Extending a Closed-Source Game with Multiplayer Functionality

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science.

by

Mark Harmer

Dr. Bobby D. Bryant, Thesis Advisor

December 2010
We recommend that the thesis prepared under our supervision by

MARK HARMER

entitled

Extending A Closed-Source Game With Multiplayer Functionality

be accepted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Bobby D. Bryant, Ph. D., Advisor

Sushil J. Louis, Ph. D., Committee Member

Jennifer Mahon, Ph. D., Graduate School Representative

Marsha H. Read, Ph. D., Associate Dean, Graduate School

December, 2010
Abstract

The process of adding additional functionality to closed-source applications is generally not well known. This thesis covers the motivation, tools and design for extending a game with multiplayer functionality. The purpose of this thesis is to introduce the background concepts and methods needed towards implementing functionality changes within any application.

Since games attract a variety of fields relating to graphics, user interfaces, storage, artificial intelligence, networking, audio, etc. they are best suited to display the ability to add additional functionality. Games are also a great motivational tool towards developing extended functionality because the results can easily be shown to people that are not experts on the application.

The game of Torchlight is chosen as a real-world target application for added functionality. Multiplayer functionality is added to the otherwise single-player, closed-source game. Multiplayer on the game of Torchlight is shown to be working.
Acknowledgements

I would first like to thank my advisor, Dr. Bobby Bryant for his support through this thesis. Additionally I want to thank my committee, Dr. Sushil Louis and Dr. Jennifer Mahon. I would also like to thank Runic Games for making such a great game to base my work off of and the awesome Torchlight player community for their constant motivation.

My thanks to my friends for their distractions (both good and bad) from this project. And finally, I want to thank my family for their continued support.

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Torchlight is trademark of Runic Games.
## Contents

Abstract ii  
Acknowledgements iii  
List of Figures v  
1 Introduction 1  

2 Background 3  
2.1 Torchlight 3  
2.2 Genre History 6  
2.3 Modding 7  
2.4 Motivation for Multiplayer Extension for Torchlight 7  
2.5 Reverse Engineering 8  
2.5.1 High- to Low-level Code 8  
2.5.2 Low- to High-level Code 9  
2.5.3 Call Stack 10  
2.5.4 Virtual Functions 11  
2.5.5 RTTI and the vtable 12  
2.5.6 Function Pointers 14  
2.5.7 Patching Code at Run-Time 14  
2.5.8 Trampolining 14  
2.5.9 Dynamically Linked Libraries 15  
2.6 Previous Work 15  

3 Torchlight API 17  
3.1 Motivation 17  
3.2 Overall Design 17  
3.2.1 Re-creating Data Layouts 17  
3.2.2 Event System 18  
3.3 Methodology 19  
3.3.1 Memory Locations 19  
3.3.2 Hardware Breakpoints 19  
3.3.3 Obtaining Class Data Layouts 20  
3.3.4 Software Breakpoints 21  
3.3.5 Obtaining Function Details 21
3.3.6 Accessing Function Arguments ........................................ 23
3.3.7 Patching Mechanism ..................................................... 24
3.3.8 Event System: Exposing Event Callbacks .............................. 25

4 Torchlight Multiplayer ......................................................... 27
  4.1 Purpose ................................................................. 27
  4.2 Overall Design .......................................................... 27
  4.3 Networking .............................................................. 28
  4.4 User Interface ............................................................ 29
  4.5 TLAPI and Callback Functions ........................................... 31
  4.6 Town Phase .............................................................. 31
    4.6.1 Character and Item Synchronization ............................... 32
    4.6.2 Chat Interface ..................................................... 35
  4.7 Dungeon Phase .......................................................... 36
    4.7.1 Level Transfers .................................................... 37
    4.7.2 Synchronizing Interactable Objects ............................... 37
    4.7.3 Attacking .......................................................... 38
    4.7.4 Character Deaths ................................................ 39
    4.7.5 Character Resurrection .......................................... 39
    4.7.6 Experience ......................................................... 40
    4.7.7 Placement new .................................................... 40
  4.8 Debugging .............................................................. 41
  4.9 Overview ............................................................... 43

5 Results ................................................................................. 44
  5.1 Town Phase .............................................................. 44
  5.2 Dungeon Phase ........................................................... 44
  5.3 Overall Playability ....................................................... 47
  5.4 Future Work ............................................................ 47

6 Conclusion ............................................................................. 48
List of Figures

2.1 A screenshot from the canonical single-player game of Torchlight in the Town level. The player’s character is always shown in the center of the screen. 5
2.2 Compilation Process from a C++ input file to the Assembly code and final executable. 8
2.3 A screenshot of the IDA Pro software used to aid in the reverse engineering process. 10
2.4 A representation of a stack frame for a function. level, x, z are arguments passed into the function. The return value is stored so the function knows where to send program control when complete. Local variables needed within the function are stored on the stack as well. 11
2.5 C++ class and vtable layout in memory. 13
2.6 Partial RTTI layout and pointer to the RTTICloneCompleteObjectLocator from the vtable. 13
2.7 Example of trampolining a function call into the intended, original function from the target application. 15
3.1 Memory layout of a CList class discovered in Torchlight. The CList class is one of the more simple classes discovered and re-engineered. 18
4.1 A screenshot of the multiplayer options screen. 30
4.2 A screenshot of the multiplayer join screen. 31
4.3 A screenshot of the multiplayer lobby screen. 32
4.4 Multiplayer within the Town. The player’s character is shown in the center of the image and the second player’s character is shown on the left. 33
4.5 Sequence diagram of the initial messages between the server and client game instances. 34
4.6 Item synchronization as seen by the server player. Three items have been dropped to the ground by the client character. 35
4.7 Item synchronization as seen by the client player. The same three items are shown to the other player, displaying item synchronization. 36
4.8 A door (trigger unit) is highlighted and available to a player to interact with. 39
4.9 The same door shown as opened by the other player, displaying the synchronization of the trigger units. 40
4.10 Microsoft Visual Studio running as the Just-in-Time debugger for Torchlight after a crash occurred. The disassembly of the instructions around the current instruction are shown, as well as memory of the stack. 42
Chapter 1 Introduction

Many games are built with only single-player capabilities. Generally, these games lack multiplayer capabilities because the developers chose to keep the design single-player only; lack of development time and resources may also be a factor. The amount of time a player may want to invest replaying a game can be described as replayability. Single-player functionality can be a defining factor for replayability because of the predictive nature of the game mechanics and entities. The only opponent a player encounters is a computer controlled entity, which tends to only show a static pattern of behavior. A player may play through a game once or twice before they become familiar with how the computer responds. Supplementing these computer controlled players with actual humans adds social interaction and much of the predictive nature is removed.

Adding multiplayer capability to any game is not only technically challenging, but requires a fundamental design change of how the game will be played. No longer are all the characters idly standing by waiting for the human player to interact with them; human players can be added to the world to interact with other players. Design changes are required to enforce how multiple players can interact with the environment cooperatively and competitively.

This thesis shows the techniques and results of an undertaking to add multiplayer functionality to the game of Torchlight, created by Runic Games. Torchlight is a commercial, closed-source game. For commercial game business models, the source code is usually not released, which complicates any game functionality modifications such as adding multiplayer functionality. It is shown here that the process involved with supplementing an existing game with added functionality requires several underlying, connected processes.

Previous work on adding multiplayer to the closed-source game Grand Theft Auto III, is described in Chapter 2. Background information relating to the game of Torchlight with an overview and an explanation of the major game mechanics are also discussed. The genre of Torchlight and a brief history is also described. An overview of the existing modification system for Torchlight, what can be done with it and why it is not suitable for supporting multiplayer is discussed as well.
The largest problem for adding functionality to a closed-source game is reverse engineering it and gaining an understanding of the underlying mechanics. Tools exist for aiding in this process, but the time spent researching the game mechanics at a low level eclipses the time spent developing functionality. The overall design, technical implementation of the design and compiler nuances must be taken into account when analyzing the game. The background on the necessary tools and processes involved are discussed in Chapter 3.

