University of Warwick

LibNRG

A Networked Multi-player Game Framework

Alex Baines

0923347

Computer Science

Supervisor - Abhir Bhalerao

2012 - 2013

Abstract

This report describes the creation and design of LibNRG, a networked multi-player game framework that aims to help programmers to implement internet play into small-scale real-time games. The components of the framework are discussed along with the reasoning behind decisions made throughout their development. In addition, the management of the project as a whole is commented upon, including a reflection on aspects that went well and those that did not.

Keywords

Multi-player, video-games, networking, library, framework, C++.

Table of Contents

Introduction

The process of creating a video game can be a daunting prospect for programmers due to the large number of required systems that must be implemented, including graphical rendition, user-input management and audio playback. When the game being created requires multi-player capabilities which work over the internet, then the bar is raised even further due to increased complexity.

For this reason, many software libraries and frameworks have been created that can aid programmers in implementing some of the aforementioned components, such as the Ogre3D rendering engine [1] or the glfw windowing and input management library [2]. Another option that is available to video game programmers is to use a full game-engine that provides all necessary systems in a single package as is the case with the popular Unreal engine [3], for example.

An important aspect of these frameworks, libraries and game-engines is the level of abstraction that they provide over the equivalent standard set of functions made available by the operating system. A low-level of abstraction remains closer “to the metal” and generally gives programmers a greater amount of power at the cost of requiring larger amounts of code to be written; whilst a high-level abstraction is designed around providing a regularly-used subset of functionality that is much easier to use.

LibNRG – Networking for Real-time Games – is a new library that is specifically aimed at helping programmers to implement networked multi-player functionality in their game while providing a high-level of abstraction that is also game-engine agnostic.

Motivation

Many libraries that focus on adding internet play to games only provide a very low level of abstraction over the standard POSIX socket functionality that is available on most operating systems [4]. Two popular games programming libraries fall into this category, LibSDL (specifically SDL_net) [5] only provides abstractions over sockets and packets that barely add anything over using the equivalent POSIX functions, while LibSFML only adds HTTP and FTP functionality on top of this [6].

It is only when moving on to game-engines that high-level networking constructs designed for games can be found; the Quake 3 engine (idTech 3) contains sophisticated networking functionality that has been well explained by Fabien Sanglard [7]. However this game-engine is also highly specialised towards Quake-type games and comes with rendering, input handling and other components that are tightly integrated together. If a programmer has already chosen a separate rendering engine that they wish to use, or wants to create a game completely different from Quake, then they will likely have to spend significant time modifying and removing code from the engine before even starting to program their game, which is not an ideal situation.

This is one reason that the creation of LibNRG was decided upon: to provide a library that is as easy to use and powerful as the networking code found in game-engines, but is not coupled with any other network-unrelated components. This gives programmers an easy to use system for networked multi-player functionality with the freedom to use whichever other components they want.

Another motive that influenced LibNRG's creation is the fact that many networking libraries come with proprietary licenses, require payment and/or royalties in order to use, or do not provide their source code. These traits limit the usefulness of such libraries, since proprietary licenses may include terms that prohibit modification whilst a closed source library inherently suffers from this same weakness. Raknet is one such popular networking library that comes with a hefty 10 page license agreement and, although being free for hobbyists, requires payment for games over a certain budget [8]. Another library with similar goals to LibNRG is Zoidcom. This library is free to use but its flexibility suffers due to the fact that it is not open source, and is only available as a pre-compiled shared library on x86 platforms [9].

Objectives

Many objectives were set for the project in the initial specification document, with the main focus on providing a library that can be used to implement networked play within small-scale games that are played in real-time between players, along with an example game to demonstrate its use. These foci were chosen to limit the scope of LibNRG to a project that could be achieved within the allocated time frame since computer networking within video games is a diverse area in which different models apply to different types of game.

Small-scale was chosen in particular to make it clear that LibNRG would not be suitable for a massively multi-player type of game. This is because the scale of such games is only achievable through large teams with a great deal of surrounding infrastructure, whereas LibNRG was intended to cater more towards hobbyist games programmers such as myself.

The choice of real-time games as an objective along with the specification's additional advice that UDP sockets should be used added some challenge and complexity to the project due to the unreliable and connectionless nature of UDP. Additionally, it added some purpose for the library to exist as way to expose the inherent advantages of using UDP in real-time scenarios to programmers without them having to worry about packets being lost.

Another major objective for LibNRG was to provide a high-level of abstraction from the underlying network technologies that it uses. This objective was designed to set LibNRG apart from other libraries that provide little benefit over using the POSIX socket functions such as the previously mentioned SDL_net.

The specification also stated that LibNRG should provide a Client / Server model and handle network-related tasks such as managing player and game state, serialising user data, and transmitting the data at regular intervals. These objectives were created to influence the functionality of library to be around core, necessary aspects of networked video games as opposed to utility functionality such as matchmaking. This was in part to differentiate LibNRG from, and allow interoperability with other services that exclusively provide these utility services such as the Steamworks service offered by Valve Corporation for games that are sold through the Steam digital distribution platform [10].

An objective to remain relatively bandwidth efficient is additionally proposed in the specification, a task that is important to consider especially in real-time games due to the clear need to relay information fast and frequently. As well as the objective of being low-latency which is compounded by the real-time focus and ties in with the suggestion of using UDP as is discussed in the Latency section of this report.

As a more implementation-related objective, the programming language suggested to be used by the specification is C++, with the library compiling on the GNU/Linux operating system. The choice of C++ as primary programming language in this case is due to the large number of video games that also use C++ as their primary implementation language, whilst the choice of GNU/Linux for operating system was to allow for the library and associated example game to run on the DCS Lab computers.

Another section of objectives in the specification say that LibNRG should be licensed under a free, permissive software license, and come with full source code available to its users along with documentation generated by a tool such as Doxygen [11]. The reasoning behind such conditions is to not limit programmers that wish to use LibNRG with any restrictions and to grant them the ability to use, modify and redistribute the library – conditions that are used to define free software by GNU.

The final suggested objective was implement support for creating replay files that can be loaded back by the library, allowing a particular moment of game play to be saved such that they can be viewed again later. This objective was intended to add a useful and interesting feature to LibNRG that is present is several other video games such as those using Valve's source engine [12].

Project Assessment

This section of the report details how the development of LibNRG faired when compared to its original objectives, the strengths and shortcomings of the library as well as a personal assessment of my own performance during the course of the project.

Completion of Objectives

Most of the objectives set for the project were completed successfully: LibNRG currently implements all the functionality necessary in order to create the type of video-game that it was tailored towards and more as is demonstrated by the example “ball duel” game that has been created alongside it.

The objective to use a Client / Server model has been realised, and LibNRG is split into both server and client-side functionality lead by the Client and Server C++ classes. This design is able to keep the overall library easy to conceptualise for its users, whilst remaining suitable for both small-scale and real-time games as is discussed in more depth soon in the report.

The specification's suggestion of using UDP sockets for data transmission has also been achieved as LibNRG creates its own abstraction over UDP to manage connections using the ConnectionIncoming and ConnectionOutgoing classes. In order to ensure that dropped packets do not interfere with the library's operation, LibNRG uses acknowledgements from the client to determine from which point in history data needs to be sent. This use of UDP fits well with the additional objectives of catering to real-time video games and providing low latency data transmission as shown in the latency results presented towards the report's end.

The goal of providing a high-level of abstraction has been a major focus throughout LibNRG's development and has led to the innovative Entity and Field class design. This design utilises features of C++ such as template classes in order to provide programmers with an efficient and concise model that works in a familiar object-oriented fashion. This abstraction is described in more detail in the design section of this report.

The objective to be bandwidth efficient has also been taken into account when creating LibNRG, which led to the Snapshot and DeltaSnapshot based design which accumulates all changes between intervals and sends the minimum set of changes that the client needs in order to recreate the server's game state.

The programming language used for LibNRG and the example game is C++ as the specification advises, making the library more relevant for the large number of games created in C++ whilst also providing the high level of performance that its compiled nature brings. Both the library and example game also follow the specification's advice to run on the GNU/Linux operating system and work correctly on the DCS Lab computers. The ability to compile on Microsoft Windows was stated as an optional objective if there was time to spare, but this objective was not completed. The library has, however, been careful to isolate platform specific functionality into a separate file: nrg_os.h such that porting it to other operating systems should not prove too challenging.

The objective to release LibNRG as free software under a permissive license is still planned, however the library is not currently available to the public – it is hosted as a private repository on GitHub.com [13] that can trivially be open sourced at any moment. The library will likely be released after the completion of this year of university but is currently being held back until a final grade has been given for this project. There are also some areas of LibNRG for which further work could be completed as detailed towards the end of this report that it may be beneficial to complete before the library is released.

The tentative objective of creating a replay system has also been completed successfully despite the acknowledgement in the project's progress report that there was a risk that it might not work. It turns out that it does work as one would expect – a replay that is saved to a file will play back the same series of actions that were viewed the first time. This correct behaviour is due to the simple design of having the file store received packets by the client and then having the client launch a ReplayServer object that sends them over again with the same timing.

The only major objective that was not met was thorough documentation of the library using a system such as Doxygen. The library is unfortunately lacking much documentation currently apart from the example game, due to the decision to hold off on creating documentation until after everything else had been completed and then subsequently running out of time to do so.

Assessment of chosen network model

LibNRG chooses to make use of the Client / Server network model for its communications. This model is fairly common in video games; it is used by games from various genres such as First-Person shooters in the Quake series and the sandbox game Minecraft [14], among many others. This model is advantageous in that it is easy to implement and has a single authority over the game's state that can help reduce the potential for cheating. A downside of this model is that the server requires enough bandwidth to send and receive data to and from all players, which scales at a $O(n^2)$ rate.

An alternative model that could have been used for LibNRG is a peer-to-peer architecture where clients communicate game state changes directly with one another. An advantage of this model is the ability to better spread out bandwidth requirements amongst all players instead of one central host while disadvantages include harder implementation and cheat-proofing [15].

A third type of model used by some video games is a synchronous architecture. This involves only player input being transferred amongst players, either directly or via a server, which each client then uses to construct the new game state. Many real-time strategy games use this architecture including Age of Empires [16] due to its major advantage that the amount of data needing to be transmitted is small even when there are very large number of entities or units. However this model has a major disadvantage in that every client must recreate exactly the same game state from the same input otherwise they will become desynchronised, and the game must therefore be deterministic based on user-input. This problem can be exacerbated by differences between types of computers such as rounding differences in floating point arithmetic and makes the architecture difficult to implement, as well as unsuitable for games with random elements [17]. Another problem with this architecture is that since every client must have a copy of the full game state, it is possible to create cheats such as “map hacks” [18] that reveal information that the client knows about but just doesn't display to the user, such as units in fog of war.

LibNRG chooses the Client / Server model due to its simplicity, suitability for a wide range of games, and better enforceability of cheat proofing. In addition, since LibNRG is focused on small-scale games, the scalability concerns of this architecture become less of a problem.

Assessment of legal, social, ethical and professional issues

One legal issue that was identified while creating the project's specification was that there are certain countries that prohibit the import or export of software that is capable of encrypting a user's data [19]. Such encryption would be a sensible thing to include within LibNRG since it is a networking library that transfers data over the internet, potentially via a route that could intercept this data. However since LibNRG is intended to be freely distributed online, legal issues could arise because of included encryption functionality.

Because of this issue it was decided that LibNRG should not provide encryption built-in, but instead provide a mechanism by which a user of the library can implement encryption if they so choose. This led to the creation of the PacketTransformation class that can apply any general transformation to data before it is sent over the network, and can be used for compression as well as encryption.

Authors assessment of the project

As suggested by the project website, I will now answer several proposed questions about my own views of LibNRG.

What is the technical contribution of this project?

