows, PlayStation 2 or others). This layer is then used by another layer above it. The library is split into a number of such layers of abstraction, described in this section.

Each level of abstraction in the library can be tapped into and used by an application, since a class on a lower layer can never use or call into a class on a higher layer. This will enable a very flexible use of the library, since the necessary restrictions in flexibility brought on by a higher level of abstraction becomes less relevant when one can hook in on any level of the library.

For example, to make a simple `tftp`[Sol92] or `wget`[Fou03] application, one could use only the socket_udp and socket_tcp classes. In fact, it is quite possible to base a web browser or FTP client on the socket layer of Gamenet.

The way this is achieved is in part through use of object oriented concepts such as inheritance, but also through use of C++ templates, combining runtime efficiency with code reuse.

4.1.1 Socket layer

This is the lowest layer, providing the concept of data delivery to the higher layers. This layer can both receive data and transmit data under a given set of limitations. Currently, the socket layer consists primarily of three classes. address is a simple IP address wrapper, used by socket_udp and socket_tcp, the actual socket classes. The TCP socket has a notion of a connection, but the UDP socket simply provides the ability to send and receive packets of data.

If the library was to be ported to other platforms, such as Playstation 2, it would be in the socket layer that changes and additions would have to be made. The socket layer is the layer that aims to hide the underlying platform, and provide a generic socket implementation that the rest of the library can use.

4.1.2 Stream layer

The stream layer extends the socket layer in order to provide the notion of a connection. It keeps track of all the other nodes it is connected to, and it is the stream layer that handles establishing game connections, and it provides some reliability for notification messages (see Section 4.4.2).

In essence, the stream layer is the class stream, which is the one that does all of the above-mentioned things. It also provides the concept of a packet, and assigns the basic header values in the packet.

If encryption were to be added to Gamenet, the best layer to insert it into would be the stream layer, since it is that layer that constructs the actual packets sent between clients. It is the stream layer that is responsible for throwing away suspicious or broken packets.

Only one socket and one port is used for all game communication. Peer nodes are disambiguated through the use of unique identifiers sent in every packet. The main reason for constructing it in this way rather than accepting connections on different ports for different clients, is that games often use obscure and high port numbers, and anyone who wants to play the game behind a firewall needs to set up the firewall to allow the packets belonging to the game to pass through. This is vastly simplified if the game only uses one single, predetermined port.

The stream gathers receives packets at the highest possible rate, but it only sends at a lower, steady rate. This is to ensure an even flow of data, and by varying the rate that packets are sent, the simulation can be adapted for faster or slower connections.

4.1.3 Message layer

The message layer builds directly on top of the stream layer. Gamenet differentiates between *packets*, chunks of data sent from one node to another, and *messages*, as in a single piece of information sent from one node to the other. A packet consists of a number of messages packed together and sent across the network, to be unpacked, and the messages delivered, when it arrives.

The message layer and the messenger in particular provides this messaging service. The messenger gathers messages from a number of message sources, and decides how many and which to transfer with each packet. It handles resending messages that are lost, and holding messages until they are in order before delivering.

4.1.4 Node layer

The node is a central concept in Gamenet. Every participant in a network is considered a node, and to have a proper simulation distributed you need the node layer. It provides a set of *managers*, classes that each fill some task in managing the simulation. There are two basic things that need to be distributed between clients, and there are two managers in the node layer that handles these things: The *Object Manager*, and the *Event Manager*.

Two other basic managers are provided with Gamenet, the *Text Manager*, and the *Data Manager*. They fill minor functions in that they are not central to the simulation, but they are important non-the-less. Other message sources can be added by the application, and in fact the application may replace these managers completely if it so wishes.

Object Manager

Objects are items that are persistent in the world, existing over a longer period of time. These objects have variables that continously change, and the job of the object manager is to distribute these changes over the network.

In Gamenet, the object manager is capable of creating objects on its own, without relying on creation events to be distributed. This is accomplished by sending the type of each object with it as it is distributed. Since this takes a bit of extra space in the packets,

there are ways to accomplishing the goal of knowing object type without attaching it to each and every message sent.

```
void register_types(node& the_node)
{
    // do this once per object type, and in the
    // same order on all peers
    the_node.objects.register_type<item>();
    the_node.objects.register_type<actor>();
    the_node.objects.register_type<trigger>();
}
```

Example 4.1: Shows how to register new object types with the object manager. Note that all types registered must inherit the object base class. Also note that the type parameter is given as a template parameter, not a regular parameter. Naturally, regular parameters cannot be types.