Once a sufficient amount of knowledge is gained through reverse engineering, an interface can be created to allow later code to use the game elements in an intuitive way. This interface provides the ability to read and modify game variables as the game is running, allowing a variety of future functionality to be more easily added to the game. Examples of such future functionality might include: the ability to change keyboard settings in-game, adding keyboard control for moving a character, or even changing the behavior of the artificial intelligence. The interface was designed to be used easily by future programmers and is described in more detail in Chapter 3.

The multiplayer modification described here is built upon the existing player interface to Torchlight. The multiplayer modification makes use of existing open-source libraries for the actual message formatting and communication between the players’ separately running games. The multiplayer modification acts as a driver for connecting the underlying components to work together. Additionally, a user interface is built using the same open-source library Torchlight uses. Chapter 4 describes the process for adding multiplayer functionality to Torchlight.

The results of this investigation show that changing the functionality of closed source, commercial projects is possible and can be successfully accomplished. Results of adding multiplayer functionality to Torchlight are discussed in Chapter 5.
Chapter 2 Background

In this section the background of the game of Torchlight, the genre, motivation for adding multiplayer and previous work are discussed. Technical background details needed for implementing multiplayer are also presented.

2.1 Torchlight

Torchlight is an action role-playing game (ARPG) created by Runic Games [1]. ARPGs generally have a common game mechanic of allowing a player to control an in-game character that represents the player. The main purpose of the player’s character in ARPGs is to control it to kill enemy characters — mostly monsters — and collect items. In addition to characters, items are also a main element of ARPGs. Items allow the player to equip them onto their character to increase their attributes and visual appearance. Items that creatures drop when killed are referred to as loot, such as swords, armor or potions. These items can be picked up and equipped. As the player kills creatures that are increasingly higher levels they will obtain more powerful items. Items typically offer attribute point bonuses when they are equipped on the character.

Character attributes are also considered a key part of many RPG games. Multiple attributes exist so that the game designer can allow for different playstyles that are not dependent upon a single attribute. For example, strength and dexterity are two common attributes found in many RPG games. Generally, strength is considered to be a melee combat oriented attribute, while dexterity offers increased ranged attacks. The separation of these attributes offers a variety of differences in play, and allows the player to choose more options to customize their character.

Character classes are a staple component of ARPGs. Character classes are designed to differ in how they are played within the game. Some classes may be considered to be the hard hitting melee type, another may use a bow and ranged attacks, and others may use minions and magic. Adding a variety of classes allows a player to go back through the game once completed to explore other aspects and playstyles, adding onto the game as a
new experience and providing for more replayability.

Many ARPGs are based within fantasy worlds. Magic is usually a prevalent part of the fantasy game genre. Character classes in such fantasy genres have a magic attribute as well. Within the game of Torchlight, magic is available in a quantity similar to the character’s health, called mana. Mana is required for a player to use their skills and spells. It is limited in quantity for a duration until replenished, therefore must be used carefully.

Skills are considered to be unique to each character class. Skills are usually set up in a dependency tree so that the more powerful skills can only be used later in the game after the player has accumulated enough skill points. Characters within an ARPG generally have a set of skills that attempt to make the character unique and add replayability. Torchlight incorporates a skill tree, so that when the player gains a level, the player is allowed to place a skill point into the tree. As the character level and skills increase, the character becomes increasingly more powerful.

ARPG characters are generally shown in a 3rd-person, top-down camera style as shown in Figure 2.1. The player issues movement and other commands to their avatar by clicking on the ground with the mouse pointer where they want to go. Players progress through the game by killing monsters and gaining experience points. As a player gains experience points, at certain thresholds they will gain a new level. Each new level allows the player to purchase increased attributes and skill points.

The ARPG genre is based heavily on replayability, since the basic gameplay can become monotonous. Replayability in Torchlight is offered through three different character classes, a quest dungeon with thirty-five levels and a near-endless dungeon that is randomly generated each time with a large number of unique levels.

Many ARPGs contain a currency, which the player uses to trade with friendly non-player characters (NPCs) that are controlled by the computer. Items must be purchased with the form of currency in the game. For Torchlight this is gold; selling any items to vendors will yield gold in return. Creatures that are killed drop gold. As with item drops, the level of the creature killed determines the amount of gold that is dropped.

Quests are yet another component of ARPGs. Quests are offered through friendly NPCs. Quests act as an additional motivational tool for the player to continue through
Figure 2.1: A screenshot from the canonical single-player game of Torchlight in the Town level. The player’s character is always shown in the center of the screen.

the game. Story-line elements of ARPGs are usually narrated through quests; the story is driven through quest progression. Quests are pursued by the player after they choose to accept them. Quest objectives usually concern finding a NPC, and killing it if unfriendly, to obtain a special quest item. Once the objectives of a quest are completed, the player must return to the designated NPC to retrieve their reward and the next quest. Quest rewards within Torchlight consist of gold, experience points and fame.

Similar to gaining experience points, a player can also gain fame points. A player’s level of fame increases when thresholds of fame points are reached. Fame is awarded to players that complete quests, and kill champion creatures - which are considered to be more powerful than the usual creatures the player faces. Boss creatures also award fame and are considered even more powerful than champions, and are generally tied into the story. The role of fame within Torchlight is to supply an additional skill point to the player as they reach each level of fame.
2.2 Genre History

The first graphical adventure game was called Mystery House (On-Line Systems, 1980). It was considered the first game to use “real” graphics, employing lines and not simple character art [2].

The first ARPG was titled Dungeons of Daggorath (DynaMicro, Inc., 1982) [3]. This was one of the first games to bring real-time games and action RPG elements together.

The method that Torchlight uses of moving the player’s character by clicking the ground is referred to as point-and-click. The method first appeared in a graphic adventure game called Enchanted Scepters (Silicon Beach Software, 1984) [4]. Point-and-click finally gained in popularity within ARPGs when the game of Diablo (Blizzard Entertainment, 1996) was released.

A sub-genre of the ARPG is what is referred to as the hack and slash genre. Hack and slash games focus on combat and killing monsters. They are usually played at a faster pace than traditional ARPGs, but still have the root elements of a story driven game, including NPCs.

Major players in the ARPG genre were the Diablo and Diablo II games that used the ARPG element with a point-and-click, hack and slash style of gameplay. The Diablo franchise is still played to this day; the multiplayer availability of the Diablo games beginning in 1996 is the main reason replayability remains.

Torchlight was created by the same designers as the first two Diablo games, and thus is similar in nature to both. An element carried over by another designer from the game of Fate (WildTangent, 2005) was the player’s pet that would follow the player, hold items and fight [5]. The concept of a pet, or minion, is a concept taken from canonical RPGs.

The multiplayer genre of ARPGs has its roots in Multi-User Dungeons (MUDs). The first MUD dates back to 1978, and was consequently called MUD, later renamed to MUD1 to differentiate it from the MUD genre [6]. MUDs usually contain text-based commands and descriptions of the virtual world. In contrast, modern RPG games usually display the virtual world to the user through 2D or 3D graphics, rather than through text descriptions. The first graphical MUD was Habitat (Lucasfilm Games, 1985) [7].
2.3 Modding

Torchlight encourages modifications, or mods, to the game and has built a reputation on its modding capability. This allows for new content to be added to the game by the player community and promotes replayability within the game. Examples of such content could be: new levels, new character classes, new items, etc. Torchlight has a freely available editor called TorchED to supply users with the ability to easily create new levels, place content into levels, create logic triggers for traps and lever systems, and set up monster spawns. Spawning a monster is the act of creating it on-the-fly and visually to the user, as the game is in progress. For example, a monster can be created after a barrel is destroyed to make it appear the monster was inside the barrel.

Torchlight has an active user community which creates and plays on a large range of mods for the game. However, the modding support in Torchlight only supports new or modified content; underlying functionality to the game cannot be easily changed. The TorchED tool has no capabilities for changing the logic of the game, or for exposing the game functionality easily.

2.4 Motivation for Multiplayer Extension for Torchlight

Due to the quick development time on Torchlight, multiplayer functionality was not added. Because of the rich history of multiplayer ARPGs players now expect the social experience afforded to them in previous games. The player base of Torchlight wanted to play with their friends in the next room or across the Internet. Multiplayer functionality offers replayability advantages; players could potentially find others to join a game with and fight side-by-side to kill monsters. Players could also join games and discuss elements of the game or trade items. Multiplayer offers a social experience within the game and keeps players coming back to play with others.