LibNRG contributes an abstraction over the standard POSIX socket functionality that allows programmers to implement networked play into their games easily and efficiently. It also provides serious advantages over common alternative approaches:

Versus the alternative of using a simple TCP mechanism for real-time games, LibNRG is advantageous due to its use of UDP that does not exhibit stalls due to retransmission. It is also advantageous in that it requires less code to be written to get a multi-player game up and running than implementing a mechanism from the ground up.

Versus the alternative of using a full game-engine or proprietary library, LibNRG is more flexible due to its open licensing and specific focus on the area of networking, allowing other components to be used alongside it.

Why should this contribution be considered relevant / important to computer science?

This project is relevant to computer-science in that it demonstrates a practical approach to using the standard POSIX network API in the context of video games, and allows others to learn from it due to it being open source and permissively licensed.

LibNRG is also of some importance in the area of programming language design, since it shows how features of the C++ programming language such as template classes and inheritance can be leveraged to provide features normally only available in other languages. Specifically, how the Entity and Field abstraction that LibNRG uses can effectively allow for data members of a class to be iterated over in a manner similar to scripting languages such as Ruby [20].

How can others make use of the work in this project?

By its very nature LibNRG is a project designed to be used by others, be them programmers creating video games or players experiencing a game that was made using the library. Its focus on providing a high-level of abstraction also adds to its usefulness, because a programmer needn't have knowledge of the underlying low-level network functionality.

In addition, since the project has been created under a free software license, other people can use what is given by LibNRG to create their own network-related library and release it themselves or simply enquire into the inner workings of LibNRG due to it being open source (when the year is over).

Why should this project be considered an achievement?

LibNRG is an achievement in that it fills a gap that no other available networking middle-ware does, by giving programmers a free, high-level framework for small real-time games using efficient and innovative methods behind the scenes.

In addition, since a fully playable multi-player game has been developed together with the library, this in itself can also be considered an achievement in that it demonstrates that LibNRG can actually be used for its intended purpose.

What are the weaknesses of the project?

The main weakness of this project in my personal opinion is that, although it contains all features necessary to implement networking for games, there are still huge amounts of game-related network features that it does not contain, mostly due to the fact that the project has only had a relatively short period of time to be developed in. LibNRG would benefit greatly from continued development after the completion of this particular project in the form of more features as well as more fully-featured games being created that make use of it.

Another weakness comes in the form of lacklustre documentation due to the respective incomplete documentation objective, though this is also something that can be fixed through further development.

Project Management

Development Model

The software development model that was adopted for the creation of LibNRG fell naturally into a pre-planning style similar to that of the traditional waterfall development model due to the requirement to produce a specification document at the beginning of the project.

The specification was not a full plan or design document that detailed exactly how each part of LibNRG would be created however - something that would typically be produced in a Waterfall methodology. Because of this, the development model became a bottom-up design approach using the specification as the guide as to which components were required, with the more detailed component design work being undertaken before the implementation of each respective component. This work-flow could be considered a more iterative design approach, with each iteration being the creation or refinement of a particular component of LibNRG, albeit without the creation of any more specifications or design documents.

This process of component development continued throughout most of the project, with care being taken to determine how each component would fit together to form LibNRG as a whole. Initially, the components that were explicitly mentioned in the specification began to be created and from doing this, the need for additional components arose. An example of this is the NetAddress class, the need for which quickly became clear from the development the Socket class.

The testing and verification stage is mentioned in more detail later in this chapter, but for the most part it ran alongside the design/implementation stage with a larger focus during the development of the example game, which could be considered a form of test-suite due to its reliance on the majority of LibNRG's systems.
Maintenance was mainly combined with the design/implementation stages also, and came in the form of bug fixing and re-factoring certain components such as the Snapshot class, which was rewritten into a more concise form during the second half of development. Since the library has not yet been released to the general public, no real post-production maintenance has been undertaken, however it will no doubt be required due to the inevitability of users discovering bugs – assuming anyone actually decides to use LibNRG that is.

Time management

During the creation of the project's specification, a preliminary timetable was created based on predictions about which components would have to be created. This timetable is presented here:

	Week																							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Socket & Packet classes		■	■																					
UDP connection abstraction			■	■	■																			
Client & Server classes					■	■																		
Entity class & serialisation						■	■																	
Sending player input events							■	■																
Updating game state from input								■	■															
Tracking player state history								■	■	■														
Interpolation & lag compensation										■	■	■												
Client-side state updating							■	■	■															
Example game								■	■	■	■	■	■	■	■	■	■	■	■	■	■	■		
Customisation options													■	■	■									
Compression & Encryption															■	■	■	■						
Doxygen documentation										■	■	■	■	■	■	■	■	■	■	■	■	■		
Replay files																		■	■	■	■			
Optimisation & finishing touches																						■	■	
	Term 1										Xmas				Term 2									

When undertaking the development of the library, this timetable was used as a reference to work from to ensure that progress was relatively consistent with what had been predicted. Of course, some components that are now within LibNRG were not considered in the original specification – such as the aforementioned NetAddress class – and were not present in the original timetable. Aside from these cases however, the progression of design and implementation did remain within a reasonable range of the original timetable, and because of this there were no amendments made to the timetable when the progress report was being written.

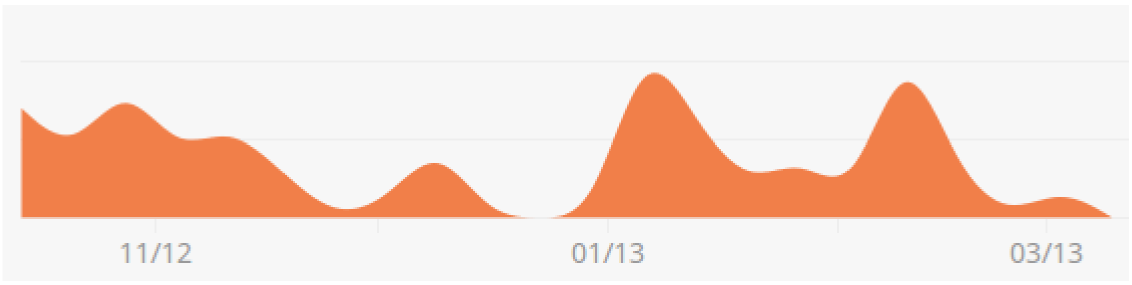
Moving on to the holiday period and the second term, the timetable was followed less strictly. The main focus of finishing off the library and creating the example game that was initially planned was still stuck to, with one notable exception of documenting the library using Doxygen.

This documentation work was originally planned to run alongside the finishing stages of the library, however due to the interfaces of various LibNRG components being fairly volatile during development, it was decided to move the documentation stage to be completed after the library was finished. This provided more time with which to complete the main focus of finishing LibNRG to a satisfactory standard along with the example demonstration game, and since the example game would act as a form of documentation about how to use the library in itself, this change was considered beneficial to the overall project.

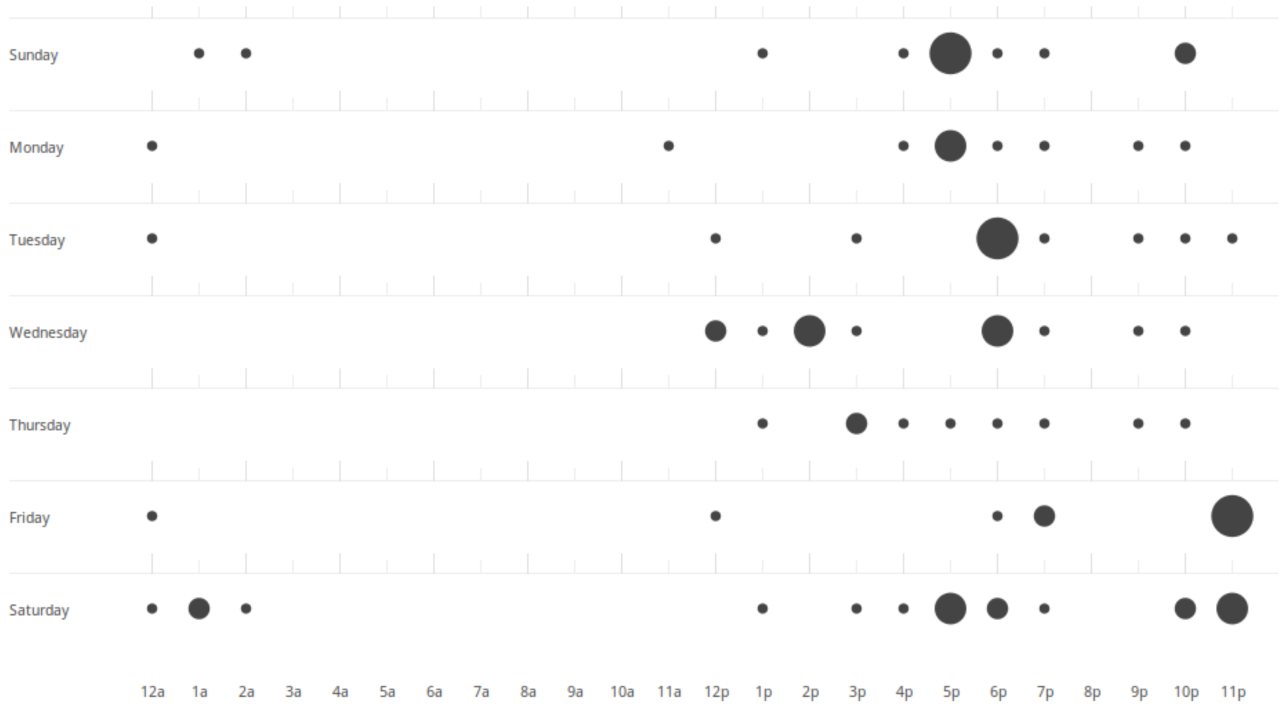
Towards the end of Term 2, LibNRG was functional and the example game completed to a reasonable level. At this point, non-critical features of LibNRG were developed such as the ability to create and watch replay files and retrieve statistics from the client-side of the library about packet loss and latency, including the ability to create a visual representation of this information in the ClientStats class. Unfortunately the addition of these features did not leave enough time to document the library's classes using Doxygen as was initially planned but this was not too detrimental due to the example game still acting as documentation.

Another deviation from the timetable is that the User-input functionality was reimplemented during the second term in order to make it more flexible for programmers by reusing elements of the Entity and Field based abstraction that LibNRG provides, which is explained in more detail later in this report.

Various statistics and graphs about time management can also be extracted from the library's git repository hosted on GitHub.com. For example the "Code Frequency" graph provided by GitHub shows that the commit activity of code is fairly spread throughout the two terms, with a lull during Christmas:



In addition, the "Punch card" figure presents the times of day that work was commited to GitHub, and shows a trend to commit code during the evenings:



Tools

Many tools were used throughout the different stages of the project's creation and verification. In the initial development phase, the text editor “gedit” was the primary tool used for writing the C++ code that was subsequently compiled using the GNU C++ compiler g++ [21]. This compilation was performed on a personal desktop computer using g++ version 4.6 as well as the lab computers in DCS using the earlier g++ version that they provide.

In order to ease the compilation process, the GNU Make [22] system was employed and a simple makefile created with rules to compile all the cpp source files into both a shared and static library by simply running the “make” command. The same system was later adapted and used to compile the example game and test programs. This system was simple to configure and considerably better than typing the entire compiler command line every time, while also improving efficiency due to its ability to compile only those source files that had changed.

In order to keep track of changes to the source files, the git version control system was used. This consisted of a local git repository as well as a remote repository stored privately on GitHub.com which doubled as an off-site backup. The use of git was beneficial when re-factoring parts of the code-base and also when implementing experimental features due to the ability to easily create branches in the code's time line. If the re-factoring work or feature turned out well, then the branch could easily be merged back with the main master branch, or just as easily discarded in the alternate case without having to manually edit or store different versions of the code in different directories.

Testing / Verification

In the early stages of development, the majority of testing and verification work involved firstly, attempting to compile the current state of the library and resolving any errors that prevented successful compilation.