One way would be to group objects so that all objects of the same type are placed after each other in the packet. If this is done, the type can be placed once before each group rather than individually per object. Another way is to use the fact that we get acknowledgement of the packets other nodes receive from us. If we know that a certain client has received this object before, we can throw out the type in that object. And if that client knows that at the time of transmission, we knew that they had gotten that object, we do not even have to send a placeholder bit to tell us that the type has been removed. In this case, the cost of implementing the optimisation far outweighs the benefits, so this piece of optimisation should only be added if it turns out to be necessary for playability.

```
struct object_list
{
    virtual ~object_list () {}
    virtual object* first_object () = 0;
    virtual object* next_object() = 0;
    virtual object* find_object(object_id id) = 0;
    virtual void add_object(object*) = 0;
};
```

Example 4.2: The object list interface, which needs to be implemented by the library user and a pointer given to the object manager, so that it can traverse the object list, regardless of how it is actually stored.

Any object types to be distributed via the object manager need to inherit the object base class. This class provides some basic information to the object manager, such as object ID and type ID. It also contains the definition of the callback used to read and write the object to a message. Each object type that inherits from object also has to be

registered with the object manager. How this is done is shown in Example 4.1. In order to use the object manager, the library user also has to provide a minimal set of callbacks the manager can use to traverse the objects in the world. This set of callbacks is given in the virtual class object_list . The members of this interface are given in Example 4.2.

The operations are generally those expected, though there is no remove_object, or equivalent, operation in the interface. The reason for this omission is that object life often has to continue in a zombie-state after the actual removal of an object, since other objects may still hold references to it. Not until those references are resolved can the actual object be removed from the world. The truth of this depends on the simulation, but the network library should not place any restrictions on object lifetime. Thus, object removal is done by the application at its leisure, even though object creation may happen as messages are received from the network.

Event Manager

```
void main()
{
    node n;
    event e("my event");

    e.target  = 0;  // send to all peers
    e. reliable  = true;  // has to arrive
    e.ordered = false;  // does not have to arrive in order

    //  fill  e.data with data to send as parameters

    n.events.send(&e);
}
```

Example 4.3: Demonstrates how to send events with the event manager. Setting the target parameter of the event structure to 0 results in the event being distributed to all peers.

Events are things that happen once and changes the simulation in some way, like for example an explosion or the firing of a gun. Events are cumbersome, since if they happen only once, they need to be sent reliably to other nodes, or else they might not happen at all. This means that retransmissions may take place, and retransmission is by its very nature against the goal of real-time interaction. Thus, it is a good idea to try to have as few events as possible, and to the highest possible extent model things in the world so that they are objects with continuous state rather than events.

For the few things that do need to be events, the Event Manager takes care of distributing them. Each possible event is assigned a unique ID, and event handlers can register themselves with the manager to receive notifications when events arrive.

```
bool my_event(event* e)
{
    if (e->name == event_name("my event"))
    {
        // do something with the event
        // e.source == ID of sender
    }
}

void main()
{
    node n;
    hcallback ehandle = n.events.connect("my event", my_event);

    // receive events, sent via calls to
    // send("my event", event_data);
    // on some other node in the network

    n.events.disconnect(ehandle);
}
```

Example 4.4: Using event_manager::connect() to connect an event receiver function and receiving events. Note that the receiver function can also be a member function, through use of STL or Boost facilites for coupling object references to functors.

Sending events can be done by anyone, given that they know the name of the event. This name is given by the sender as a string, and this string is then hashed into an identifying number by the event manager. This hash can be retrieved by the application through the function uint32 event_name(**const** string& name).

The call used to send a packet is the member function **void** event_manager::send(event * e). The parameter is an event structure, filled out by the sender. It contains the name of the event, some flags and parameters to the event manager (reliability, priority etc.) and the actual data to be transmitted.

Receiving events is done with the connect() member function. This function is very flexible, and can take both regular functions and member functions as parameters, with the use of a member function connector such as std::mem_fun_ref() [Str97] or boost::bind() [Com04]. This flexibility is essential in minimising the intrusiveness of the library. An example of sending an event is given in Example 4.3, and in Example 4.4, the corresponding example of connecting and receiving events is given.

Text Manager

The Text manager is the simplest manager of them all. All it does is send and receive strings of text between nodes. Target peer ID can be given to only send messages to a specific peer. Message relaying, the ability for peers to send messages through each other, has not been implemented, but for example in the case of two clients connected to a server attempting communication, such a relaying scheme would be necessary.

Data Manager

The Data manager distributes bigger chunks of data over a period of time, by splitting the data into smaller chunks and sending these as regular messages. It can be used to send, for example, textures or voice communication data.

4.1.5 Game layer

The Game layer, or Application layer, is the layer of abstraction provided by the game engine using the library. The node class, or for that manner, any other part of Gamenet, does not distinguish between server or client. It is up to the Game layer to make this distinction if it desires.

4.2 Concepts

The idea of concepts comes from C++, where it is used to denote general ideas and structures that shape the code. Not necessarily in a direct way as function names or class layouts, but as generic ideas that are repeated often. Examples of this in C++ is the concept of an *iterator*, a class that enables traversal over complex data structures. Gamenet uses a number of such concepts, and some of them are described in further detail here.