The player response and the challenge of creating multiplayer functionality drove the motivation for the initial work on a multiplayer modification.
2.5 Reverse Engineering

This background focuses on the technical background required for developing towards multiplayer functionality within Torchlight.

Reverse engineering (or simply, reversing) is the process of reconstructing how something works [8][9]. For the remainder of this thesis, “reverse engineering” will refer to the process of determining how the software of Torchlight works.

2.5.1 High- to Low-level Code

High-level programming languages are used almost exclusively in development for modern games. High-level programming languages offer programmers code that is easier to write and read, and in most cases reduces the number of bugs produced versus implementing software in a low-level programming language such as assembly. Torchlight was written in C++, as determined from the use of the Ogre and CEGUI libraries [10][11], and by analyzing the executable.

The high-level language, in this case C++, is passed through a compiler. The compiler will transform the high-level C++ code into low-level assembly code\(^1\) as shown in Figure 2.2. Assembly code is used because it is easily translated into machine code by an assembler. Machine code is considered to be the final output of the program-building process for the CPU (Central Processing Unit) to recognize and execute. The C++ language was designed to be more human readable, in contrast to the assembly code generated, which is a lot larger because each instruction does a simpler task.

![Compilation Process diagram]

Figure 2.2: Compilation Process from a C++ input file to the Assembly code and final executable.

Comments made by the programmer are stripped out by the compiler and are not included in the assembly code output produced. Additionally, if the compiler flag for including

\(^1\)Some compilers will skip the assembly code generation and just create machine code.
debugging symbols is off (such is the case with Torchlight) any variable names are lost, and function and class names are usually lost.

2.5.2 Low- to High-level Code

Translating machine code back to C++ is considered hard [12]. An automatic tool to do the translating is called a decompiler. All existing decompilers can only reverse trivial code. Humans are needed to translate the assembly code into a high-level C++ construct. Several tools exist to aid in this process:

**Disassemblers** take a program as input and display the assembly code to the user as output. Disassemblers are useful for static analysis, which is the act of analyzing a program’s logic without actually running it. Static analysis tools will generally try to do as much grunt work with the assembly code as it can to aid the user in determining interesting program features. Interesting features are dependent upon what is being done with the executable, but usually they are: function locations, function argument types and count, and cross references to other functions that call a function.

**Debuggers** also take as input a program and will display assembly code. The difference is they generally do not add additional information to the display output. They are useful for dynamic analysis, which is running the program and allowing the user to investigate values at run-time. This is considerably helpful when tracking down a specific function.

The Interactive DisAssembler (IDA) [9] was used as an initial static analysis tool for retrieving the assembly code from the binary and creating basic function information. IDA contains debugger functionality and was used during the execution of Torchlight to further aid in reversing. IDA is a commercial disassembler with older freeware versions available. IDA is heavily used today for reverse engineering [13]. A screenshot of the IDA Pro software is shown in Figure 2.3.

---

2Class names will sometimes not be stripped due to C++ Run-Time Type Information (described in Section 2.5.5). Function names will sometimes not be stripped because of C++ name-mangling for DLL import and export functions.
Figure 2.3: A screenshot of the IDA Pro software used to aid in the reverse engineering process.

2.5.3 Call Stack

The call stack on the x86 CPU is memory reserved for functions, it is also referred to simply as the stack. The stack is used in most cases through the push, pop, call and retn opcodes. These opcodes will insert (push) or remove (pop) values from the top of the stack. When functions are called, most of the parameters of the function are pushed onto the stack. Additionally, a call opcode will push the return address onto the stack, and transfer control to the new function. The retn opcode will pop this return address from the stack and return control the calling function. The stack always grows downwards in memory, as values are pushed onto the stack the ESP CPU register is automatically decremented to point to the top of the stack [14].

The stack within C++ is comprised of stack frames (or, activation records). Each stack frame contains information for the lifetime of a single function. The stack frame for the function is active between when the first argument is pushed onto the stack until the

\[3\] The order in which function parameters are pushed onto the stack or set through registers is what is referred to as the calling convention.
function returns control. The size of this stack frame is dependent upon the number of the local variables used within the function, the number of arguments passed in and the return address. An example stack frame is shown in Figure 2.4.

![Stack Frame Diagram](image)

Figure 2.4: A representation of a stack frame for a function. `level`, `x`, `z` are arguments passed into the function. The return value is stored so the function knows where to send program control when complete. Local variables needed within the function are stored on the stack as well.

### 2.5.4 Virtual Functions

Virtual functions are used in many high-level programming languages today. They were used heavily in Torchlight for most of the classes. Virtual functions are member functions that can change in functionality depending upon the class used in the inheritance tree. They offer ease of programming because the derived class does not need to be known; common virtual functions to a shared base class work regardless of what derived class is being used. Example code that uses virtual functions is shown in Listing 2.1. Reverse engineering a virtual member function used across several classes can help realize the behavior of the code. The use of virtual functions is also important for reversing class names as shown in Section 2.5.5.

Classes that use inheritance, if implemented properly, contain virtual destructors. Virtual destructors are useful within C++ when `delete` is used on a base-class pointer. Instead of automatically calling the destructor for the base class, the virtual destructor allows the correct destructor for the derived class to be called. This allows proper memory cleanup to occur that would otherwise not happen if the derived class had dynamically allocated its own memory.

If a class contains at least one virtual function, a virtual function table (vtable) is
```cpp
class A {
public:
    virtual int getNumber() const { return 1; }
};

class B : public A {
public:
    virtual int getNumber() const { return 2; }
};

int main()
{
    A* obj1 = new A();
    A* obj2 = new B();

    obj1->getNumber();  // returns 1
    obj2->getNumber();  // returns 2

    delete obj1, obj2;
}
```

Listing 2.1: Example of C++ code that uses virtual functions.

created by the compiler to hold member function pointers to the correct class member functions, as shown in Figure 2.5. Since the vtable’s address in memory is static (always the same) the compiler can easily access it to gather the virtual function address. If a class has a vtable, any classes that reference that same vtable are considered to be the same class type. The virtual destructor is normally the first virtual function within the virtual function table.

### 2.5.5 RTTI and the vtable

Run-Time Type Information (RTTI) in C++ allows for additional class information and inheritance support for code at run-time; in most other high-level languages this is referred to as reflection [15]. RTTI is needed if a `dynamic_cast<>` operator is used, which does a class-type inheritance cast and allows for type-safety usually afforded by the compiler, but at run-time. Since the `dynamic_cast<>` operator is used in Torchlight, RTTI is enabled [16].

RTTI contains the name and inheritance information for each class that has a virtual function. A pointer to a structure called the RTTICCompleteObjectLocator is placed before the vtable in memory [17] as shown in Figure 2.6. IDA has support through the Class
Informer plugin for statically analyzing RTTI information and populating the vtable assembly output with information on the class, such as the name of the class and the inheritance from other classes [18].

The name of the class can be obtained at run-time when using RTTI, as it is contained within the information the compiler supplies. Obtaining the class name does not directly help in the reversing process. However, when variables of the class type are passed through functions it offers a huge benefit in determining what data the function is potentially using and the purpose of the function.
2.5.6 Function Pointers

Like regular C++ pointers, function pointers simply contain a memory address. However, function pointers contain addresses to code, in contrast to regular pointers which contain addresses to data. Function pointers are checked for type-safety at compile-time. Function pointers are useful when reverse engineering because at the executable level functions exist at specific memory locations. When these functions within the application being reversed are discovered it is helpful to sometimes call these functions outside of the control of the target application.

2.5.7 Patching Code at Run-Time

Patching refers to replacing original code with different instructions to change the logic of the application. The original code that is replaced is either replicated elsewhere, or simply discarded if it is not needed. The term hooking is similar in nature to patching, however it generally refers to attaching to existing, original code.

Microsoft Detours [19] is a freely available 32-bit\(^4\) library for hooking existing functions. Detours does not have the capability of patching instructions other than function calls. Another library, Dyninst, exposes several patching capabilities for applications at run-time [20].