As the intial components such as the NetAddress and Socket classes were completed to a satisfactory level, a small test case program was written to make sure they worked correctly when composed, which printed out the result of various related functions. The output could be compared to known correct output through manual inspection in order to verify the classes worked as they should. Another similar small test was created later to confirm the clone behaviour or the Entity class worked when used with the EntityHelper class described in design section.

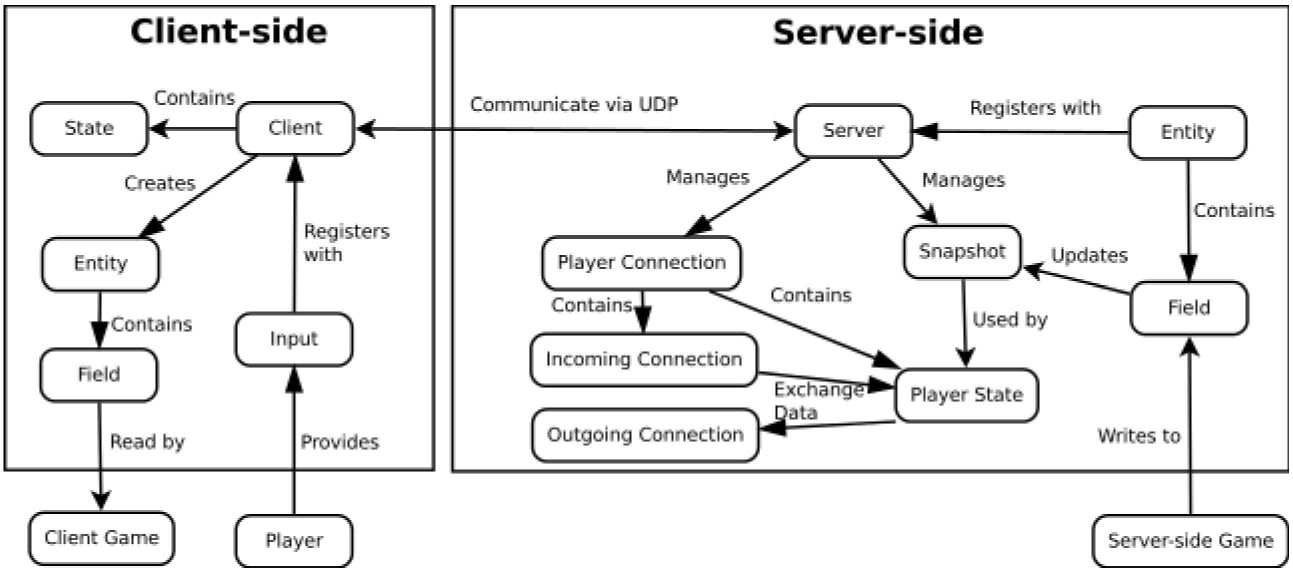
As the project progressed and the example game began to be worked on, the game became a replacement for further individual test cases, since it was designed to encompass the vast majority of functionality that was exposed by LibNRG.

In order to verify that the data being sent over the network by LibNRG was in the expected format laid out by the code, the Wireshark packet capture program [23] was used. Wireshark allowed all packets to be listed in order as they were sent over the loopback address of the development system, and provided raw hexadecimal dumps of their contents for inspection.

The GNU debugger gdb [24] was also used to set breakpoints in the library and analyse its behaviour when it was not functioning as expected. Additionally, the memory checking utility valgrind [25] was used in order to detect cases when memory was being accessed incorrectly or being leaked. The current version of LibNRG has been confirmed to correctly deallocate all memory associated with its Server and Client classes when their destructor is called by using valgrind.

Design

LibNRG has been designed as many separate components that form together into both a client-side and a server-side. A diagram indicating how the main components are related and how the user of the library interacts with the system is presented here:



This diagram shows that the basic premise of LibNRG's functionality. On the client side this is to obtain user-input, send it to the server, and parse any data received to create Entities and/or update the values contained within an Entity's set of Field objects. The server-side periodically creates Snapshots from Fields that have changed since the last update, calculates which changes need to be sent to which clients and then sends them. The server-side also deals with accepting connections from new clients and relaying updates in user-input to the programmer so that they are able to update their game's state based on it.

The client-side is also designed to run one Snapshot behind the most recent one sent out by the server. This allows for interpolation to occur between the two most recent snapshots on the client-side providing a much smoother game play experience, as is demonstrated by the interptest program included with the library.

Component Design

The design and implementation of the individual components of LibNRG is now described in greater detail, roughly following the chronological order in which they were created.

Socket

The Socket class was the first component that was designed and created during the project and is intended to wrap the standard POSIX socket functions together with a friendlier interface for programmers.

When an instance of the Socket class is created, it acquires a file descriptor using the C socket function, which is later released in the object's destructor. This follows the common RAII C++ idiom that ensures that the resources associated with the object are managed correctly [26].

The class makes several methods available that act upon the socket file descriptor in order to bind the socket to a port, connect it to a remote endpoint, send data, receive data and get the addresses to which the socket is bound or connected. The methods that receive and send data have two important differences from the POSIX send and recv functions:

Firstly, instead of a void pointer for the data argument, the methods take an nrg::Packet reference. This change ensures that the programmer does not supply an incorrect length for the data they are sending since this length is stored and managed by the Packet object, removing the possibility of a buffer overflow. Additionally since a reference is used as opposed to a pointer, it is not possible to accidentally supply NULL to the methods, removing the opportunity for a null-pointer dereference. Both of these situations can lead to security issues such as remote-code execution (in the case of buffer-overflows) [27] or denial of service (in the case of a remote null-pointer dereference) and thus the fact that LibNRG eliminates them helps programmers to have a more secure code-base.

Secondly, instead of taking a struct sockaddr* for the address argument, the nrg::Socket methods use an nrg::NetAddress reference. This avoids the ugly casting that is usually required from the various POSIX sockaddr_in types as well as providing other advantages that are discussed in the nrg::NetAddress section.

The Socket class also provides the ability to make a socket non-blocking, along with the dataPending method to check if there is data ready to read. This is important in the context of video games that need to draw a video frame at regular intervals and cannot block waiting for network activity that is not guaranteed to arrived at a particular time.

The socket class is also capable of recording timestamps for when the last packet was received. This feature is able to make use of the Linux-specific SO_TIMESTAMP socket option which, when used in conjunction with the readmsg function, can return the exact time that the packet was received by the kernel [28]. This provides much better accuracy than the alternative approach of simply getting the system time when the Socket's read method is called, since the packet could have been received some time before the read method is called. The accuracy of this measurement is important since it directly influences the effectiveness and smoothness of the client's interpolation feature.

Users of LibNRG can create instances of this class if they wish to send raw data between networked computers, but the Socket class is also used internally by the Client and Server classes that exert a greater level of control over the data that is transferred over the network and are part of the library's main high-level abstraction.

NetAddress

The NetAddress class was not originally considered in the specification, but its requirement was quickly realised when creating the Socket class. It is responsible for holding onto an IP address in conjunction with a port number, and represents an unique internet endpoint.

Internally, this class uses the POSIX getaddrinfo function in order to resolve a given string argument into an IP address whilst also providing methods that can retrieve the associated IP address as a string, the address family, and the port. Another useful method is toSockAddr which returns a const struct sockaddr* that is equivalent to the NetAddress object on which it was called; this structure can be given to the standard POSIX send / recv functions without the need for any type casting.

This class has been designed to be IPv6 aware, and can hold either IPv4 or IPv6 addresses inside by making use of the POSIX sockaddr_storage structure, which was designed to have enough space to hold either type of address [29]. When determining a string representation of the address, the NetAddress class also uses the inet_ntop function which can handle either type of address as opposed to the older BSD inet_ntoa function that was only designed around IPv4 addresses [30].

As an additional bonus, the NetAddress class provides the standard operator<, operator== and operator!= methods that allow a total ordering to be applied between multiple NetAddress objects. This is especially useful when using NetAddress objects inside a STL container such as std::map – which is exactly what the nrg::Server class does – since these containers use the operator< method in order to store the objects correctly and require another template argument if this method is not implemented [31].

Packet

The Packet class acts as storage for data that is about to be sent or has just been received over the network. It contains several methods that can be used to write data of a certain size into the packet or equivalently read data out, as well as retrieving the amount of data that is stored inside. This class also takes care of converting the byte-order of data when it is read or written so that the data sent over the network is always in network-order [32] / big-endian, while the data extracted is always in the byte-order of the host machine. This is achieved by using template methods similar to the ntoh{s,l} / hton{s,l} functions that are provided in the C standard library except that LibNRG's versions have been designed to work for any size of data in a single function and are defined in nrg_endian.h.

This class has been designed to work in a similar manner to the standard C file IO methods whereby a pointer is maintained and advanced whenever data is written to, or read from the file (or in this case nrg::Packet). Additionally, all the methods that read or write data return a reference to the same packet so that it is possible to chain together multiple read or write operations into a single statement, e.g.:

```
nrg::Packet packet;
packet.write16(val).write8(val2).write8(val3);
```

This works correctly because the pointer is incremented after each write operation, and therefore the data is written sequentially into the packet. One disadvantage of this approach however, is that a method that only reads from a packet cannot be passed a const reference, since reading data modifies the packet's internal pointer.

As well as the methods that explicitly specify the size of the data being written such as write8 and write16 for 8 and 16 bit values respectively, a generic write method has been implemented utilizing C++ templates. This method is able to use the sizeof statement in conjunction with the C memcpy function to write the correct number of bits for any type automatically. An equivalent set of read8, read16 e.t.c. methods along with a generic read template method have been implemented for the case of extracting data instead of writing it. One downside of using these template methods is that they require the programmer to be careful that the type they are writing or reading is the same size between different architectures, for example the size_t type may be 8 bytes on a 64bit operating system, but only 4 bytes on a 32bit operating system [33]. It is therefore recommended that the types from stdint.h that are guaranteed to be a certain size across all architectures [34] are used when using the template version of the read / write methods.

One major feature of the Packet class is that it will automatically expand its underlying data store when data is being written into it that no longer fits. Because of this, the Packet class is able to prevent buffer-overflow vulnerabilities that are one of the most common vulnerabilities in network-related software [27].

Connection

There are two classes related to managing a connection within LibNRG, ConnectionIncoming and ConnectionOutgoing. Together they are responsible for splitting large packets into several smaller ones, reassembling them, and adding and removing header information to and from all packets that are passed through them. This header information describes in which order the packets were sent, along with a few flags that indicate if a packet has been split or is the final packet that is expected to be sent.

Since UDP is based on the underlying IP protocol stack, it is capable of being automatically fragmented and reassembled by routers – in fact this is a requirement for internet hosts as laid out in section 3.2.1.4 of RFC1122 [35] – so the fact that the connection classes duplicate this functionality may seem unnecessary. However the buffer that hosts have in which to reassemble packets is only required to be a minimum of 576 bytes by the same RFC, which means that any packet sent over the internet larger than this is not guaranteed to be routed. Over Ethernet, the maximum transmission unit (MTU) is required to be 1500 bytes [36] which is more reasonable, and LibNRG therefore splits packets at 1400 bytes by default which leaves enough space for the IP headers.

The reason that the connection classes add sequencing information is due to the fact that UDP sockets have no guarantees that packets will be received in the order that they are sent, or that they will ever arrive at all. The sequencing information can be used to parse received packets in the correct order as well as reconstruct split packets correctly, and is similar to the sequence number that is part of the reliable TCP protocol [37].

PacketTransformation

The purpose of the PacketTransformation class is to allow for arbitrary post-processing effects to be performed on packets before they are sent over the network, such as data compression or encryption. The class itself is abstract and requires two methods to be implemented: apply to perform the transformation and remove to return a transformed packet back to its original state.

Once an derived class instance has been created, it can be given as an argument to the setTransform method of the ConnectionIncoming / ConnectionOutgoing classes, and from that point on, all packets that are passed through them will have the transformation applied / removed automatically. A NULL pointer can also be supplied in order to remove the transformation.