4.2.1 Bit iterator

The bit iterator concept used in Gamenet is a specialisation of the Iterator concept used in the Standard Template Library [Str97]. Instead of writing a special data vector class for storing bit-level information, a class referred to as a *bit iterator* is used to write on a bit-by-bit basis to a generic byte stream. Through the use of templates, the bit iterator can be generic enough to operate both on regular C arrays and STL vectors, or any other such data structure that may be used in the game. This enables the bit iterators to take advantage of pooling and specialised memory management. To get the entire library to use a different kind of allocation scheme is currently not possible, but through the use of the iterator concept for bit level access, the implementation of such capabilities is made quite a bit easier should one desire it.

The iterators in Gamenet work somewhat like STL iterators, but with some crucial differences. The iterator is based around the concept of a buffer Proxy (see Section 4.2.2). The proxy concept says that, to use the iterator with a certain type of buffer, a proxy

```
uint8 buffer [1];

bi_pointer i = bi_pointer_iterator (buffer , 1) ;

i . write (5, 3) ; // write 3 bits containing the number 5
i . write (30, 5) ; // write 5 bits containing the number 30
i . flush () ; // finalise writing
i . rewind() ;

unsigned int a, b;
a = i . read(3) ; // read the number 5
b = i . read(5) ; // read the number 30

// this prints out "5 = 5"
cout << a << " = 5" << endl;
```

Example 4.5: Using the bi_pointer iterator class to both read and write bit-aligned data to and from a regular C array. Normally, bit writing is done through the serializer interface rather than directly through the bit iterators, since the bit iterators only have interfaces for writing unsigned integer data.

class needs to be implemented, to provide the buffer an interface that the iterator can understand.

In Example 4.5, bi_pointer is **typedef** to the type bit_iterator <bi_pointer_proxy>. bi_pointer_iterator () is a function that aids in the construction of a bi_pointer. As the example shows, to construct and use a bit iterator for regular C arrays, we need to provide the size of the array as a constructor parameter. This is so that the pointer proxy can throw an exception if an attempt is made to write outside the buffer. The number 5 has a bit pattern of 101. This bit pattern can be fitted into only three bits, but by default it is encoded using eight bits or more, since the byte is the lowest granularity commonly used. If we supply the number of bits to use as a parameter, we can ensure that we always use only the fewest number of bits needed to encode a given number. Thus, in the example, only three bits are used to store the number five. We can then use the remaining five bits to store another number, and extract it later on.

In Example 4.6, bi_vector is the bit iterator adapted to the standard library class std :: vector<uint8>. The major benefit of this iterator over the regular pointer iterator is that it can expand the vector when it becomes full, and thus no maximum space has to be pre-allocated or given as parameter.

4.2.2 Proxy

A proxy class is a class that works as a new interface to a class, making it possible to combine classes whose interfaces differ. In Gamenet, the proxy concept is used to adapt different buffer types to the bit iterator, so that the bit iterator can be used on many types

```
std::vector<uint8> v;

bi_vector i = bi_vector_iterator(v);

i.write(15, 4);
i.write(400, 9);
i.flush();
i.rewind();

unsigned int a, b;
a = i.read(4); // read the number 15
b = i.read(9); // read the number 400

// this prints out "15 = 15"
cout << a << " = 15" << endl;
```

Example 4.6: Shows how to use the bi_vector bit iterator to both read and write bit-aligned data. Since the vector can expand if necessary, no size parameter has to be given to the iterator.

of buffers. Currently, there are proxy implementations for `std::vector<uint8>` and regular C arrays, but, for example, if using a different library which provides new container classes, new proxy classes can be written to enable bit iteration over those containers as well.

4.2.3 Messages

A message is at the basic level an ID with an assigned amount of data. What the data is depends on the ID. Both sides of a communication channel need to agree on how to handle a given message, since the format of the data may vary. They also need to agree on how to assign message IDs, since conflicts may otherwise arise. In Gamenet, there are three significant ways of assigning ID numbers, used in different places.

In the stream layer, each connecting client to a specific node is given an ID by the node it connects to. This means that in a peer-to-peer game using Gamenet, everyone would still have to connect to a single computer at first, in order to assign client IDs properly. In the message layer, each message is assigned an id when a `message_id` object is created, usually as a data member in a message sender class. Once a message id has been assigned, it is never free to use by anyone else. Finally, in the event manager, event ID's are given through a hash function, so that events are given proper names which are then hashed into numbers to use as IDs. This means that there is a possibility that two different events get the same ID if their strings hash to the same number, although with a 32 bit hash, the theoretical risk of this happening is 1 in 2^{32}.

4.2.4 Serialisation

Serialisation in Gamenet denotes reading and writing the state of an object to and from a bit stream. This is done with bit iterators, wrapped in a serialisation interface that provides further functionality and a simpler interface.

By using a combination of function operators and reference variables, the two problems of reading and writing state to/from the stream can be unified in a single interface. The same member function is called both during a read operation and a write operation. The benefits are great compared to a system with separate read and write functions, since much code duplication is avoided, and reading and writing are only done in separate code paths when needed. Thus a big source of possible errors is avoided.