2.5.8 Trampolining

Trampolining is a mechanism by which a function call is re-routed to a different location in memory. This is useful for re-routing code that would normally execute an original function, but instead would route to a hooked function. The hooked function then has the ability to process its own code before deciding to call the original function, or simply return without ever calling it [21][20]. The term trampolining comes about because the hooked function acts as a trampoline and bounces a function call to the original function. This process can be seen in Figure 2.7.

\(^4\)Detours also has a 64-bit version that is available commercially.
2.5.9 Dynamically Linked Libraries

Programs frequently re-use the same code. To avoid building the same code into every application, which can create large programs, dynamic libraries are used to hold a single binary image of commonly used code. Programs that wish to use this common code can allow the operating system to load any dynamic libraries the program requires when started. The process of loading these dynamic libraries into memory and allowing them to communicate is the job of the dynamic linker [22]. Because of the requirements to dynamically link up the application and DLLs, functions that are used between the two are exposed. Functions that are exposed to dynamic linking are placed within import and export tables. These tables can give an indication of what functions in the libraries are being used by the target application.

2.6 Previous Work

A similar project of adding multiplayer to the closed-source single-player game, Grand Theft Auto III has been created through the Multi Theft Auto (MTA) modification [23][24]. MTA has similar ambitions of this project, including established code. Due to the intricate nature of MTA and Torchlight, the existing code was not used or modified for Torchlight Multiplayer. Much of the initial Torchlight Multiplayer work was investigating the executable to discover how it worked. Because the target applications were completely different this research could not be carried over from elsewhere. MTA uses a common networking library
called RakNet [25] that is also used in this project. The majority of work involved is reverse engineering the game mechanics, which differs considerably between the two games.

Similar projects, such as the Broodwar API (BWAPI) for the game of Starcraft, exist for programmers to more easily interface with the game [26]. BWAPI is attached to the game when it is started and exposes functions and classes. These functions and classes can be used to read and modify values of the game as it is running. For example, the units of a game within Starcraft could be obtained from a simple function call. The API for Torchlight Multiplayer discussed in Chapter 3 is similar in overall purpose to BWAPI.
Chapter 3  Torchlight API

3.1 Motivation

An Application Programmers Interface (API) was created to interface with the Torchlight game. The purpose of this API is to supply a clean interface to any programmer wishing to interact with the game. Currently, the multiplayer implementation only interacts with the API and not the game itself. This interface is called the Torchlight Application Programmers Interface (TLAPI). Both the TLAPI and Torchlight Multiplayer are open-source projects.

3.2 Overall Design

The primary design for TLAPI is to re-create the C++ classes used in Torchlight and expose these through an interface to other applications. This interface allows for other applications to read and write any of the class information. Simply reverse engineering Torchlight’s classes is not enough. There must be a mechanism in place to ‘hook’ into the game as it is running to extract and insert information while Torchlight is running. To support this design, patching the Torchlight code to access such information is required and will be discussed in Section 3.3.7.

3.2.1 Re-creating Data Layouts

C++ classes contain member functions and variables. The variables are considered the data and the functions are considered the operators. For each instance of a class, the operators are the same - only the data changes, therefore only one function is used and shared for every instance of that class. Each instance of a class is differentiated by the this pointer. The only difference between class instances are the data that is operated on, therefore the compiler will generate a class layout of the data; and all instances of the class will use this layout for their variables (data). This allows the member function (operator) to always use the same offset to the data for each class instance it works on.

Reverse engineering the class layout for each class is required for progressing with TLAPI. Classes contain pointers to other classes and structures within the layout. For
example, a list and wstring structure were discovered in several of Torchlight’s classes. The Torchlight CList class was used in several classes and thus was discovered as a general structure shown in Figure 3.1. The first element within the CList class points to an array of objects. As different Torchlight classes were investigated it became apparent that the list pointer would point to different array object types. Because of the generalized nature of pointing to varying object types, it was determined this class was most likely templated, and was re-created as such.

Figure 3.1: Memory layout of a CList class discovered in Torchlight. The CList class is one of the more simple classes discovered and re-engineered.

3.2.2 Event System

An event system was created to interface with external code using TLAPI’s patched functions. The event system makes use of function pointers to supply callback functions when Torchlight calls its own functions. The underlying method of hooking this up to Torchlight is discussed later in Section 3.3.7. Any attached code interfacing with TLAPI can register their own functions to be called when certain functions in Torchlight are used. This allows the external code access to the function arguments, the this pointer (if it is a member function) and the return value.
3.3 Methodology

The following contains the methods used for obtaining the class data layout and interesting functions. Only a small subset of all the functions in Torchlight are looked at, for the sake of time and towards implementing a multiplayer modification.

3.3.1 Memory Locations

Once Torchlight is running, all the code and data it uses are available in memory. Beginning to reverse engineer any program is initially difficult: where do you start?

Since the Torchlight code is so large it is generally better to jump into the game and look at some interesting values. For instance, the player’s character contains data values for its health. Simply reading the health value from the game’s interface one might see the health value read 60 for the player’s character. Using this information, a memory search could be performed. A memory search can be performed by an external tool that simply reads and searches the memory of the process. Armed with the value of 60, a search for the value in memory can be executed. The memory search will return the address of any locations it found with the value of 60\(^1\). As there is usually more than one memory location with the same value, additional searches within the results are needed. It is usually more productive to continually modify the health value in the game and rerun the memory search to narrow down the location faster.

3.3.2 Hardware Breakpoints

Hardware breakpoints first require an explanation on breakpoints. Breakpoints are used in debuggers to break the program execution at a specific point. Generally, software breakpoints are used for code and hardware breakpoints for memory. Hardware breakpoints are checked with the CPU (hardware) whenever a write or read/write is performed on a memory location [14]. From the previous example, a memory location could be obtained for the player’s health. While Torchlight is still running, a debugger (such as IDA) can be attached to investigate code and data further. Once a debugger is attached to the process

---

\(^1\)It is important to know the datatype of what you are searching, a float with the value of 60 is represented differently than an integer in memory.
the memory location obtained can be viewed, in the case of the example the memory would contain the value 60.

A hardware breakpoint can be placed on this memory address with it set to write. The power of the hardware breakpoint and debugger come into play; every time the game writes to the memory of the player’s health, the hardware breakpoint will step in and halt the program at the exact memory location of the executing code. The point at which the process was halted is within a function that is performing an update to the player’s health. Further investigation into what arguments, return values and code semantics would be needed to determine the exact purpose of the function.

### 3.3.3 Obtaining Class Data Layouts

Based on what is known from the C++ class data layout the health data is residing in the class data memory containing the player character. Exploring around the memory location of the player character’s health and looking at memory values would obtain a clearer picture. During TLAPI development, the location of other character attributes were obtained using this method. Exploring around the memory location also allows for faster class layout recreation by removing the need to use the memory search tool for each variable. Additionally, variable values may not be known to the player and would be nearly impossible to find otherwise. Values at run-time displayed under IDA’s debugger for a portion of the player’s character data can be seen in Listing 3.1.

```plaintext
// Player ECX = 083A7450

// @Offset = 394h
debbug233:083A77E4 dd 348.0 ; Current Health
debbug233:083A77E8 dd 168h ; Maximum Health
debbug233:083A77EC dd 0
debbug233:083A77F0 dd 348.0
debbug233:083A77F4 dd 0
debbug233:083A77F8 dd 4 ; Dexterity
debbug233:083A77FC dd 0Ah ; Strength
debbug233:083A7800 dd 0Ah ; Defense
debbug233:083A7804 dd 3 ; Magic
debbug233:083A7808 dd 25.0 ; Current Mana
debbug233:083A780C dd 19h ; Max Mana
```

Listing 3.1: Values shown at run-time of the player’s health and other attributes nearby.
The entire data layout could be investigated by moving backwards until a vtable pointer was encountered for the character class. The vtable pointer always exists as the first data element in a class with at least one virtual function.

Certain data elements of the layout might be pointers. Pointers can easily be followed inside a debugger into a different address of memory. In most cases these pointers will result in jumping directly to another class’ data layout. A pointer to a class will always point to the first element, which would be the vtable if it is a polymorphic class. The vtable would then give access to the RTTI information and hence the class name.

The TLAPI code was developed as elements of layouts were discovered. For TLAPI, specification files were used to hold each class’ layout as a C++ struct type. Classes were not used because the function locations would not align with Torchlight’s in memory.