LibNRG comes with an optional derived class of PacketTransformation called PacketCompressor which uses the zlib C library [38] in order to compress packets before they are sent, and decompress them again once they are received. This can be used to increase the bandwidth efficiency of games that send large amounts of data with similar structures that are easily compressible.

State

The State class within LibNRG is a fully abstract class that represents the state that any one single player connection is in, and is used internally by both the Client class and each Player object that is created in the Server class.

Its purpose is to allow a connection to progress through different stages that each send and receive different formats of packets; for example when a player first makes a connection to a server they will be in a handshaking state that may or may not then progress into a game-playing state. These two cases are laid out on the client-side by the ClientHandshakeState and ClientGameState classes and equivalently on the server side by the ServerHandshakeState and ServerGameState classes.

All these classes derive from the State class and implement several virtual methods that are described here:

- bool addIncomingPacket(Packet& p)
  - This method adds a packet to the State for processing which returns true if the packet was well formed.
- bool needsUpdate(void)
  - Is called to determine whether the update method should be called.
- StateUpdateResult update(ConnectionOutgoing& out)
  - Is called to allow the state to update itself and optionally send any packets it wants via the *out* parameter, and returns an enumeration describing if the State is expecting more packets, is finished or has encountered an error.

By making use of these three virtual methods, many different protocols can be implemented and used with the Client and Server classes instead of using their default behaviour without much code change necessary.

Client

The Client class is the main component for the client-side of LibNRG and wraps the previously detailed classes into a single package to be used in conjunction with a host that is running the nrg::Server class.

This class is designed to be very user friendly, and as such contains only a few important user-facing methods: Its constructor or the connect method which specifies the host to connect to, an update method that sends user input to the server and processes any packets that have been received, a pollEvent method that is used to receive Events which are described later and the registerEntity method that is used to let the client know about the types of Entities that it will receive from the server.

There are also optional features exposed via the Client class such as the ability to obtain statistics about whether or not received packets have been dropped via the ClientStats class, and the ability to start and stop recording a replay file that can play back the received packets to recreate the game as detailed in the Replay section.

The main job that the Client class does is to maintain a stack of State objects, handing over packets it receives to the topmost one and popping them off the stack if they signal that they've finished via their own update method, with the majority of the legwork actually taking place inside the individual states.

Server

The Server class is the counterpart to the Client class and performs similar duties – with the exception that it maintains multiple clients simultaneously that all have their own individual state. It was designed to use the std::map associative container with an nrg::NetAddress as the key for each connection and a Player object as the value.

In contrast to the Client class – which has an update method that returns immediately – the Server class maintains a set interval at which it will send out packets and it will block by calling the select function [39] on its socket (via the Socket's dataPending method) until this interval has been reached. When the select call returns early to indicate that data is available for reading, the Server immediately passes the received packet to the Player in its map with the corresponding source address key (if it exists) for processing. This technique helps to reduce the latency of the system which is covered in more detail in the Latency section of the report.

In addition to players, the Server maintains a master Snapshot which contains the current data for all Entities that have been registered with it along with an array of DeltaSnapshots that represent all the changes that have occurred for the last 32 (by default) update intervals. This data is used by the ServerGameState class to send outstanding changes to each client connected to the server.

Player

The Player class itself is a fully abstract class that represents the functionality that users of LibNRG can perform to individual player connections, such as getting their latency – something that might be useful to display in a scoreboard – or kicking them out of the game with a rude message. Player objects can be accessed by the programmer whenever the nrg::Input::onUpdate method is called server-side, so that they can make changes to their game depending on which player made which input action. They can also be obtained via the Server's getPlayerByID method.

Internally, the Server uses a subclass of Player called PlayerImpl that contains additional functionality that programmers using LibNRG don't need to worry about, such as managing a stack of states in a similar fashion to the Client class. This design choice of splitting the implementation details from the interface adds to the overall encapsulation of LibNRG and is advocated by many C++ programmers such as John Lakos [40]. It is possible in this case since Player objects are never constructed explicitly by the user of the library, they are only passed to users by LibNRG itself.

Field

The Field class is one of several classes in LibNRG that make up its high-level abstraction of data-transmission tailored towards video games. It is designed as a C++ template class that takes any type as its first template argument along with an optional functor capable of performing data serialisation / deserialisation as its second template argument.

The first argument represents the type that is to be transmitted from server to client whenever it changes – for example the creation of an nrg::Field<uint32_t> would allow the programmer to transfer an unsigned integer. By default the type is serialised and deserialised using the small nrg::Codec class that just calls the generic nrg::Packet::write and nrg::Packet::read methods – although the Codec class is specialised for certain types such as std::string that require more complicated serialisation methods. In some cases a programmer may want to choose their own serialisation format, or use a tried and tested alternative such as JSON [41]; the Field class has been designed with this flexibility in mind which is why any object implementing an encode and decode method can be plugged into its second template parameter to be used instead of nrg::Codec.

Nrg::Field is actually also a derived class of the nrg::FieldBase class. This is necessary because the data serialisation / deserialisation system needs a single class type to iterate through and call the readFromPacket and writeToPacket virtual methods on, and since nrg::Field is a template class, an nrg::Field<int> is considered completely different to an nrg::Field<float> unless they share a common base class.

The Field class overrides its operator= method so that it is able to check if the new value is different to the one it is currently storing. If there is a difference then it will store the new value and call the markUpdated method of its container, which is responsible for acting upon the change in value. For programmers that don't approve of such operator overloading, then there is an equivalent set method.

There is also a specialisation of the Field template class for array types that allows for sending of <index, value> pairs when a particular array index is updated, as opposed to the full array that would be sent in the default implementation. This specialisation also uses a template meta-programming technique in order to use the smallest possible integer type when encoding the index;

for example an array of size 256 or less only requires a single byte to store an index, whereas 257-65536 would require two bytes for each index. The meta-programming technique uses the two classes min_sizeof and size2type as defined in nrg_util.h in order to determine this minimum size at compile time.

A further notable design decision for the Field class is that it contains two values inside itself instead of just one, called data and data_next. This is used for the interpolation feature of LibNRG that is available on the client-side of the library: data holds the value of this field according to the previous snapshot whereas data_next contains data from the most recent snapshot. When the field's getInterp method is called, it returns an interpolated value that is based on data, data_next and a number between 0 and 1 that is obtaned by calling client::getSnapshotTiming. This number represents how far through the server's interval the client current is, with 0 being at the start and 1 at the end. By default, linear interpolation is used using the nrg::lerp functor, however it is possible to change this behaviour on a case by case basis by supplying the getInterp method with an alternative. When a snapshot has not been received in time, the client::getSnapshotTiming method actually returns a number greater than 1, which causes extrapolation to occur instead. This extrapolation may well miscalculate the next state leading to a jerk in motion when it gets the next snapshot, but it is better than the alternative of completely halting the game until the next snapshot is received in most cases.

FieldContainer

The FieldContainer class is responsible for holding onto several nrg::Fields – nrg::FieldBases to be exact. The purpose of requiring such a class is to have a mapping that mimics standard C++ structs / classes with fields – except that in the case of LibNRG, the container provides the ability to iterate through the fields that it contains, something that is not possible with standard C++ classes.

This iteration ability is implemented as follows:

- Upon construction, an nrg::Field will receive an nrg::FieldContainer pointer and call its addField method.
- The addField method builds a linked list by looking at the container's field_head variable and calling its getNext() method until it returns NULL, at which point it calls setNext().
- After all fields have been constructed, the container's getFirstField() method can be called to get the head of the linked list, which can be iterated through calls to getNext() as before.

This field iteration feature is used internally by the Client and Server classes and their States so that they are able to automatically serialise and de-serialise the data for any fields that have changed without any work needed by the programmer.

The FieldContainer class isn't designed to be created by programmers itself, instead there are two derived classes within LibNRG that should be used instead, which are focused towards a specific aspect of games: the nrg::Entity class for game objects and the nrg::Input class to represent user input.

Entity

The Entity class within LibNRG is designed to represent any object in a game that may experience some change in state that needs to be relayed from server to client. It is a derived class of the FieldContainer class as previously mentioned, and is therefore also designed to contain instances of nrg::Field.

It adds the virtual clone and getType methods on top of the FieldContainer class, and operates by the user of LibNRG creating their own class that derives from it and including nrg::Fields for whatever data they want to have transmitted over the network. From here, LibNRG takes care of the dirty work of determining which data needs to be sent, serialising it on the server-side, transmitting it over the network, de-serialising it on the client-side and finally notifying the client application that new data is available.

In order for this automatic series of steps to take place, the programmer needs to register every entity they create on the server with the actual Server class via its registerEntity method. By doing this, the entity obtains a pointer to the Server object and can then call Server::markEntityUpdated whenever its own markUpdated method is called by one of its fields being modified.

On the client-side, the programmer just needs to register a single entity of each type with the Client object. After this, the Client is automatically able to create Entities from the data is receives by making use of the nrg::Entity::clone method on an entity of the same type as the one received. The clone method invokes the copy constructor of the derived class in order to create a duplicate object, and since the nrg::Field classes require a container pointer in their constructor you would be forgiven for thinking that a programmer using LibNRG would be forced to implement a non-default copy constructor for all their entities because of this.

However the Field and FieldContainer classes are able to make use of a clever design that works similarly to the offsetof macro that allows the default compiler-implemented copy constructor to be used, saving users of LibNRG some work. This macro is used by large scale software projects such as the Linux kernel, where it is part of a clever implementation of linked lists [42]. This lends some credibility to this technique, despite it appearing somewhat hacky at first. In the case of nrg::Fields, when their copy constructor is called, they determine what their container's address is by looking at the offset between the copy and the copy's container. This way they do not need to be explicitly given their container's address in their constructor when entities are cloned.

EntityHelper

The EntityHelper class is a C++ template class that can be used to more easily derive from nrg::Entity. It uses the Curiously-Recurring Template Pattern (CRTP) as detailed by Abrahams and Gurtovoy [43] in order to automatically implement the polymorphic copy construction that is required by nrg::Entity's virtual clone method.

It additionally takes a uint16_t as its second template parameter that represents the Entity's type; this is used to automatically implement the required getType virtual method of nrg::Entity for the programmer.

As an example of the difference between deriving from nrg::Entity and nrg::EntityHelper, consider the following two equivalent declarations of a user created entity:

Using Plain Entity:

```
class MyEntity : public nrg::Entity {
public:
      MyEntity() : example_field(this){}
      virtual Entity* clone() {
             return new MyEntity(this);
      }
      virtual uint16_t getType() const {
            return MY_ENTITY_TYPE;
      }
private:
      nrg::Field<uint32_t> example_field;
};
```

Using EntityHelper:

```
class MyEntity : public nrg::EntityHelper<MyEntity, MY_ENTITY_TYPE> {
public:
	MyEntity() : example_field(this){}
private:
	nrg::Field<uint32_t> example_field;
};
```

This clearly demonstrates the advantage that this class aims to provide: less code needs to be written when making use of it. Deriving from the Entity class is still useful in cases where the clone method needs to perform other work or not use the default copy-constructor of the entity however.

Snapshot

The Snapshot class is designed to hold a set of Entities and their Fields, and to provide the functionality necessary to transform them into a raw chunk of bytes that can be sent over the network in one or more packets.

The Server object stores a single Snapshot called master_snapshot that contains all entities and fields that are known to the system. This is used to create packets to send to newly connected clients that require knowledge of the full game state. Most of the time however, clients are not sent the master snapshot, but are instead sent only the changes between the current snapshot and the previous one. In order to do this, a second class is used: DeltaSnapshot.

Whenever an entity is added to a DeltaSnapshot, the snapshot first adds a new EntityData entry to its map using the entity's ID as a key. This EntityData object is a simple struct that holds some useful metadata about an Entity, such as the current values of it's fields after they've been serialised.