Example 4.7 shows how the serialisation of one variable can influence the serialisation of others. By inserting boolean values into the serialisation stream like this, the need for a common bitfield telling what variables have been serialised is removed, in favour of a more intuitive approach. Each bit can be placed next to the variable it governs rather than in a separate place.

```
void my_object::net_sync(serializer& sync)
{
    if (sync(my_bool))
        sync(my_int, bounds(0, 100));

    sync.vec<3, 16>(my_pos);

    sync.bits<12>(my_float, bounds(0.f, 10.f));
}
```

Example 4.7: Using the serializer class passed to the net_sync() member function of distributed objects. The function operator has been overloaded for the serializer class to allow a simple and readable syntax for serialising variables. The vec<C, B> and bits<B> member functions write arrays of bit-packed data and bit-packed floating point values, respectively.

The function operator for boolean variables returns the variable it was given as parameter, so it can be used in conditional statements, to improve readability. The `vec<N>` member template of the `serializer` class can be used to synchronise arrays of data.

4.2.5 Delta compression

Delta compression is accomplished through a number of schemes. One simple way to do delta compression takes advantage of the bit packing procedure to embed boolean values into the bit stream.

The delta compression in this case works by, on the server, only transmitting actual data if the data has changed since the last packet known to have been delivered. A way

```
void my_object::net_sync(serializer& sync)
{
    if (difference(current_data, last_packet_data) > threshold_value)
    {
        sync(true);
        sync(current_data);
        last_packet_data = current_data;
    }
    else
    {
        sync(false);
    }
}
```

Example 4.8: The server of delta-compressed data checks the current value against the stored last transmitted value, and if the difference is too great, transmits a new value. If no value is transmitted, a boolean variable (a single bit) is transmitted to notify the receiver of this fact.

of doing this is shown in Example 4.8.

The client implementation is trivial, since the actual work is done server-side. All the client has to do is check the transmitted boolean value, and if set receive the updated data. This is shown in Example 4.9.

```
void my_object::net_sync(serializer& sync)
{
    bool tmp = false;
    if (sync(tmp))
        sync(current_data);
}
```

Example 4.9: The receiver of delta-compressed data only has to check the value of the transmitted boolean to see if a new instance of the variable has been submitted.

Normally, these two steps are combined into one procedure, using the serializer member function **bool** serializer :: writing ().

4.3 Transmit cycle

A network library host has an internal transmit cycle, which is the rate at which it transmits messages to other nodes connected to it. This rate can be changed during the course of the game, and a theoretic possibility would be to adjust this rate depending on the

quality of the current connection. On each network update cycle, the node cycles through all objects known to it, gathering state information. It is up to the objects themselves to dynamically vary the quantity of state that is transmitted (or if it is to be transmitted at all).

```
name              bytes     description
< type/flags >        1     The type of packet + packet flags.
< source id >         2     ID number of the transmitting node.
< notify id >         1     ID number of the notification.
< string >            n     Null-terminated parameter string.
```

Figure 4.2: The NOTIFY packet has a minimal header, with no optional fields.

4.4 Network protocol details

The protocol is divided into two separate types of packets that can be sent.

The first type is the NOTIFY packet format. This type of packet is used to connect to a node or getting more information about a node without sharing the simulation. It can also be used to send messages between nodes.

The NOTIFY packet format is described in detail in Figure 4.2. Since the Source ID is not necessarily known (the other node may not be part of the current simulation), it may be 0. The possible NOTIFY messages and their parameters are described in Section 4.4.2.

The second type of packet is the DATA packet format. This is used to transmit game simulation messages such as events or object state. The checksum is currently calculated only over the messages transmitted, a more secure variant would also include the header itself in the checksum calculation.

Many of the parameters in the DATA header are optional (as denoted by the use of square brackets in Figure 4.3), and their inclusion depends on a set of five possible flags given in the first byte of the packet.

RELIABLE If set, the packet contains reliable messages and thus a sequence number is included.

ORDERED Set if the packet contains ordered messages.

UNRELIABLE Set if the packet contains unreliable messages.

ACK If set, the packet includes the sequence number of the last successfully received packet.

CHECKSUM If set, the packet has a checksum.

PING If set, the receiver returns a small ping packet as quickly as it can, to measure roundtrip time. This ping value is not entirely accurate since packets are received

at a fixed rate, and thus the ping value will be a multiple of this rate. However, it gives a notion of link quality.

Every stream has an associated transmit id, which it sends in the header when communicating to others. This is used as an identification method rather than the source IP address, so that we can disambiguate different nodes residing behind a common proxy server.

```
name              bytes     description
< type/flags >        1     The type of packet + packet flags.
[ checksum ]          2     A checksum of the packet data.
< source id >         2     ID number of the transmitting node.
[ sequence nr ]       2     Only sent with reliable messages.
[ ack nr ]            2     Last received sequenced packet.
< messages >          n     The messages sent in this packet.
```

Figure 4.3: The DATA packet header has a number of optional fields, depending on flags set in the first byte. No sequence number must be transmitted if a packet contains no reliable data, since such a packet might be dropped and never resent, but the receiver would have no way of knowing that the dropped packet did not contain reliable messages.