The preprocessor macro `#pragma pack()` [27] was used to ensure the compiler packed the data into 1 byte alignments to enforce the exact layout of the Torchlight class. Otherwise, when compiling TLAPI the compiler would create unexpected padding for data elements within the structure. This simple modification prevents inadvertently throwing variable offsets that are defined past the padding out of sync with Torchlight’s layout.

### 3.3.4 Software Breakpoints

Software breakpoints are used for memory that contains executable code. Extending upon the previous example of obtaining the function where the player health was updated; the beginning of the function could be found through the use of a debugger or disassembler. A software breakpoint could be placed at the first instruction within the function. Instead of using the hardware breakpoint to simply break when one class instance’s health was updated, the function will now break anytime any of the class instances call it. In effect, it will halt the program when any character’s health is updated, in contrast to halting only when player character’s health is updated.

### 3.3.5 Obtaining Function Details

Continuing further with the example, Torchlight can be run with the software breakpoint in place. The debugger breaks as Torchlight calls the character health update function and
allows function arguments to be viewed.

Torchlight uses the Microsoft Visual C++ compiler (MSVC). This compiler will generate consistent assembly code when passing arguments into functions. The simplicity of calling a function in C++ requires additional work in assembly code. The compiler must generate prolog and epilog assembly code for any functions. The compiler will also generate additional assembly code at the location of where a function is called. Different calling conventions exist to determine how to allow other sections of code access to the variables. For almost all of the functions in Torchlight the _thiscall convention is used because they are member functions of a class. The _thiscall functions take a hidden argument, the this pointer to differentiate the class data locations.

Different compilers will generate different assembly code to allow the calling mechanism of C++. For _thiscall, MSVC passes the this pointer through the ECX register and pushes all arguments from right-to-left order onto the stack [9]. Lastly, it will push the address of the code onto the stack so the function knows where to return control when completed. Finally, a jump to the memory location of the function is taken. An example of a C++ call is shown in Listing 3.3 and the equivalent assembly call in Listing 3.2.

| mov ecx, [esi+2Ch] | ; Move the CCharacter pointer into ecx |
| push ebx          | ; Push the low 32-bits of skillGuid |
| push ebp          | ; Push the high 32-bits of skillGuid |
| call CharacterUseSkill | ; Push return address onto stack and jump |

Listing 3.2: Assembly instructions for passing arguments to the function.

CCharacter->CharacterUseSkill(u64 skillGuid);

Listing 3.3: C++ instruction for calling a function.

Once the jump is taken into the function address, the code within the function prolog handles extracting the arguments that the function uses that were pushed onto the stack. The stack uses a special register called ESP that will point to the current position of the top of the stack, as items are pushed and popped off ESP is updated. Since all the arguments

\footnote{The assembly instruction call handles the pushing of the return address and the jump in one instruction.}
(except for this in ECX) are pushed on the stack, to extract arguments ESP and a known offset to the variable\(^3\) can be used.

To determine the number of arguments the function takes, the assembly instruction \texttt{retn} can be investigated. \texttt{retn} will pop the value off the top of the stack which is the memory location to return control to the calling function. The \texttt{retn} instruction can also have a value as an operand that will tell the CPU how many elements to pop off the stack. This value is the indication of the total size (in bytes) of arguments that were passed to the function. Additionally, local variables are cleaned up prior to a \texttt{retn} by simply subtracting ESP by the total size, thus moving the stack pointer down in the stack.

The \texttt{retn} value indicates the total size in bytes of the arguments passed into the function. Using this knowledge, all of the arguments passed to the character health update function can be obtained. Once the software breakpoint that was applied for the character health update function is hit, the stack can be investigated for the argument information; the memory location in ECX (this pointer) can also be analyzed. ECX will indirectly give the vtable, RTTI and the class name. The arguments to the functions can be either memory locations (pointers), built-in types (\texttt{int, float}, etc.), or user-defined types.

MSVC also returns any values through the EAX register at the end of the function. Looking at the value of EAX can sometimes yield the return type of the function. An example of output from IDA of a simple function that returns the pointer to the player character is shown in Listing 3.4.

```
.text:0052D190 sub_52D190 proc near ; CODE XREF: sub_41FC50+677
.text:0052D190 ; sub_424340+65
.text:0052D190 mov eax, dword_E1A5A8
.text:0052D195 retn
.text:0052D195 sub_52D190 endp
```

Listing 3.4: Example of a function returning a value through the EAX register.

### 3.3.6 Accessing Function Arguments

Function arguments are pushed onto the stack. The state of the stack just after a call to a function contains the return address as the top element, with arguments to the function in

\(^3\)The compiler will sometimes generate BP-based code that uses the EBP register to hold the base of the stack frame. Local automatic variables are sometimes referenced through an offset from EBP.
left-to-right order moving down the stack.

Local variables are given space after a function is called by simply incrementing ESP by the total size in bytes. Because the compiler will push and pop values throughout a single function to store and retrieve memory the value of ESP will change constantly throughout a function. However, the compiler knows exactly where a specific argument or local variable is relative to the current value of ESP and will access these arguments through a memory read from [ESP+(relative offset)+(variable offset)]. Additionally, the compiler may use the stack base pointer stored in EBP to obtain local variables. Snippet code for a small example of how this is used is shown in Listing 3.5 and the equivalent assembly output as shown in IDA in Listing 3.6.

```cpp
int stackTest (int & a, int b);    
int main ()
{
   int a = 5;
   return stackTest(a, 4);
}
```

Listing 3.5: Example C++ code that uses the stack to pass variables into the stackTest function.

### 3.3.7 Patching Mechanism

Existing libraries for patching a process were presented in Section 2.5.7. Because of the open-source and transparency of development on the project, a user contributed library for patching code was developed and used prior to either of the previous works being discovered. The process by which all of the libraries use is essentially the same.

The Windows operating system uses a strict process for searching for DLLs when loading an executable [27]. The first location a DLL uses is the same path the application resides in. Taking advantage of this fact, a rogue DLL can be created with the same name and placed within the directory of the application. In the case of TLAPI, a DLL is chosen that is found in a later search path so as to not overwrite the original DLL.

In the case of TLAPI the DLL name chosen was winmm.dll; the original winmm.dll contains commonly used code for multimedia activities. Torchlight has no knowledge of the rogue winmm.dll being loaded and expects any function calls to winmm.dll to proceed
and return normally. To appease Torchlight so that it does not crash, these function calls are trampolined (Section 2.5.8) into the original winmm.dll; any results are returned back through to Torchlight. In this system, the rogue DLL is acting like a silent intermediary passing information back and forth. The only functional difference is that the rogue DLL gains the ability to run its code when it is loaded into the process.

### 3.3.8 Event System: Exposing Event Callbacks

With the underlying patching functionality in place, it was necessary to keep the outward facing API clean\(^4\). With the implementation of the patching/hooking designed to be as general as possible, the `this` parameter and other arguments are not type enforced by the compiler. Their types are considered to be simply 32-bit integers — whether or not they are pointers or part of a structure passed by-value — this allows for dangerous code: if function arguments do not have a type then the programmer using the API must always cast, and in doing so may use the wrong cast and cause a segmentation fault when using as a pointer later.

\(^4\) A “clean” API is considered to be simple to understand and easy to use.
function was also desired.

To simplify both of these approaches, the implementation consisted of three C++
macros:

**EVENT_DECL** Contains a large portion of code creating a declaration, designed to be
used within a C++ Class.

**EVENT_DEF** Handles static `std::vector` definitions that were created the EVENT_DECL
macro. These vectors hold the function pointers for any registered callback function.

**EVENT_INIT** Wrapper for patching the function, to be used for each patched Torchlight
function.

Both EVENT_DECL and EVENT_DEF are designed to enforce compile-time type
safety with the arguments. The EVENT_DECL in Listing 3.7 handles all the type-casting
so that the API exposes the correct types of the function. When functions are registered
the compiler will check to make sure the function contains the arguments with the same
type. This also offers anyone building on TLAPI to use the functions where the types are
intuitive, therefore reducing bugs and development time.

```cpp
EVENT_DECL(CCharacter, void, CharacterUpdateHealth,
(CCharacter*, float, bool&),
((CCharacter*)e->this, *(float*)&Pz[0], e->calloriginal));
```

Listing 3.7: CCharacter’s CharacterUpdateHealth event declaration. Line 2 contains the
required argument types for the registered function to copy. Line 3 contains the specific
casting mechanism from the patching API to the types described on line 2.
Chapter 4  Torchlight Multiplayer

The purpose of this chapter is to show how to create additional functionality for Torchlight that is simple to implement on the TLAPI. As functionality for the multiplayer modification is needed TLAPI is revisited and revised to include support. The name for this modification is Torchlight Multiplayer (TLMP).