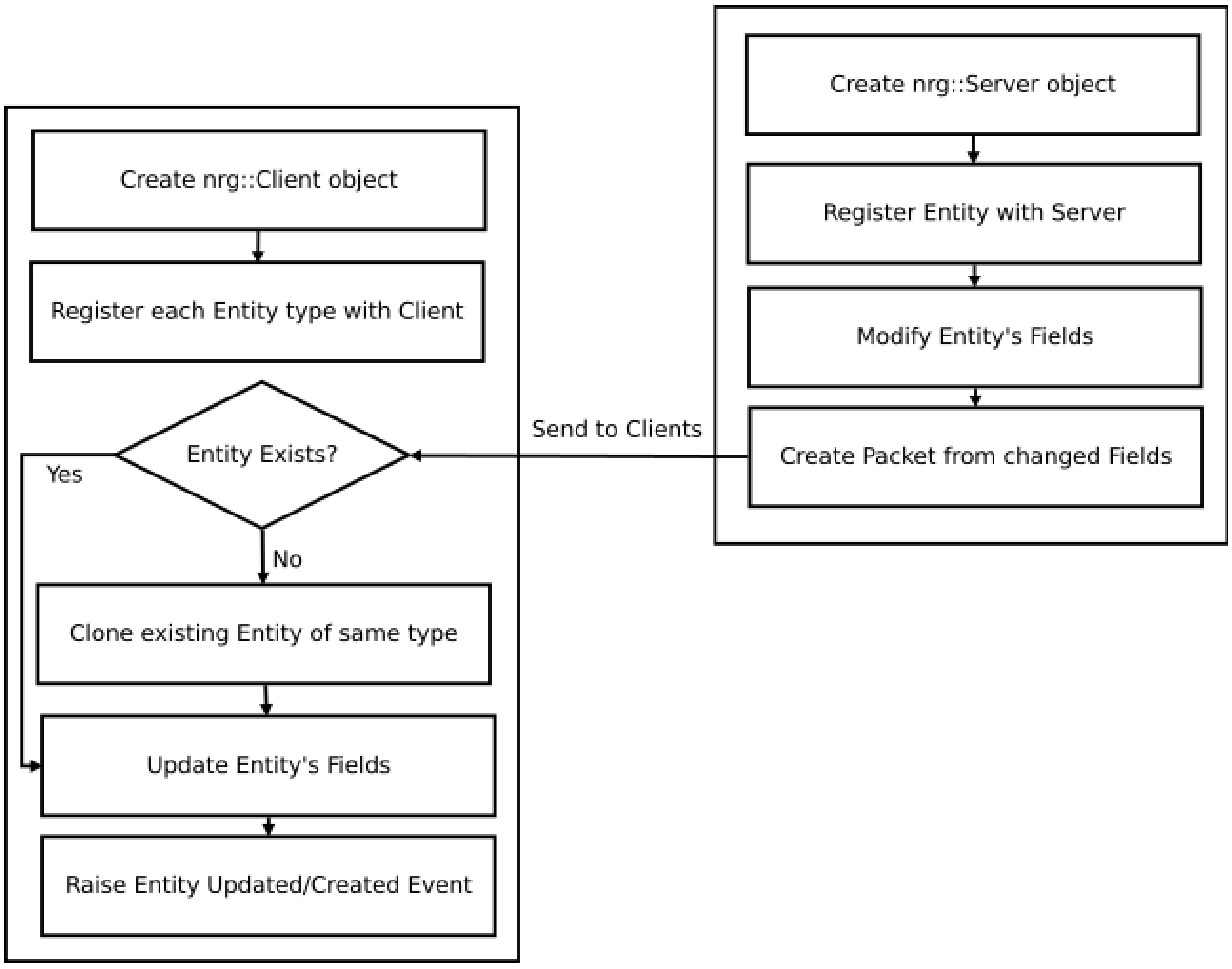
The snapshot then iterates through the entity's fields and checks which ones have been updated using the wasUpdated virtual method that all fields expose. If this field was updated then it is written to an nrg::Packet contained within the EntityData struct using the field's writeToPacket virtual method. This procedure is used so that the snapshot does not need to keep a pointer to the added entity or any of its fields, since the entity may have a shorter lifespan than the snapshot and accessing the pointer in this case would lead to a use-after-free situation.

When the DeltaSnapshot's own writeToPacket method is called it iterates through all the EntityData entries in its map, writes the ID and type of the entity to the supplied packet argument and then begins writing the field data. In order to write this data, the snapshot first looks at which fields were updated and writes a series of bits to the packet that represent this with a 1 indicating that it *was* updated and a 0 indicating otherwise. After this, the data from the packet in the EntityData struct is copied to the new packet just after the bitmask and this process is repeated for any remaining entities in the map.

On the client-side, the ClientSnapshot class takes care of de-serialising the data. When it receives a DeltaSnapshot packet it reads data for each entity in the order that it was written. Firstly the entity's

ID is read and used in order to lookup whether the client already knows about this entity. If it does then it skips to the next step, however when it doesn't it reads the entity's type and uses it as the key in a find operation on the client's entity_types map. This map stores an Entity pointer for each type of entity and is (supposed to be) filled by the programmer by calling the client's registerEntity method during initialisation.

Assuming the type was registered, then the clone method is called on the returned entity in order to create a new entity instance with its ID set to the one that was read from the snapshot. After this the ClientSnapshot queries the number of fields within this type of entity and reads one bit for each into a buffer. The buffer is then checked in a loop and if a 1 is found the readFromPacket virtual method is called on the corresponding field, thereby retrieving all the data that was serialised on the server-side by the snapshot. This process is more concisely described by the following flow chart:


Create nrg::Client object
Register each Entity type with Client
Entity Exists?
Yes
No
Clone existing Entity of same type
Update Entity's Fields
Raise Entity Updated/Created Event
Send to Clients
Create nrg::Server object
Register Entity with Server
Modify Entity's Fields
Create Packet from changed Fields

The ClientSnapshot also has the capability to delete entities when they are deleted on the server-side by the programmer. When receiving a normal Snapshot, the client-side can just delete every entity that is not mentioned within it, since a normal Snapshot contains the full game-state. However in the case of a DeltaSnapshot the client cannot use this logic since only modified entities are mentioned – those not mentioned still exist on the server but have just not changed during this interval. The deletion for a DeltaSnapshot is achieved by encoding the deleted entity to a packet as usual but setting the bitmask that describes which fields have been updated to all zeroes. Because a DeltaSnapshot would never usually include data for an entity that has no updated fields, the ClientSnapshot can interpret a fully zero bitmask as a request to delete the entity with the given ID.

In order to cope with the occurance of DeltaSnapshot packets being lost due to UDP's unreliability, the client will record the ID of the last Snapshot / DeltaSnapshot it received and send it back to the server when it sends user-input. This ID is recorded by the server and when it is sending an update to this client, all DeltaSnapshots with an ID between the recorded one and the master_snapshot's are combined using the mergeWithNext snapshot method in order to create a packet containing all the outstanding data that the client needs.

Event

Instances of the Event class are the main mechanism by which LibNRG is able to communicate to its users when certain things happen; both the Client and Server classes have been designed to relay events through their pollEvent methods.

This pollEvent method places an nrg::Event into the reference argument it receives if there is a new one available in the event queue, and returns true or false depending on whether or not an Event was available. The method is modelled after the similar PollEvent methods that are available in the popular games-programming libraries LibSDL [44] and LibSFML [6] and the method's signature is specifically designed to be used in conjunction with a while loop as is demonstrated in the following example:

```
nrg::Event e;
while(client.pollEvent(e)){
        switch(e.type){
                [process events here]
        }
}
```

There are several derivatives of nrg::Event for different types of occurance, such as EntityEvent which is fired whenever entities are updated or created on the client-side and DisconnectEvent for when the client is disconnected or kicked. However these events do not use the typical C++ inheritance style and instead opt to use a more C style approach that involves the plain Event type being a union object that contains one of each specific event type.

The reason for using a union instead of an inheritance style is that if an inheritance style were used, then every Event object would need to be dynamically allocated with an Event pointer held in the event queue. This is because C++ derived types are not homogenous without casting unless accessed via a reference or pointer, and references cannot be held in STL containers due to the requirement for copy construction. Additionally, the base Event class would need to provide virtual methods to static_cast itself depending on its type.

Using the union style means that the queue can simply hold Events and not Event pointers that can be allocated more efficiently inside the queue itself. Also no casting between types or virtual method calls are necessary. Not only does the overall efficiency of LibNRG benefit from this choice, but programmers have a much nicer interface without any casting required.

It is worth noting that LibSFML is also programmed in C++ yet chooses to use the C union style over the C++ style for its own Event system [45], presumably due to the same reasons outlined above.

Input

nrg::Input is a class designed to represent user input events that will be sent from the client to the server. Like nrg::Entity, this class derives from nrg::FieldContainer and works with the same paradigm of programmers storing values that they wish to be transferred in nrg::Field<*type*> objects inside the Input object.

The class is handled in a more straightforward way than Entities however: the client simply chooses a naïve serialisation approach whereby all Fields inside the Input object are sent to the server each time it is updated without performing any delta-compression from the last update.

The reasoning behind this approach is twofold. Firstly, user input should not be a large amount of data if the programmer is "doing it right" – ideally only state such as mouse x/y coordinates and keyboard / controller buttons should be included. Secondly, user input is likely to change between every game frame in a real-time game. Both of these factors mean that making the client track, and send only changes in input would add complexity to the system without a significant overall improvement in bandwidth utilisation, in fact, it may increase bandwidth utilisation if every field is updated during every frame, something that is explored in more detail in the Bandwidth measurement section of this report.

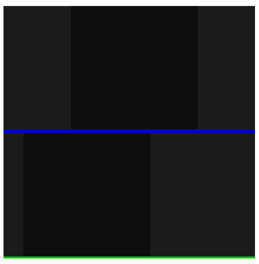
Another difference that Input has from the Entity class is that LibNRG only works with a single instance of the Input class on both the client and server-side. This removes the need for the clone and getType methods that are required by nrg::Entity to create new instances of itself dynamically, simplifying things even further.

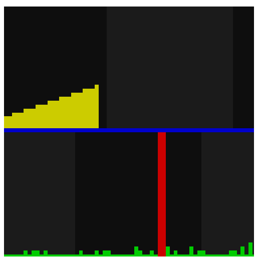
ClientStats

The ClientStats component is a utility class that was created towards the end of LibNRG's development. It holds statistics about the latency of Snapshots sent by the server – including whether any snapshot packets were dropped. In addition, it holds further statistics about whether or not the data that can be obtained through the Field's getInterp method is based on the interpolation of two received snapshots, or is having to be extrapolated due to a snapshot not arriving on time.

This class also contains a method named toRGBATexture which takes both sets of statistics and creates a texture corresponding to a “lagometer” that is based off the lagometer that can be found in the Quake series of video games [46]. This “lagometer” displays two bar graphs on top of one another that represent the previously described statistics. The top graph draws a bar every time the Client's update method is called. A blue pixel is drawn if the client is interpolating, or a yellow bar if it is extrapolating, with the height of the bar relating to how far ahead the extrapolation is. The bottom graph draws a green bar for every snapshot that is received by the server with height corresponding to the latency of the client that has been calculated by the server. If a snapshot is received with an ID that not exactly 1 more than the previous, then the client assumes a snapshot was dropped and a full height red bar is drawn to represent this.

These two images show a blank texture representing a good connection, and another that was having to extrapolate due to some packets being dropped:





This provides a simple view for library users and players to show them whether their game experience is being degraded due to a poor connection at a glance.

Replay

The Replay component of LibNRG allows a section of game-play to be recorded to a file that can then later be loaded back in order to re-watch what happened. This can be used as a tool for programmers to aid them in debugging their game or as an alternative to a video-capture of the game with the benefit of a typically smaller file size.

This functionality is implemented using two classes: ReplayRecorder which is responsible for creating the replay file and ReplayServer which is responsible for playing-back a supplied file in order to recreate the initial capture.