4.4.1 The message format

Just about all information that is transmitted across the network is done so in the form of messages. Both events, things that happen only once, and object state updates are sent as messages. The core library knows of nothing at a higher level of abstraction than a message. It decides what messages to transmit each frame based only on the information given in the message structure and the message header. Thus events and state updates can be treated in the same way, and deciding what information to discard or delay when a packet becomes full becomes a single unified task instead of having to do the same thing for two different types of data. Also, more freedom is given to the application. It is fully possible to create new types of messages, or use the message transmission system for other purposes than game communication.

Some of the information stored in the message structure is duplicated in the stream header. When transmitting the message, this information is not included, but the information stored in the stream header is used to reconstruct this information on the other side.

4.4.2 Notification protocol

In order to establish a connection between two nodes, a certain amount of negotiation is required. This negotiation takes place under a set of rules governed by the notification

protocol. The NOTIFY packet type consists of a message type denoting what kind of notification this is, and a parameter string containing whatever information might be sent together with the notification. The different predefined message types are described below. However, an application is free to add new internal types as it pleases, or use the predefined message types in other ways than the described.

CONNECT Sent by the connecting node. The parameter string has the format "<client ID> <time>", where time is given in milliseconds.

ACCEPT This is the reply given to CONNECT, if the connection is accepted. The parameter string has the format "<client ID> <new client ID> <server ID> <time> <version>". The reason for two client ID numbers being sent is because the ID the client had assigned to itself might be clashing with the ID of some other client, and so the connectee assigns the connecting node a new ID number for the simulation.

ACCEPT2 This message is sent back from the client in response to the ACCEPT message, in acknowledgement of the new client ID, and also to complete the time roundtrip for both parties so that they can both guess at the current network latency. The parameter string is "<client ID> <time>".

ERROR Some error has occurred remotely. A text message detailing the error can be given. This notification is also used internally when something goes wrong in the socket and stream layers, to notify the application of what has happened. This error does not have to be fatal.

MESSAGE A generic text message sent to a node, in order to be displayed to the user. The sender does not have to be part of the simulation. The text manager intercepts MESSAGE notifications and puts them into its incoming queue, making retrieval of chat messages from connected clients and unconnected clients a unified task. Unconnected clients are identified by the reserved client id O.

WHO This can be sent to a node, to get a list of its incoming connections. No parameter is given.

WHO2 Sent in reply to the WHO message, this is a list of connected clients. If the client has an associated textual name, that is given rather than the client ID.

INFO Sent to a node to get some form of information. Every node has a set of name-value pairs associated to it, and by sending INFO-messages, these name-value pairs can be queried. The parameter is the name of the variable requested.

INFO2 Sent in response to the INFO-message, the parameter of this notification is "<name> <value>"

DISCONNECT A node can send this to another node in order to close the current connection in a graceful way. A message can be given as parameter.

CONNECT_FAILED This is primarily sent internally to the application if the connection attempt fails, but it can also be sent by a server as reply to a CONNECT, if the server does not want to initiate connection.

PING Parameter: <time>, given in milliseconds. Used to measure latency of the connection, and also as a keep-alive if a connection stays silent for too long.

PONG Parameter: <time>, given in milliseconds. Usually a second PONG is returned, so that both nodes get updated latency information.

If any message in the connection phase is dropped or disappears, it is resent a number of times before giving up. This means that both nodes need to be aware of the possibility of duplicates arriving.

Chapter 5

Conclusion

The goal of this thesis was to introduce the concept of a game network library, and to explain the theories underlying the implementation created. The library created is not quite ready to be used in a fullscale game yet. It has not been ported to any platforms beyond the Windows PC platform on which it was designed, and quite a few of the more advanced features are not fully implemented. However, it is a good basis for further development.

Hopes were to have a working multiplayer implementation of a game at Coldwood. This has not quite been accomplished, although much of the work has been done. The network library in itself is functional, but there was not quite enough time to integrate it properly into the game simulation.

5.1 Benefits and drawbacks of the Gamenet architecture

In this section, a number of benefits and drawbacks of the chosen architecture and structure of Gamenet will be listed and examined.

Among the major benefits of the chosen structure are the following:

- It is layered, meaning that lower-level features are accessible separately and that the higher level constructs can be circumvented for smaller applications and use in simple tools.
- The message architecture is suitable for a variety of applications, not only games.
- Gamenet is designed with efficiency in mind, and the use of advanced C++ enables much of what in a different library might involve a lot of virtual calls and inefficient structuring to be inlined and very efficient.
- The conventions it imposes upon the application are kept as few as possible, without compromising efficiency.
- The coding style and naming convention of Gamenet is kept as close to the STL as possible, in order to avoid a jarring clash with any user code. Most programs of

today use the STL for many tasks, and so the Gamenet function calls and classes will fit in as well as the STL classes do.