4.1 Purpose

The purpose of Torchlight Multiplayer is to use TLAPI to create, modify and delete game elements at run-time. In the case of a two-player scenario, there are two separate Torchlight games running that can be considered independent simulations of the same environment. The concept of multiplayer ensures that two or more players of a single game share the same experiences. Therefore, the multiplayer modification must insure that these separate games can talk to each other and share information between players. If information is inaccurate or late it can create a differing experience for one player, e.g. perhaps a monster is in a different position or a sword does not have the same attributes. These differing experiences, depending upon the magnitude, can lead players to become detracted from the game. Therefore the primary purpose of multiplayer is to keep the players’ experiences as synchronized as possible.

4.2 Overall Design

The design of the game follows a client-server architecture. This architecture ensures that one instance of the game acts as the server and will drive the simulation for any clients that are connected. This also allows for easier implementation when determining which game instance controls the characters. In most cases the client is designed to request actions that they wish to perform from the server. This allows the server to control most of the game and respond to the clients with a valid response. The exception is when a player moves their character on a client instance, the client will inform the server of the movement and not wait for a response, to reduce the visible latency and make the game more interactable.
To more easily define and test progress for TLMP, the final task of adding multiplayer functionality was split into two phases: a Town phase that contains a more simple game with no combat or enemy characters, and a Dungeon phase that contains the Town phase plus combat, level transfers, loot and monster generation, boss encounters, etc. The town within Torchlight is considered a safe-area, while the dungeon is the combat aspect of the game.

A prototype lobby system to more easily chat and find games was also created. A lobby is comprised of a dedicated remote server that is very simple: it accepts connections and pulls in basic player information like their name, and supplies it and any chat to other players that are connected as well. Additionally, the lobby server’s primary purpose is to collect information on games that are currently available to join. This information would otherwise need to be listed somewhere or given out on a per-individual basis. The lobby server does not route information from games themselves, it simply acts as a matchmaking service for those wishing to host and join.

4.3 Networking

The network library used by the modification for communicating between two computers was RakNet [25]. RakNet is a networking library that is free to use for open-source projects. It uses the user-datagram protocol (UDP) and builds control and reliability on top, as needed by the application. Pushing and pulling data into/from RakNet’s buffers can be error-prone and tedious for handling different messages. Google’s open-source Protocol Buffers [28] (protobuf) was used to handle formatting the messages sent over RakNet.

Protobuf allows for a clean setup of message formats defined in a .proto file. This .proto file is pre-processed to generate specification and implementation files. From these generated code files, Torchlight Multiplayer can benefit from the easier message handling and only worry about pushing data through the compiled type-safe message functions provided by protobuf. An example of a character creation message defined in a .proto file is shown in Listing 4.1.
message Character {
  required int64 guid = 1;
  required string name = 2;
  repeated Character minion = 3;
  required Position position = 4;
  optional int32 id = 5;
  required int32 health = 6;
  required int32 mana = 7;
  required int32 level = 8;
  required int32 strength = 9;
  required int32 dexterity = 10;
  required int32 magic = 11;
  required int32 defense = 12;
  repeated string skills = 13;
  required int32 alignment = 14;
}

Listing 4.1: Protobuf formatted input file for describing a character in a network message.

4.4 User Interface

Support for easier human interaction with the added multiplayer functionality was added through a User Interface (UI) system. Conveniently, Torchlight uses Crazy Eddie’s Graphical User Interface (CEGUI) library, which is open-source [11]. CEGUI contains a UI manager for handling all the windows and events that an interface may need. Windows are described in an XML-formatted file, called a layout. These window layouts contain information for window location on the screen, its size, and its type (dropdown box, editbox, button, label, etc.). Each window has a unique name string attached to it that allows it to be referenced programmatically.

The same instance of the CEGUI UI manager that Torchlight uses can also be accessed from TLAPI. Torchlight is dynamically linked at runtime with the CEGUI library [22]; dynamically linking TLAPI with CEGUI will use the same library state and functions that Torchlight uses. CEGUI makes use of a singleton design pattern for the UI manager, allowing only one instance of the class to be instantiated. This allows TLAPI to access the same class that Torchlight uses and to modify the layout programmatically.

The UI manager under CEGUI was used to load additional interface windows to Torchlight through specific layout files for the multiplayer modification.

Once the XML layout files are loaded into CEGUI, each window can be retrieved
programmically through the unique name string given to it. This allows the use of CEGUI’s internal event system when events occur in a window, such as when a user clicks a button. Other attributes of the window can also be set or retrieved. For example, the text of an editbox window can be retrieved by referencing the unique name given to it. Retrieving text allows the user to input string values to the system; this can be used if a player is inputting text via a chat interface, or entering information about which server they wish to connect to. The XML layout files use both Torchlight’s internal interface and images, in addition to an open-source layout and image set provided through Ogre and CEGUI called OgreTray.

Currently the modified UI has a multiplayer button added to the existing main menu of Torchlight. Clicking this button allows the user to enter a multiplayer setup screen to either host a game or join an existing game. The host setup screen sets options for which network port the game should listen on. This information is translated to the RakNet library when the host (server) is started via a host button event. Both of these screens can be accessed through the multiplayer options interface as shown in Figure 4.1.

![Figure 4.1: A screenshot of the multiplayer options screen.](image)

A setup screen for joining an existing server is available as well. Joining has an additional option for specifying which host to connect to, this can be either an IP address or a
domain name in addition to a port. This is shown in Figure 4.2. The lobby interface for chatting is shown in Figure 4.3.

![Figure 4.2: A screenshot of the multiplayer join screen.](image)

### 4.5 TLAPI and Callback Functions

Since TLAPI allows registered functions to be invoked when the original Torchlight functions are called, functionality is gained from TLAPI to suppress the original functions from being executed. This mechanism allows, for example, halting any character creation done on a client instance of the game. It also allows halting level changes, item creation, item drops or pickups, attacks, trigger unit activations, etc., which must be controlled by the server during a multiplayer game.

### 4.6 Town Phase

The first phase of implementation was limited to inside a safe level, where combat is not allowed by the game. This level allows the player to interact with non-player characters (NPCs), which are controlled by the computer and are considered friendly characters in the level. Since the town within Torchlight is fairly limited in player interaction, it makes a
Figure 4.3: A screenshot of the multiplayer lobby screen.

A good starting point for adding multiplayer functionality.

Since this implementation phase was limited to the town, the game forces players to load into the town via a patched function in TLAPI. Function executions are suppressed when a player attempts to move outside of the town, preventing them from leaving the safe area.

### 4.6.1 Character and Item Synchronization

Character and item creation need to be suppressed on the client. This is because dynamic memory pointers are not the same across two instances of the game; the operating system will assign the memory block of a player character to a different location each time Torchlight is run. The difficulty is determining how to synchronize the characters between the games; how does the client know which character the server is talking about?

A naive approach might involve checking the character names; if they are the same then it might be determined that the characters are the same. However, two players could enter the game with the same name. Additionally, Torchlight contains many monsters in the dungeon game that share the same names.
The approach used in TLMP was to simply turn off character creation altogether for the client. The server then controls what characters are created, where they are placed, and most importantly, assigns a unique network identification number (ID) to each character. Upon receiving a character creation message from the server, the client uses the TLAPI architecture to create a character through Torchlight’s own functions. Once a character has been created the function will return a pointer to the character representation. The client then maps the network ID of the character to its pointer. This ID allows the client and server to communicate and ensure they are referring to the same character when issuing commands. The resulting character synchronization can be seen in Figure 4.4. The same technique is used for creating and synchronizing items such as swords, staves, potions, etc.