ReplayRecorder works by taking nrg::Packets that it is supplied, and writing them along with some metadata such as the packet's size to a file along with a header that identifies the file as an NRG replay. In order to reduce the size of the resulting file, ReplayRecorder uses the gzwrite function supplied by zlib instead of the standard C/C++ I/O functions when writing data. This function transparently compresses data as it is written to the file in the standard gzip format and reduces the file to approximately 35% of its original size [47].

To allow a replay to be recorded at any point in time, the ReplayRecorder requires to be supplied with the current state of all the entities that are known to the client when it is started. This is because the next packets that the client receives might be DeltaSnapshots with only partial state, and if this were the first packet to be recorded to the replay file then the initial state when viewing it back would not match the state when the replay was recorded. The client class provides a startRecordingReplay method that takes care of supplying the entities for the programmer behind the scenes.

ReplayServer operates by loading a replay file from the path supplied, checking its validity and then subsequently binding to a port in much the same way that the standard nrg::Server class does. It then waits for an nrg::Client to connect to the port at which point it begins reading packets from the replay file, adding the standard ConnectionOutgoing header, and sending them to the client – ignoring any input that the client sends in return.

An important facet of this design is that the client-side operates in exactly the same manner that it does when connecting to an actual nrg::Server as opposed to an nrg::ReplayServer. This means that the programmer does not need to implement specific code in their application to handle the case of watching back a replay other than loading the file in the first place – any client application will “just work” with the replay system.

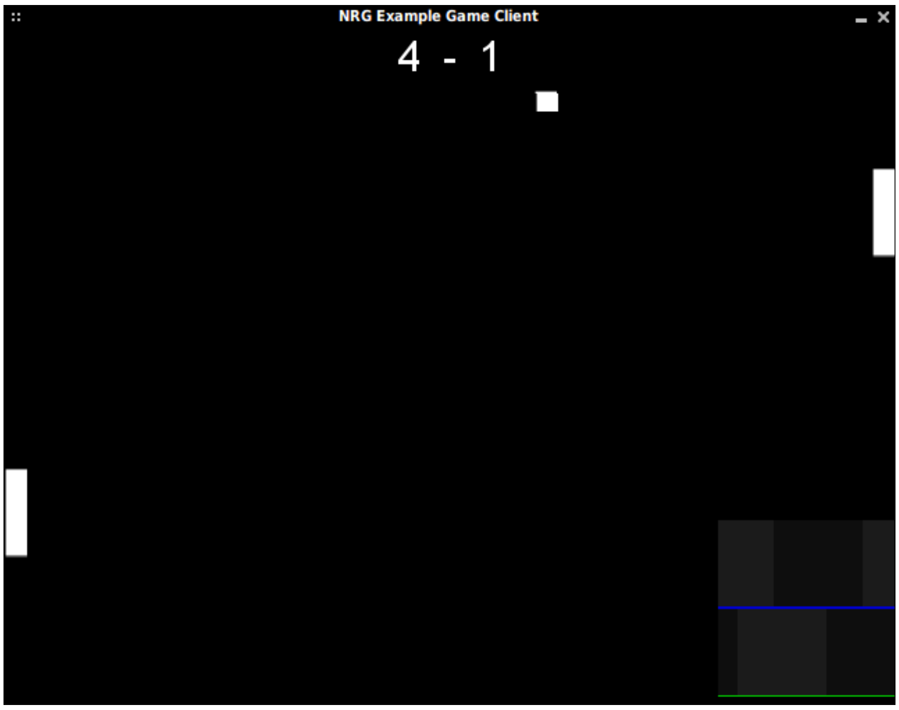
Design of the Example Game

The principle design methodology behind the example game was to create something simple, so that it could more easily be used as an aid for programmers to learn how to use LibNRG. Because of this, the classic Pong arcade game [48] was chosen as a theme for the example game to use.

The example game was developed as two separate applications that correspond to the client-side and server-side of LibNRG. The game is played by running an instance of the server application on some host and then subsequently connecting two instances of the client application to this host. The first and second connection will have control over the left and right paddle respectively, and any further connections to the server will act as spectators that can view the game without any control over it.

The previously mentioned LibSFML third party library was used in order to display graphics and collect mouse input within the client application since these were not functions that LibNRG on its own was designed to handle.

A screenshot of the example game is shown here:



In the bottom right hand corner of the screen, the texture that is obtained via calling nrg::ClientStats::toRGBATexture is presented as described in the ClientStats component design section. In this case, both graphs are flat lines indicating ideal network performance due to the client and server being run on the same host.

The code of the example game also demonstrates some important points about LibNRG such as the fact that it is still possible to use standard C++ inheritance in addition to deriving from an nrg::Entity. This is achieved by basing both the paddle and the ball on a single entity type called EntityBase that contains the fields common to these two entities - specifically the x and y positions - and then having the player entity contain an additional field for the score that the client displays.

Since the example game was designed to demonstrate as much of LibNRG as possible, it also contains the ability to record and play back sections of game play using the library's built in replay functionality, although watching a replay of a pong match could be considered a similar situation to watching paint dry.

Measurements / Results

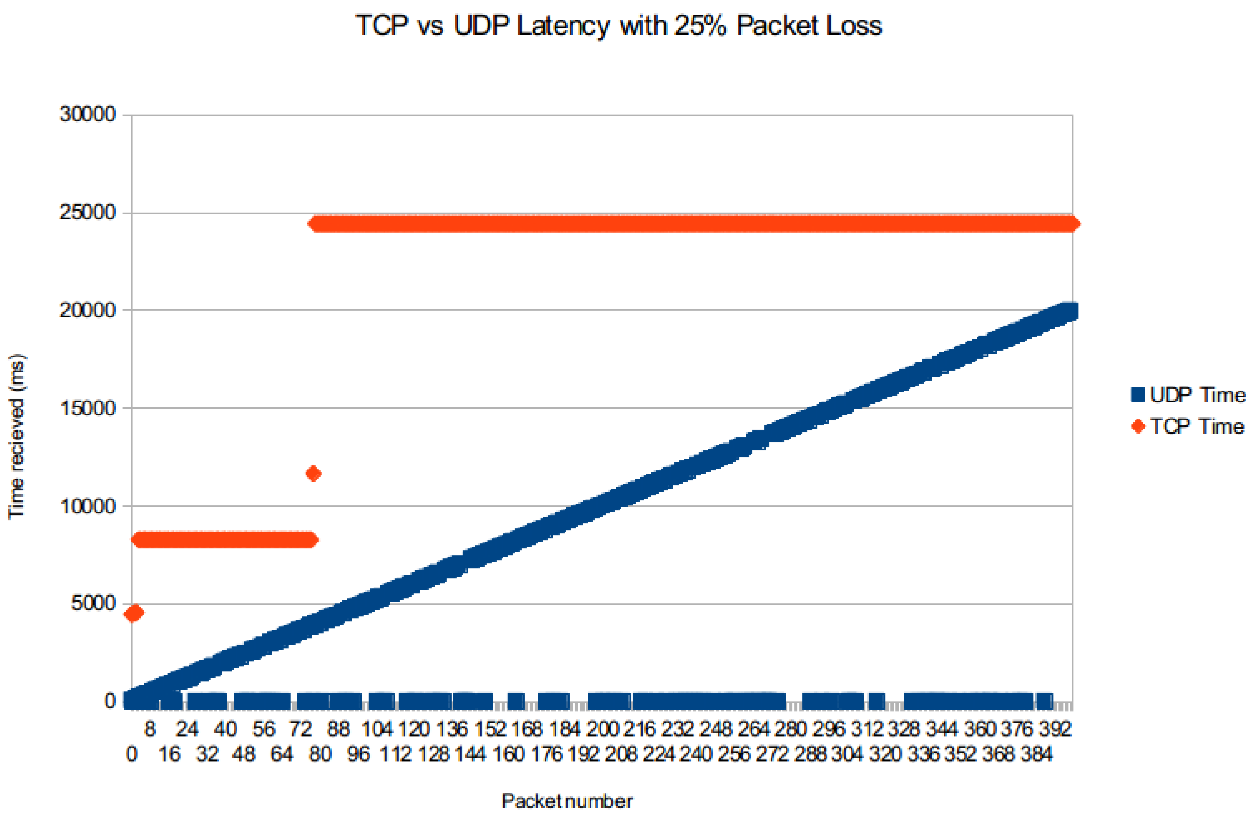
UDP vs TCP

In order to justify the use of UDP for LibNRG over TCP, a short test program was written that sends a series of 400 packets over the loopback interface and records the time at which they are delivered using both protocols.

To represent a worst case scenario, significant (25%) packet loss was added to the loopback interface by using the tc Linux command as follows:

```
# tc qdisc change dev lo root netem loss 25%
```

Results from this test were fed into a spreadsheet and the following graph produced:



Here it is possible to see that when TCP is used, packets arrive in bursts a significant amount of time after they are sent due to TCP's requirement to detect dropped segments and resend them, a phenomenon that would cause intolerable delays in a real-time game.

With UDP, the packets arrive immediately as one would expect – with the problem that some packets never arrive at all. This problem is tolerable within LibNRG however due to the small interval between subsequent snapshots. Since newer snapshots will contain more up to date information about the game state, there is not any point to providing a resend mechanism similar to TCP's – it would only induce additional latency. Instead, the client acknowledges snapshots in its user-input packets as previously mentioned, which lets the server know how much game state the client needs even when packets are dropped.

Latency

The latency of LibNRG is an important aspect to consider as it has been shown that players actively seek to play on servers with lower latency [49].

The most appropriate measure of the library's latency is the time that it takes between a user performing an action and the user seeing the result of that action on their screen.

In this scenario the user input must be recorded by the client application, be sent over the network to the server, be received and processed by the server and finally the updated game state needs to be sent back to the client which is then able to update its view for the user.

The minimum amount of latency present in this chain of events is the latency of the channel from client to server plus the channel latency of server to client, which would be present regardless of whatever networking software that the client and server are running. For simplicity, the latency of these two channels can be considered equal, however in practice there may be different latency in different directions depending on the underlying routing mechanism of the network – but this will not be accounted for. This channel latency can therefore be denoted as $2c$, where c is the channel latency of either direction.

Internally, the major factor affecting latency within LibNRG is the interval that the server component is set to send updates at – by default this is 50ms but it can be modified by the programmer – this can be denoted as *int* for the purpose of calculating the library's latency.

On the client side of the library, the latest set of user input will be sent to the server whenever the program makes a call to nrg::Client::update(), which is expected to be once per rendered video frame. Assuming that the program polls for input immediately before this call, then there will be a maximum latency of one video frame induced at this stage. For an application running at 60 frames per second, this value would be approximately 16.7ms, or 1000 / 60. However, this same latency would also be present in any application running on the same system, whether it uses the network or not, and can therefore be discounted when measuring the latency of LibNRG specifically.