- By using a functional library rather than writing the networking code from scratch, many headaches can be avoided and implementation time can be focused on higher levels of abstraction.

Drawbacks of using Gamenet include the following:

- Gamenet is written entirely in C++, and uses advanced C++ features like templates and member function templates. These are techniques only supported by the latest compilers, and for example to use the library on an older platform or with third-party tools for the PS2, some of these features may need to be worked around for that platform.

  Also, use of C++ makes the library impossible to use in a strict C environment, since it is designed to be compiled as a static library, not as a shared library (which could then have had an alternative C front end).

- Gamenet imposes structural limits on how the game works, and some classes need to be inherited from the library. This may create problems if the library is to be used with a game designed radically different. For example, the object manager requires that any objects registered to it inherit the `object` class, and currently also that the objects can be constructed using the default constructor, with no parameters given. If this cannot be fulfilled by the application, the library needs to be changed, or the object manager replaced.

- Currently, the Gamenet library is not fully tested in a production environment, and may have serious issues that prevent it from being as highly performing or memory-efficient as possible. No such issues have arised during the limited testing done while writing this thesis, however.

5.1.1 Performance

The design choices made when creating Gamenet were often made with performance in mind. For example, much focus was placed on speed rather than memory efficiency. These choices have worked out well, and despite them, the library does not take insane amounts of memory. It is hard to correctly judge the library's performance in comparison to other similar libraries, but it is a fair assumption to make that it is at least not worse off than any others.

Still, there are things that could be more efficient than they are, but these are only minor issues and not anything to worry about at this stage. For example, memory buffers for data that is to be transmitted are copied at least three times per transmission, but since the amount of data usually ranges between 50 - 100 bytes at the most and the copies happen maybe 20 times per second, getting rid of these copies is probably not even a noticable improvement.

5.1.2 Portability

The layered approach was taken in part to ensure ease of porting, since only the bottom layer would have to be modified when moving the library to a new platform. There are other things that may prevent a smooth porting, though, such as the use of advanced C++ template features. Some parts of the library makes use of new additions to the C++ standard, such as template member functions and partial specialisation. If only an older version of a compilator is avaliable for a certain platform, these features may become problems. Memory use will almost certainly be an issue when porting to a game console, since every byte needs to be accounted for when developing for a machine with very little memory. Gamenet makes liberal use of heap allocations, and the memory use may have to be rethought if porting to a console with very little memory.

5.2 Extendability

Adding further features to the library is made as easy as possible through a clear, layered structure and source code availability. Each layer in the library is designed to build on top of the previous in a way that makes it easy to use in a general environment and as unrestrictive as possible.

Things like account management and login server support can easily be added in parallell to the simulation structures, by circumventing the notification chain from the stream layer to the messenger, so that messages relating to the login server are managed by a different entity.

The areas most likely to be extended and modified are object state transmission and client area of interest algorithms. These parts of the library are already designed to be modified, with the object state transmission given as a bit iterator at the lowest level, and the area of interest algorithm as a simple callback to user code. To add new algorithms for compression such as prediction schemes or arithmetic packing is as simple as writing new functions that takes game variables as parameters and writes them in a compressed manner to the bit stream (and vice versa).

5.3 Future development

Currently, Gamenet is a working network platform for simple games written for the Windows PC platform. This might not be all that is needed to make a true production game. Some things may also need to be changed or modified to be stable enough for practical use. For example, it is in the current implementation only possible to send messages to nodes that are directly connected to our own node, but in a client-server game, most of the other nodes connected to the network would not be directly connected. A message relaying scheme would have to be added to the messenger, so that it can receive and retransmit messages destined for other peers.

Right now, there is no code for possible game arbiters. If making a commercial game, you might want a central game server to which players can connect to check high scores,

chat with other players and find others interested in starting a game. Such a game server needs to be written for Gamenet if it is to be used in a real game.

5.3.1 Security

A security protocol would need to be implemented and added to the library, to prevent cheating and stealing of CD keys or other similar problems that might occur.

A way to implement such security would be to add a simple encryption stage to the stream transfer, so that all data passing through the stream class is encrypted via some encryption scheme. If a public/private key scheme is employed, this should prevent most casual hackers from modifying the transferred data en route.

Preventing client-side cheats is more complicated. It would envolve client-side checksums of executables, but this is an infinitely complicated problem since the client-side is never trusted, and thus any work done there is susceptible to being hacked.

OpenSSL

The login process itself might involve the exchange of CD keys, passwords or other such sensitive information. One way to protect this data would be to handle the initial handshaking over a secure protocol, for example OpenSSL. However, I know too little about this kind of network security to say for certain, or to adapt the library for this kind of use.

5.3.2 Massively Multiplayer

The phrase *Massively Multiplayer* [Ent03] is used to refer to persistant online worlds with hundreds of thousands of users, and a network of servers connected to databases and login servers. There are a number of things missing from the library needed if it is to be used in a massively multiplayer environment [MZ03].