Figure 4.4: Multiplayer within the Town. The player’s character is shown in the center of the image and the second player’s character is shown on the left.

There is a caveat to this method of suppressing the creation of characters and items: the server has no knowledge of the client’s character and items. Therefore, the client must have an exception to allow the game to load its own character and any items they own. Since the server must be informed of this information to properly display the client character on its own display, the client sends this information to the server when it first joins the level. Figure 4.5 is a sequence diagram that displays some of the startup message flow between
Figure 4.5: Sequence diagram of the initial messages between the server and client game instances.
the server and client.

Since all the characters and items have a unique network ID attached to them, TLAPI can be used to register functions for handling character movement, item drops and pickups, equipping the item on the character, item identification, item destroying and retrieving gems, etc. Figures 4.6 and 4.7 display an identical game from two different player screens.

This process has worked very well with all characters tested, including those with a variety of items. A small amount of graphic latency is produced as new characters are loaded into the level, due to Torchlight loading the model data and textures from disk.

Figure 4.6: Item synchronization as seen by the server player. Three items have been dropped to the ground by the client character.

4.6.2 Chat Interface

Chat is a large part of any multiplayer game. It allows the users to talk via text messages while playing the game. The original Torchlight user interface was supplemented with a chat dialog system: a history window displaying old messages and an entry text field for entering new messages. Implementation of chat functionality was trivial and easily handled through the use of the CEGUI, protobuf and RakNet libraries. A window layout was created to display a history of chat for each player, as well as an editbox for entering text messages.
Figure 4.7: Item synchronization as seen by the client player. The same three items are shown to the other player, displaying item synchronization.

Once a player has finished entering text in the editbox they press the return key to send the message. The multiplayer functionality handles this event and simply grabs the text from the window, formats it into a specific format via protobuf and sends it to the server or clients. For receiving chat messages the process is simply reversed: the message is pulled from RakNet, parsed via protobuf and sent to the history window via CEGUI. All network messages, including chat messages, are polled at each game update by a patched function within Torchlight that performs an update for the game.

### 4.7 Dungeon Phase

The second phase builds upon the work done in the Town phase. The dungeon contains additional problems of synchronizing interactable objects, character attacks, health and other attribute changes, character deaths, and level changes.

Dungeons within Torchlight are the focused area of play in the game, they contain the action and replayability of the game. To support replayability, dungeon layouts are randomized using a seed value. Differing seed values in pseudorandom number systems create differing sequences of numbers. Using the same seed value each time will produce
the same sequence. Torchlight uses the seed value to generate a sequence of pseudorandom numbers which it in turn uses for determining the level layout. In certain cases this level layout is always static, e.g. for the town and for levels with scripted story driven cinematics or boss encounters.

Instead of reverse engineering the dungeon layout process and pushing the generated information to the client, it was sufficient to capture the seed value and send it to the client to clone the deterministic level generation procedure. The seed value exists in a relatively static memory location\(^1\) and thus can be directly written to and read from, without the use of any function. Synchronizing the seed values across the server and client forces the same dungeon layout to be created.

### 4.7.1 Level Transfers

The most difficult aspect in implementing the dungeon phase was the ability for characters to move between levels. The multiplayer design incorporates a process of having the server drive the entire simulation. A problem arises if a client joins and is not playing on the same level as the server’s character. The client does not have the ability to load characters, therefore no characters will be seen on the client’s level. Compounded onto this problem is the fact the game actually contains two separate dungeons: a dungeon for the storyline, and a near-infinite, randomly generated dungeon for replayability. The storyline dungeon is randomly laid out for levels without scripted NPC encounters; the environments of each level are also restricted depending upon how deep the player is within the levels.

### 4.7.2 Synchronizing Interactable Objects

Interactable objects exist within every dungeon level. An interactable object is considered to be anything the player can interact with in the level, e.g. a door or barrel. A trigger unit is a sub-class of an interactable object that handles attached logic, such as a lever that activates a bridge.

Trigger units are mapped to items within a level that are interactable. The player is

\(^1\)Relatively static means that it most likely changes each time the operating system loads the game, but it can be computed relative the the base address of where the operating system places the executable in memory.
allowed to click on them to trigger an event. The most common type of trigger unit is a closed door that can be clicked (via the mouse interface) so that it opens. Various other trigger units exist in the game to open other areas of interest, activate bridges via levers and act as traps.

Since trigger units are automatically created when an item is placed into the level, they are more difficult to synchronize. An approach to synchronizing trigger units would be to suppress the creation on the client and have the server create them. Simply creating a trigger unit was difficult because detailed knowledge of the dungeon layout would be needed, including knowledge of the interactable items in the layout.

The method that was the easiest to implement and that worked the best for synchronizing trigger units was based on whether the objects’ positions were the same. This works well since the dungeon layout is considered to be identical across all instances when the seed value is synchronized, and will always create the same items at the same positions. The function that is called when trigger units are activated by the player was obtained. Registering to this function, in addition to invoking it, allowed the multiplayer functionality to synchronize trigger unit actions. The trigger unit synchronization can be seen in Figure 4.8 and Figure 4.9, which show a door initially closed and then opened by the other player, respectively.

4.7.3 Attacking

Character attacks provide some difficulty for multiplayer synchronization. Attacks have a minimum and maximum range at which they can be applied to characters. Because the client and server are not run in lock-step to fully synchronize all instances of the game, small differences in character positions arise. These differences can affect when a player can attack another. A client attempting to kill an enemy by moving close to swing a sword may not have the server agree that it was close enough to hit the other character. The current solution is to check the character’s position on the client with that reported by the server. If the difference in positions is greater than a constant threshold value, the character on the client is forced to the position reported by the server. Forcefully moving a character to a position can cause visual jumps that the player may see.
4.7.4 Character Deaths

Every character has a health pool, whether it be a monster or a player. By registering (via TLAPI) a patched function for character health updates, health changes can be synchronized across the network. Since the server drives the simulation, any client character damage taken on the server’s instance is transferred, along with damage dealt to monsters. By explicitly calling the patched health update function it allows the client to see the monster and player deaths on their instance of the game when the character health reaches zero.

4.7.5 Character Resurrection

Player death is a normal part of the game and must be handled correctly. Torchlight offers players the ability to resurrect their character at their current location, at the beginning of the level, or in the town, for varying penalties. Resurrection is also synchronized and simply resets the player character’s health back to full. Currently, the multiplayer design does not factor in where the player wishes to resurrect and will simply place the character where they had died.
Figure 4.9: The same door shown as opened by the other player, displaying the synchronization of the trigger units.

4.7.6 Experience

As enemy characters are killed player characters gain experience points. Since the monster death is attributed to only one player, the experience gained must be shared across any characters connected. These experience points are distributed by the server when a monster is killed. Any attempts to add experience to a player on the client without server permission is suppressed.

4.7.7 Placement new

C++ allows dynamically allocated memory to be used from a pre-existing buffer, instead of the normal method of requesting it from the operating system. The C++ nomenclature for allocating from an existing buffer is called placement new.

Most of the Torchlight C++ classes are derived (directly or indirectly) from a common base class called CRunicCore. Any dynamic memory allocations that are done explicitly\(^2\) require that CRunicCore be inherited from Ogre::AllocatedObject as shown in Listing 4.2.

\(^2\)Explicit memory allocations include using new ourselves and not going through a factory function to create the object through regular Torchlight means.
This is required because the Ogre library uses its own memory manager and the C++ placement `new` [29]. Attempts at deleting this object when it is expected to be in existence in a pre-allocated buffer will crash the game. Therefore, TLAPI pulls in the Ogre headers and static library to build against to properly allocate any objects derived from CRunicCore onto Ogre’s pre-allocated memory buffer.

Placement `new` solved problems when attempting to create an object of type CItemGold, which displays gold on the ground. Gold is different from regular items because it is simply a currency value, not a physical item that can be placed into the inventory or dropped. CItemGold was created on the fly and not through a factory class, it was therefore required to be created the same by the multiplayer functionality indirectly using placement `new`.

```
struct CRunicCore : 
Ogre::AllocatedObject<Ogre::CategorisedAllocPolicy<Ogre::MEMCATEGORY_GENERAL>>
```

Listing 4.2: CRunicCore specification for using Ogre’s AllocatedObject to properly allocate memory with the placement `new` operator.