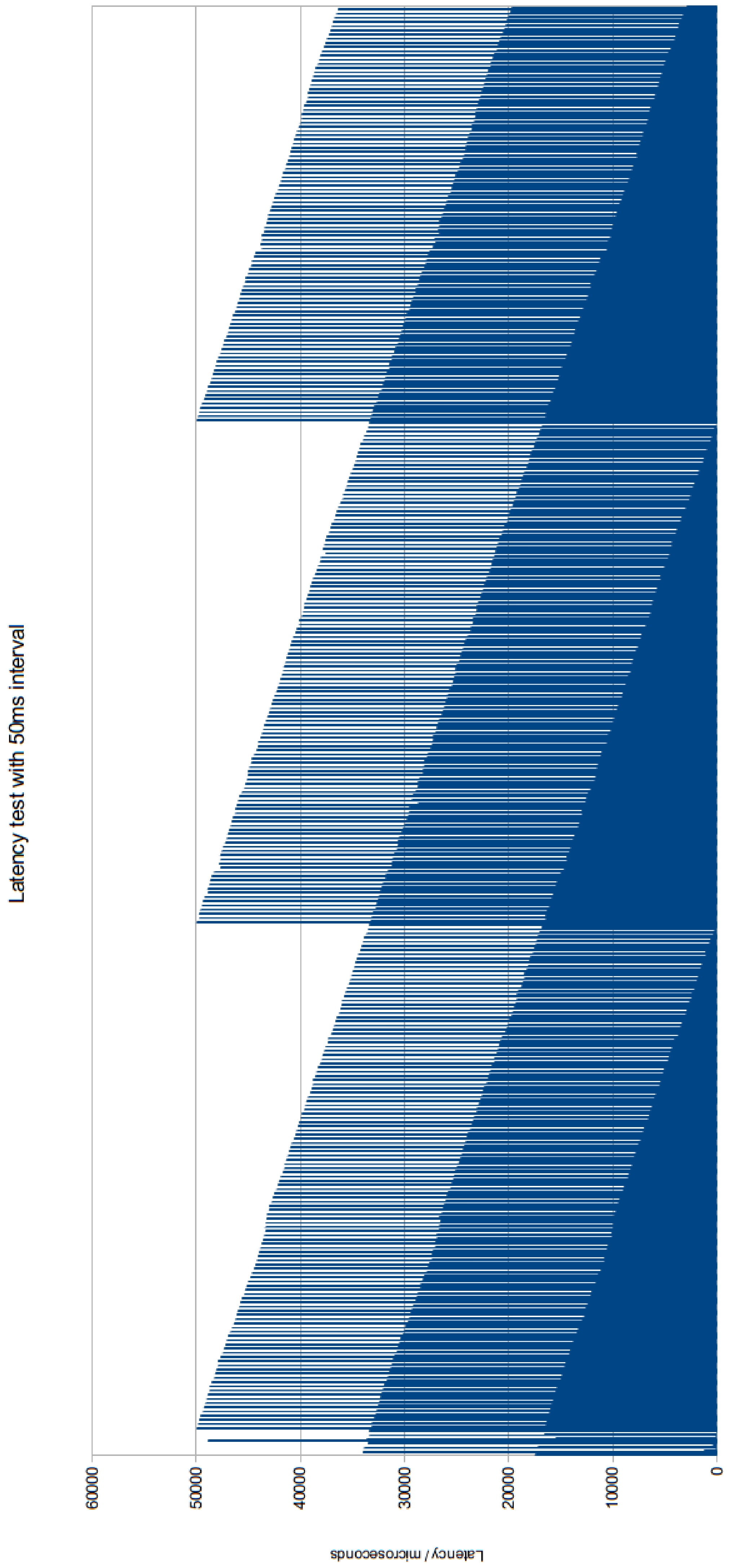
On the server side, a call to nrg::Server::update() will cause the server to perform calls to select() on its socket until the time to send a new update to clients arrives. The select function blocks until the specified amount of time has elapsed or until a new packet is available, in which case it returns immediately. The amount of time that the server is calling select will therefore vary with how much work is done by the server program between calls to nrg::Server::update - in the case of the example pong game the amount of time spent outside of this call is negligible; this fact means that the server will receive and process any received packet as soon as the operating system has received it and woken up the server process in its select call.

The server will then parse the received packet and call the nrg::Input::onUpdated callback immediately, which will update the server-side game state, meaning the amount of latency between the server receiving a packet and updating the games state is also negligible.

However, once the server's game state has been updated, it must still wait until the update interval has passed before sending out this new game state to its clients. In the worst case, the server will receive a client packet just after an interval has passed, in which case there will be a latency of *int* before the data is sent out to the client, and in the best case the server will get the packet just before an interval, giving close to no latency. On average, this waiting will give a latency of *int* / 2,

In theory, this gives LibNRG an average latency of 2*c* + *int* / 2, which is not much greater than the minimum possible latency of 2*c*, depending on the value that has been set for the update interval. However this approximation does not account for dropped packets, which may incur an additional latency of up to *int* for each successive dropped packet in the worst case.

To determine the latency of LibNRG in practice, the time between the server receiving an input packet and sending out the next snapshot was recorded over a short period of time, and each record was then plotted on a graph. This graph is presented on the next page.


Latency test with 50ms interval
Latency / microseconds
60000
50000
40000
30000
20000
10000
0

This graph is consistent with the theory in that the maximum latency is approximately 50ms, the same as the interval at which the server was set. In addition, the average latency is also approximately half of this value at 25ms as the theory predicted. For this case, the channel latency is not measured as it is wholly dependent on the particular connection between the client and server hosts.

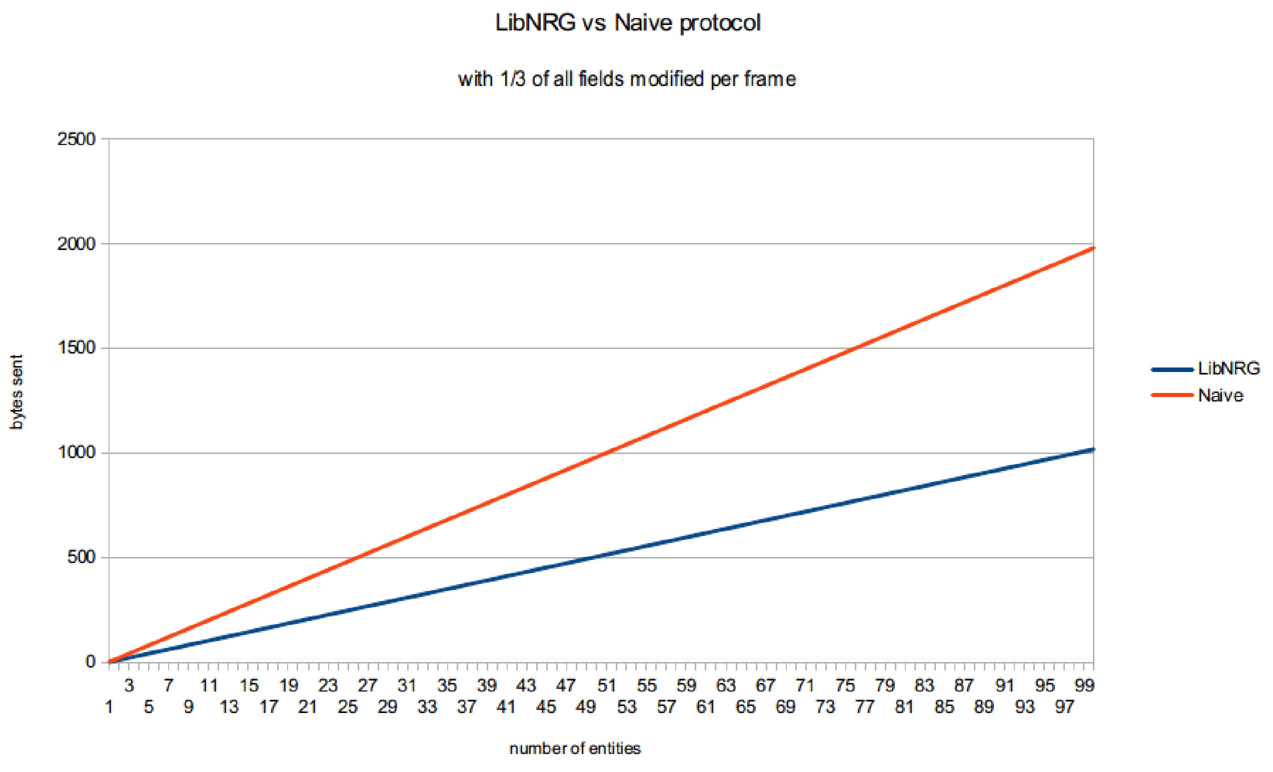
The graph's sawtooth shape is due to the differing rates at which client and server send out packets – ~16ms for the client and 50ms for the server. Since the rates are not divisible, the latency will gradually drift downwards until it gets sends a packet that arrives just after the server updates, at which point it will experience the worst case latency.

The reason that consecutive packets have such different latencies is because the client can fit ~3 input packets into every 50ms interval, some of which will be nearer it's beginning or end than the others.

Bandwidth usage

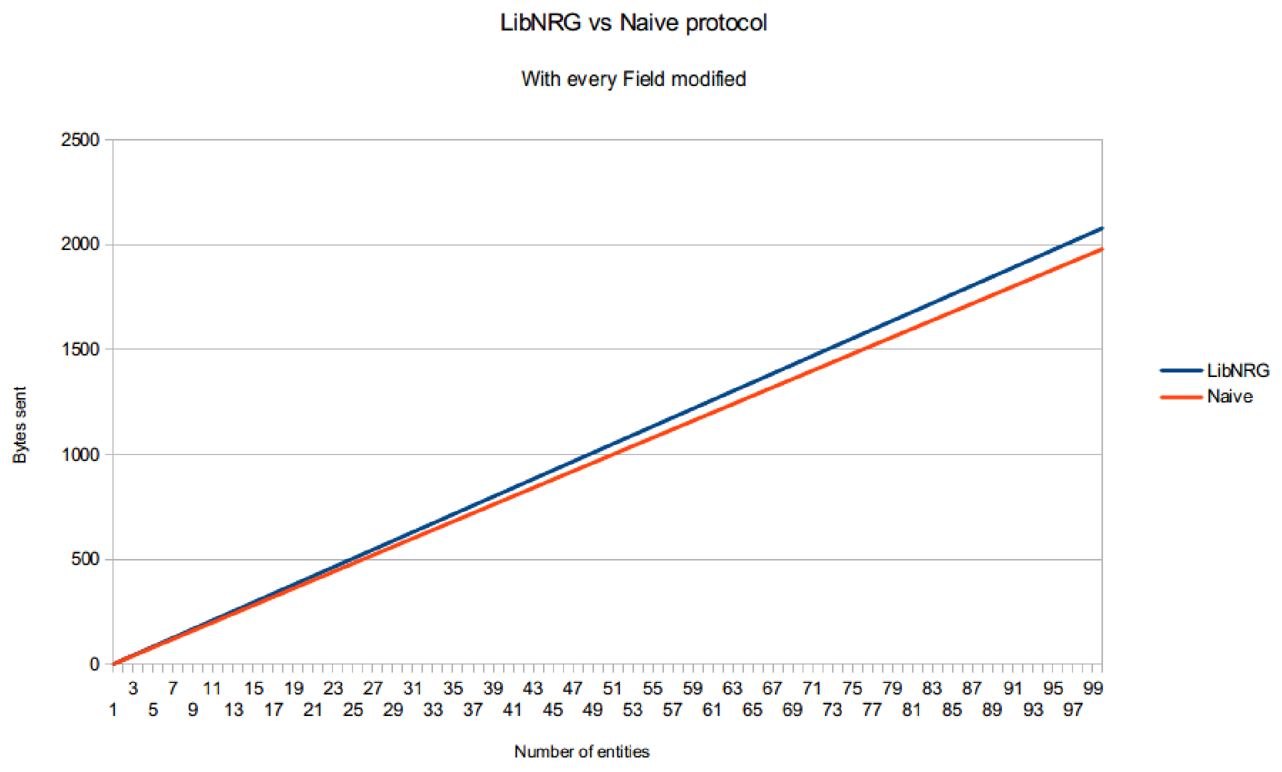
The amount of data sent by LibNRG is minimized by its Entity and Field system that only sends changes in state to clients. In order to determine the effectiveness of this approach, the number of bytes transmitted over the network can be analysed in scenarios in which different numbers of Fields are modified on the server-side each frame, and compared to the naïve approach of sending the full game state.

The two methods are compared in a situation with increasing numbers of entities, each of which contains 4 Field<int> values. In the first case, 30% of all fields are modified by the server at each update period. The graph for this situation is shown here:



This graph shows that in comparison with the naïve protocol, LibNRG requires roughly half as much bandwidth to serialise the data. Compression is not taken into account in these tests, since it would have an approximately equal effect for both approaches.

In the situation where <u>every</u> Field in the application is modified each frame, LibNRG does not perform as well, however:



This is due to the fact that LibNRG's protocol includes a bitmask for each Entity that informs the client which fields have been modified that is not necessary in the naïve approach, which causes the library to use slightly more bandwidth, albeit in the rare case that every single field was modified this frame. This also justifies the decision not to use the more advanced approach in the Input class, since all its fields are likely to change each frame, as previously mentioned.