- A secure protocol is needed to transfer account information.
- Client-side checks of client executable integrity, to prevent modified clients from being used.
- Protocols for intra-server communication, account management, central user database, login server management are needed.

List of Figures

List of Examples

Bibliography

[Aro97] Jesse Aronson. Dead reckoning: Latency hiding for networked games. http://www.gamasutra.com/, September 1997. (January 2004).

[AW01] Fabio Policarpo Alan Watt. *3D Games - Real-time rendering and Software Technology*. Addison-Wesley Publishing Company, 1st edition, 2001.

[Bo03a] Jonathan Blow<jon@number-one.com>. Using an arithmetic coder: Part 1. *Game Developer Magazine*, August 2003.

[Bo03b] Jonathan Blow<jon@number-one.com>. Using an arithmetic coder: Part 2. *Game Developer Magazine*, September 2003.

[Com04] C++ Standards Committee. Boost libraries. http://www.boost.org/, January 2004. (January 2004).

[Ent03] Sony Online Entertainment. Everquest. http://www.everquest.com/, 2003. (January 2004).

[Fou03] The GNU Foundation. Gnu wget utility. http://www.gnu.org/software/wget/wget.html, October 2003. (January 2004).

[GSN94] W. Png G. Singh, L. Serra and H. Ng. Bricknet: A software toolkit for network-based virtual worlds. *Presence*, 3(1):19–34, 1994.

[Huf52] David A. Huffman. A method for the construction of minimum-redundancy codes. Technical report, Massachusetts Institute of Technology. Cambridge, Mass., 1952.

[iS97] id Software. Quake ii. http://www.idsoftware.com/, November 1997. (January 2004).

[KE94] P. Francis K. Egevang. The ip network address translator (nat). http://www.ietf.org/rfc/rfc1631.txt, 1994. RFC 1631.

[Keg03] Dan Kegel<dank@alumni.caltech.edu>. The c10k problem. http://www.kegel.com/c10k.html, November 2003. (February 2004).

[KGJJ03] J. Reichwald K. Grönlund<c99kgd@cs.umu.se>, A. Norberg, O. Johansson, and N. Johansson. Panzer - in the future, there can be only tanks. http://panzer.sourceforge.net/, 2003. (January 2004).

[Lan03] Sam Lantinga<slouken@libsdl.org>. Simple directmedia layer. http://www.libsdl.org/, 2003. (January 2004).

[Mic04] Microsoft. Directplay. http://www.microsoft.com/directx/, 2004. (January 2004).

[Min01] Nelson Minar. Distributed systems topologies: Part 1. http://www.openp2p.com/pub/a/p2p/2001/12/14/topologies_one.html, December 2001. (February 2004).

[MZ99] S. Singhal M. Zyda. *Networked Virtual Environments: Design and Implementation*. Addison-Wesley Publishing Company, 1st edition, 1999.

[MZ03] H. Trefftz M. Zyda, I. Marsic. Handling heterogeneity in networked virtual environments. *Presence: Teleoperators and Virtual Environments*, 12(1):37–51, 2003.

[oST94] National Institute of Standards and Technology. Digital signature standard. http://www.itl.nist.gov/fipspubs/fip186.htm, 1994. (February 2004).

[Pos80a] J. Postel. Transmission control protocol. http://www.ietf.org/rfc/rfc0761.txt, 1980. RFC 761.

[Pos80b] J. Postel. User datagram protocol. http://www.ietf.org/rfc/rfc0768.txt, 1980. RFC 768.

[Smi03] Russell Smith<russ@q12.org>. Open dynamics engine. http://opende.sourceforge.net/, January 2003. (January 2004).

[Sol92] K. Sollins. The tftp protocol, revision 2. http://www.ietf.org/rfc/rfc1350.txt, 1992. RFC 1350.

[SP97] S. Prashad S. Prasad, S. Prasad. *Multithreading Programming Techniques*. McGraw-Hill Companies, 1st edition, 1997.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley Publishing Company, 1st edition, 1994.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 3rd edition, 1997.

[vdMea03] J. van der Meer et al. Rtp payload format for transport of mpeg-4 elementary streams. http://www.ietf.org/rfc/rfc3640.txt, 2003. RFC 3620.

Appendix A

Sample implementations

The main purpose of writing the network library was to create a basis for Coldwood to further build their networking platform on top of. Thus, it was designed to be upwards extensible but also portable and scalable. None of these features are easy to demonstrate since the prototype only exists for MS Windows.