Evaluation

The library

At the projects completion, LibNRG currently provides a good set of functionality that can successfully be used to implement networking in video-games as was originally intended. The choice of UDP as its basis has brought it good advantages, and the inherent disadvantages of UDP's unreliability has been well considered, dealt with, and made transparent to the users by means of the Entity and Field high-level abstraction.

LibNRG makes good use of well-defined components to add to its code re-usability and provides programmers with features such as interpolation of data between snapshots that would otherwise be difficult to create from scratch. In addition, LibNRG's design gives it low latency and reasonable bandwidth efficiency as demonstrated.

It's major downside is currently a lack of proper, thorough documentation other than the example game and this report due to the documentation objective unfortunately being neglected. The library is currently also quite light on its feature set due to having only around 20 weeks of development put into it. Both of these shortcomings can fortunately be addressed through further work, however.

The example game

The example game provides a good demonstration of how LibNRG can be used, including how to create entity classes that can be used in conjunction with the library's high-level abstraction, and how to handle events that are raised during its operation. It also shows off the optional features of LibNRG such as the replay file recording and playback functionality and the diagnostic lagometer that can be drawn using the ClientStats class that comes as part of the library.

One drawback of the example game is its simplicity. Although it was intentionally designed to be very basic to allow for users to learn from it, the game could have added some complexity in order to test out the library's scalability by being more than a two player game. Since scalability was explicitly mentioned as something that LibNRG would not be focused around in the specification, the lack of more than 2 players is fair, but it would have made the game more interesting regardless.

The creation process

Creating LibNRG and its example game has been a process that was at times challenging and stress inducing, but at others rewarding - such as seeing the example game working for the first time.

The main good choices that were made regarding the project's management were making heavy use of the git version control system from its beginning and using a development process that involved iteration and refinement of components over time.

More debatable choices were using the gedit text editor for development instead of an IDE that may have increased productivity with time-saving features. On the other hand, unfamiliarly with such IDEs may have led to more time being wasted in order to learn such features.

Lessons learned

Looking back on the development process of LibNRG and it's example game many lessons have been learned that would most likely lead to changes if the project were to be undertaken again. I believe a greater focus on user-testing may be beneficial in order to determine which features need to be added or which features are not necessary in a network-related library like LibNRG, instead of simply relying on my own views of what I personally would want.

I also feel that the library could have been designed with a greater focus on separating implementation from interfaces by using constructs such as the abstract factory pattern to create instances of the Client and Server classes. This would also ensure a greater binary compatibility for the library as it evolves.

A greater focus on providing thorough documentation and more detailed test cases for the library as it was being created would have also been beneficial, perhaps even choosing a test-driven development model where tests are the first thing to be created.

Further work

Throughout the development of LibNRG many additional features and improvements have been considered that can be made to the library in the future. The aforementioned lack of Doxygen documentation is, first and foremost, something that can be addressed in future, perhaps even before the library is released to the general public.

Other features such as support for Entities that are only sent to certain players could prove useful for certain types of game, and the addition of separate networking models such as a peer-to-peer architecture into LibNRG is certainly something that can be considered going forward.

The replay system within LibNRG is also an area that can be expanded upon in future, in order to provide functionality such as pausing, rewinding, slow motion or even rendering to a standard video format. A simple modification to the system could also allow multiple people to watch a replay together simultaneously.

Another good area to work on in future is creating another demonstration game that is more fully featured and stands up on its own right. This would allow the weaknesses of LibNRG to become more apparent, and would provide insight as to more useful features that can be added if they are needed by the demonstration game.

Acknowledgements

All code present in LibNRG and the example game was written solely by myself.

Some components within the library draw influence from the Quake 3 engine and the associated Quake 3 architectural overview written by Fabien Sanglard [7] such as the idea for Delta-snapshots and the lagometer that can be drawn by the ClientStats class. However the implementation of these features within LibNRG is completely my own work and distinct from those in Quake 3.

The zlib 3rd party library [38] is used by LibNRG in order to optionally compress packets with the PacketCompressor class, and is also used in order to compress and uncompress the replay files that are written by the ReplayRecorder class and read by the ReplayServer class.

The SFML 3rd party library [6] is used by the example game in order to draw its graphics and collect mouse input events.

References

[1] OGRE "OGRE – Open Source 3D Graphics Engine" [Online] Available: http://www.ogre3d.org/

[2] Elmindreda "GLFW – An OpenGL Library" [Online] Available: http://www.glfw.org/

[3] Epic Games, Inc "Game Engine Technology by Unreal" [Online] Available: http://www.unrealengine.com/

[4] IEEE and The Open Group "Base Specifications Issue 7" [Online] Available: http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_10 Section 2.10

[5] Sam Lantinga et al. "SDL_net 1.2" [Online] Available: http://www.libsdl.org/projects/SDL_net/

[6] Laurent Gomila "Simple and Fast Multimedia Library" [Online] Available: http://www.sfml-dev.org/

[7] Fabien Sanglard "Quake 3 Source Code Review" [Online] Available: http://fabiensanglard.net/quake3/index.php

[8] Jenkins Software LLC "Raknet – Multiplayer game network engine" [Online] Available: http://www.jenkinssoftware.com/purchase.html

[9] Jörg Rüppel "Zoidcom Network Library" [Online] Available: http://www.zoidcom.com/index.html

[10] Valve Corporation "STEAMWORKS" [Online] Available: http://www.steampowered.com/steamworks/

[11] Dimitri van Heesch "Doxygen: Main Page" [Online] Available: http://www.stack.nl/~dimitri/doxygen/

[12] Valve Corporation "Demo Recording Tools" [Online] Available: https://developer.valvesoftware.com/wiki/Demo_Recording_Tools

[13] GitHub "GitHub" [Online] Available: https://github.com/

[14] MinecraftCoallition "Protocol - MinecraftCoallition" [Online] Available: http://mc.kev009.com/Protocol

[15] D Pittman "Cheat-Proof Peer-to-Peer Trading Card Games" IEEE Press Piscataway, NJ. ISBN: 978-1-4577-1934-9

[16] Bettner, Paul, and Mark Terrano. "1500 archers on a 28.8: Network programming in Age of Empires and beyond." *Presented at GDC2001* 2 (2001): 30p.

[17] Forrest Smith “Synchronous RTS Engines and a Tale of Desyncs” [Online] Available: http://www.altdevblogaday.com/2011/07/09/synchronous-rts-engines-and-a-tale-of-desyncs/

[18] Bursztein, Elie, et al. "OpenConflict: Preventing Real Time Map Hacks in Online Games." *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011.

[19] The Wassenaar Arangement “List of Dual-Use Goods and Technologies” [Online] Available: http://www.wassenaar.org/

[20] Ruby “Ruby programming language” [Online] Available: http://www.ruby-lang.org/en/

[21] GNU “GCC, the GNU Compiler Collection” [Online] Available: http://gcc.gnu.org/

[22] GNU “GNU Make” [Online] Available: https://www.gnu.org/software/make/

[23] The Wireshark Foundation “Wireshark – Go Deep.” [Online] Available: https://www.wireshark.org/

[24] GNU “GDB: The GNU Project Debugger” [Online] Available: https://www.gnu.org/software/gdb/

[25] Valgrind Developers “Valgrind Home” [Online] Available: http://valgrind.org/

[26] Stroustrup, Bjarne. “The Design and Evolution of C++”. (2001) Addison-Wesley. ISBN 0-201-54330-3.

[27] Cowan, Crispin, et al. "Buffer overflows: Attacks and defenses for the vulnerability of the decade." *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*. Vol. 2. IEEE, 2000.

[28] Max Vlimpoc. “Measuring latency in the Linux network stack between kernel and user space.” [Online] Available: http://vilimpoc.org/research/ku-latency/

[29] IEEE and The Open Group “Base Specifications Issue 6” [Online] Available: http://pubs.opengroup.org/onlinepubs/009696799/basedefs/sys/socket.h.html

[30] Linux manual page “inet_ntop(3)” [Online] Available: http://man7.org/linux/man-pages/man3/inet_ntop.3.html

[31] SGI “map<Key, Data, Compare>” [Online] Available: http://www.sgi.com/tech/stl/Map.html

[32] Cohen, Danny. “On holy wars and a plea for peace.” *Computer* 14.10 (1981): 48-54.

[33] Dan Saks “Why size_t matters” [Online] Available: http://www.embedded.com/electronics/blogs/programming-pointers/4026076/Why-size-t-matters

[34] IEEE and The Open Group “stdint.h – Integer types” [Online] Available: http://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdint.h.html

[35] Braden, Robert. “Requirements for Internet hosts-communication layers.” (1989). [Online] Available: http://tools.ietf.org/html/rfc1122

[36] Hornig, Charles. *RFC 894:* “Standard for the transmission of IP datagrams over Ethernet networks”. RFC, IETF, April, 1984.

[37] Postel, Jon. “Transmission control protocol.” (1981). [Online] Available: http://tools.ietf.org/html/rfc793

[38] Jean-loup Gailly and Mark Adler. “zlib - A Massively Spiffy Yet Delicately Unobtrusive Compression Library” [Online] Available: http://zlib.net/

[39] IEEE and The Open Group “pselect, select - synchronous I/O multiplexing” [Online] Available: http://pubs.opengroup.org/onlinepubs/009695399/functions/select.html

[40] Lakos, John. “Large-scale C++ software design.” *Reading, MA* (1996). ISBN-10: 0201633620

[41] Crockford, Douglas. “The application/json media type for javascript object notation (json).” (2006). [Online] Available: http://tools.ietf.org/html/rfc4627.txt

[42] Kulesh Shanmugasundaram “Linux Kernel Linked List Explained” [Online] Available: https://isis.poly.edu/kulesh/stuff/src/klist/

[43] Abrahams, David; Gurtovoy, Aleksey. “C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond”. Addison-Wesley. ISBN 0-321-22725-5.

[44] Sam Lantinga et al. “Using the Simple DirectMedia Layer API – Events” [Online] Available: http://www.libsdl.org/intro.en/usingevents.html

[45] Laurent Gomila “sf::Event Class Reference” [Online] Available: http://www.sfml-dev.org/documentation/2.0/classsf_1_1Event.php

[46] Marco Von Ballmoos “Understanding the Lagometer” [Online] Available: http://earthli.com/quake/lagometer.php

[47] Jean-loup Gailly and Mark Adler “zlib 1.1.4 Manual” [Online] Available: http://www.gzip.org/zlib/manual.html

[48] David Winter “Pong Story: Arcade Pong” [Online] Available: http://www.pong-story.com/arcade.htm

[49] Armitage, Grenville. “An experimental estimation of latency sensitivity in multiplayer Quake 3.” *Networks, 2003. ICON2003. The 11th IEEE International Conference on*. IEEE, 2003.

Appendix

Source code (on CD)

The source code for LibNRG, the example “ball duel” game, the UDP vs TCP test program, and the interpolation demo is provided on the CD that accompanies this report.

LibNRG's code is stored in the directory of the same name, with its public header files in the include folder and source cpp files along with private header files in the src folder. The example game's source code is stored in the LibNRG/examples/ballduel/ folder whilst the interpolation demonstration is in the LibNRG/examples/interptest/ folder. The LibNRG/examples/misc/ folder contains the two small test cases that were written prior to the example games development began. Each item also contains a README in its directory with more information.

All this code can be compiled using the provided makefiles in the respective directories. The example game and interpolation test require the LibSFML library in order for their client components to compile and run. On Debian/Ubuntu LibSFML can be easily installed by running apt-get install libsfml-dev. On other Linux systems such as those in DCS, the library will have to have its header files and shared libraries placed somewhere GCC can find them by adding -I and -L parameters to GCC in the makefiles. A tarball of LibSFML is provided on the CD in the 3rd Party directory for convenience.