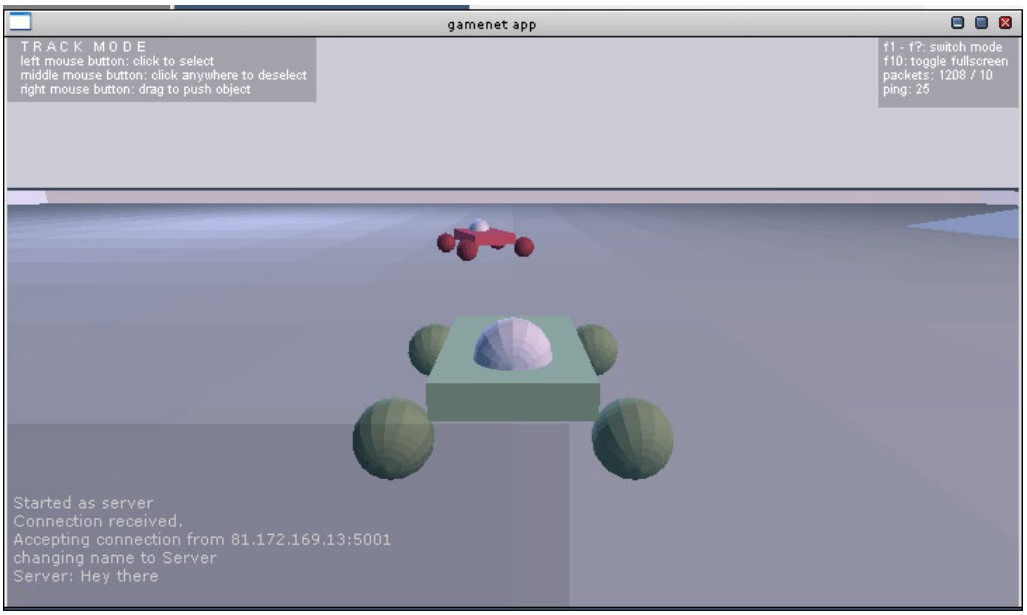

Figure A.1: A screenshot of the server version of the sample implementation. Statistics are automatically updated by the stream class, and can be read by the application.

Demonstration

A simple demo application was developed during implementation. It is a simple physics simulation using the ODE library [Smi03] for physics simulation and the SDL library [Lan03] for graphics and input.

The goal with this demonstration is to show how a simulation can be distributed across computers in a network. Both the server and the client each control a simple vehicle and can drive around the small level of the demo. There is one object besides the two cars, a box, that can be manipulated by both sides of the simulation.

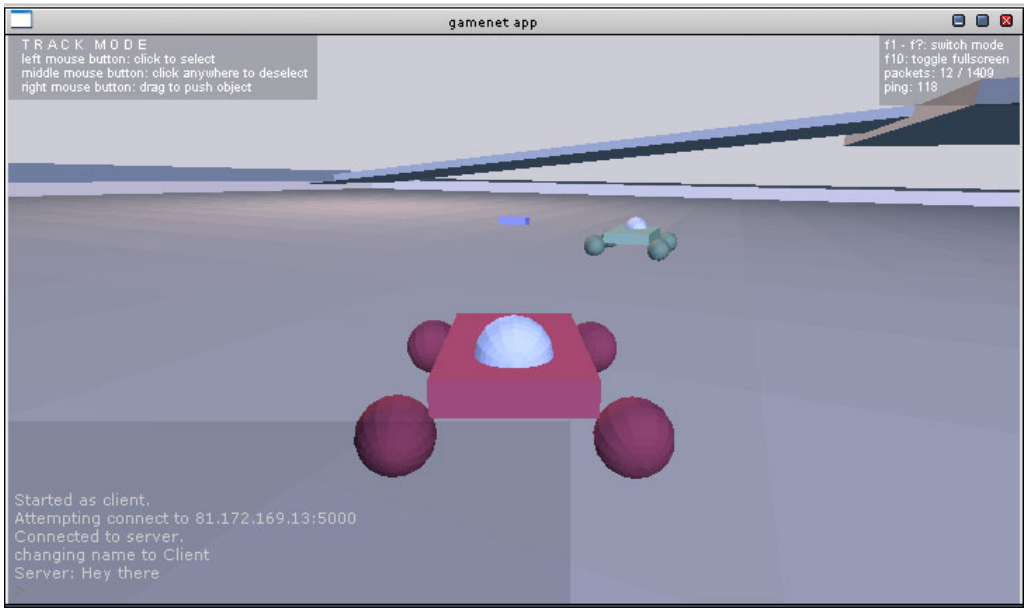

Figure A.2: A screenshot of the client application in the sample implementation. The client can control his own car. Currently, only the server actually moves any entities, so there is a constant latency in the response of the client. A proper network application would update the local simulation directly on the client, and smoothly correct any errors as data arrives from the server.

Coldwood game

One of the goals with this thesis was to have a working multiplayer game simulation. This goal has been reached in the demo application, but not quite in the actual Coldwood games that were originally intended. A lot of issues arise when inserting a new piece of code into an older codebase, and the integration process, getting things to work together, has taken a lot of time.

As of the moment of publishing this thesis, the Gamenet library is not fully integrated

into the game and it is not playable over the network. However, as the demo application shows, the goal of multiplayer interactivity is not far away.

Figure A.3: A screenshot from one of the Coldwood games that may one day use Gamenet for network communication. Work on integrating the network library into a racing game is, as of the writing of this thesis, under way.