### 4.8 Debugging

The game can easily crash if a small bug is introduced. Bugs are more prominent when interacting with closed source applications because the programmer of the modification has little knowledge of the implementation details of the original application.

Bugs can be introduced directly within the multiplayer functionality, or indirectly by misusing Torchlight’s functionality. As an example, the client suppresses character creation in the level, however if the character creation function is patched and returns NULL the game will crash when attempting to add the character to the level. This is because the game does not check for a NULL pointer before dereferencing it, and in the real game the character would always be successfully created. The solution towards this was to allow the character creation, however the character would not be added to the level. Additionally, the character would be flagged to be destroyed and Torchlight would automatically clean it up the next game update tick.

In the case that a bug does occur, the Windows operating system will allow a just-in-time (JIT) debugger to be run. A JIT debugger has the ability to attach to the program
before the operating system kills the process, allowing valuable information to be obtained, most notably the last instruction executed that caused the exception. Microsoft Visual Studio was used as the JIT debugger in the case of Torchlight crashes where it was not immediately known what caused an exception. A screen capture of Visual Studio after Torchlight crashed and shown being debugged is shown in Figure 4.10. The development process for TLAPI and TLMP was to introduce small changes of functionality and then test the game. Debugging was made easier because the small changes introduced were usually the cause of the problem and could be more easily tracked down.

Figure 4.10: Microsoft Visual Studio running as the Just-in-Time debugger for Torchlight after a crash occurred. The disassembly of the instructions around the current instruction are shown, as well as memory of the stack.

For programs where one has the source code, the debugger will drop into the high-level language (C++) source at the point where an exception occurs. Variable names, source code, the call stack, memory and variable values are all easily available to the person debugging the program. Since Torchlight contains no supplied debugging symbols or source code, the debugger must drop into the low-level assembly language and only has the ability to display assembly instructions, the last executed instruction, register and memory values, and a limited call stack.
With knowledge of where code exists in memory it is also possible to narrow down bugs to the module they occurred in. In terms of a Windows process, modules are executable libraries such as the program executable itself, or DLLs that it uses and are loaded at runtime. Because Windows has knowledge of where these modules exist in memory, it is possible to examine the last instruction’s memory address and cross-reference it with the known loaded modules, which in fact is done automatically by the debugger.

A JIT debugger is a useful tool when source code is unavailable and can give valuable information on what caused the application to crash.

4.9 Overview

An overview of the design and development process for Torchlight Multiplayer was discussed. A majority of problems arose with synchronizing across game instances. The problems and solutions in each case of synchronizing different game entities were described. Finally, a debugging method for the occasional crash within Torchlight was also shown.
Chapter 5   Results

This chapter covers the results of the multiplayer implementation. Specifically, the playability in both phases of the multiplayer application and the overall playability of the game. Difficulties are also presented as well as existing problems that could be fixed in future work.

5.1 Town Phase

The target functionality of the Town phase works very well. Players have the ability to create and join games and interact with other players. Interaction is done via game mechanics such as moving the character, or dropping and picking up items. Social interaction is available through a supplied chat interface that allows players to communicate with one another. Player character pets are also supported within the game, as well as any other attached minion characters to the player. NPC inventories and player inventories are synchronized such that any items purchased or sold will propagate to all the connected clients. It is therefore possible to sell items to a NPC vendor and have these items be purchased through the vendor by another player.

The Town phase was a success with its overall purpose. It tested what could be accomplished and offered the ground work for basic, working functionality.

5.2 Dungeon Phase

The main objective of having players working together to fight monsters is working. The current version of multiplayer allows client characters to move between dungeon levels with the host and help fight monsters. Trigger units have been successful: as characters open doors, levers and use other interactable items, the action is propagated to all connected game clients. Breakable items such as barrels are also synchronized. Item creation and drops from monsters are synchronized properly. Monster creation, positioning and deaths also work. The player resurrection option for resurrecting in the town is removed due to level transfer issues between the game instances described in Section 4.7.1.

Because of the server-centric design of controlling most aspects of the simulation for the
client, it is currently not possible for the client character to separate away from the host’s current level. For multiplayer this is not a large issue since players will be cooperating with one another and will be on the same level anyways.

Some artificial intelligence (AI) problems are exhibited when a client character moves away from the server character. The AI within the game is very simple, and uses a proximity check to determine if the monster should engage the player. Because the server game controls all monster movement, and the fact that the server processes the AI which only acts upon the server player’s position, monsters will not attack client characters as they become close. A solution to this problem may require that Torchlight’s proximity function be patched to check distances for client characters too.

The attack mechanism for client characters is also not working fully. At times the client will attack a monster correctly, at other times they will continually miss, and in some cases perform enormously large attacks that instantly kill monsters. It seems that these attack problems are a combination of several factors: the client character’s position is somewhat misaligned on the server’s game instance due to the latency of the network, and the underlying attacking mechanism is relying on data for the client character that is not properly set up. Multiplayer games using a client-server architecture will not update character positions on the client until the server has told it to do so. This usually incurs an overhead latency since actions on the client must be sent to the server and propagated back to all clients if valid [30]. Aside from character movement Torchlight Multiplayer uses this method; when the client does an action it is first sent as a request to the server. This round-trip time creates a latency between when the client wants to move their character and when the character responds and moves. Aside from the latency the design change would solve the misalignment of the characters in separate game instances.

The dungeon phase has game saves turned off. This is to remove any problems encountered in any multiplayer games from affecting the saved game in single-player mode. Currently, other player characters and minions are removed from each game’s instance before a saved game is created. Additionally, the game only saves on a level transfer. TLMP will remove all character data between transfers anyway, so it is safe to remove the other player characters before the game is saved. Character data is repopulated when a level
transfer is completed.

The current difficulties in the dungeon phase are synchronizing the skills of characters across the parallel game instances. The character animation for when such skills are used correctly, however the visual effects are not set up properly. Level transfers on the client are currently not working fully if the client starts on a different level than the server. Forcing the client’s character to the town when loading and then moving it to the server character’s level was one solution for synchronizing the character’s positions. This solution is not fully working in all cases and will occasionally have the client continually load between two levels. This is a problem with how the game was designed with two separate dungeons (story-line or endless dungeon), and problems occur if the server character is one of the dungeons and the client is in the other.

Cinematic events display lingering characters on the client that should otherwise be hidden to the player. This is due to the visibility flag of the character not being properly transferred. The flag is not transferred because the server will update it on each game update, initially setting the character to visible and then determining whether it should be hidden. The amount of network traffic involved would be large for each game update. In the future a solution towards discovering this flag every few seconds and propagating it to the client may work.

Occasional crashes do occur within the game. If the game does crash it usually occurs on level transfers. Level transfers clean up player characters that were created for the multiplayer functionality and repopulate the game with them as the level is loaded. A lot of information is passed around and must synchronize correctly when Torchlight is ready to use such information. This process is likely the cause of crashes.

Some user interface problems do occur, both with the original Torchlight and the added UI functionality. Torchlight’s UI will remove any windows in memory if the game window is resized. To handle repopulating the UI with the window information, the resize function is registered and when it is called the UI will load the layout files. This process will effectively reset the UI and the user will have to navigate to where they were previously.
5.3 Overall Playability

The overall playability of the multiplayer works well as a prototype system. Major portions of required multiplayer functionality have been implemented and tested. Character and item synchronization work, which are the common elements of any ARPG. Because of this basic synchronization, players are able to enter the game world with their characters. Once a player is in a game they can talk, trade items and co-operatively play the dungeon crawler aspect of the game.

5.4 Future Work

There is still much to be done regarding the multiplayer extension. One aspect of the project would be to test this with a large number of players in the lobby and within a game. However, bug fixes would be required before a large player base could use the system.

The complexities of patching and using code at run-time may cause future updates from the creators of Torchlight to break the multiplayer extension. Because the functions rely on specific memory addresses, and these can change once an update is applied, functions that use an incorrect address will crash the application. Although the work involved in updating these offsets is very little, it is a time consuming process that requires a human to update the offsets. An automatic tool could be developed to compare the old and new executable and determine where the new offsets exist.

Addressing the issues presented in the Dungeon phase would also be of benefit towards the players of the game. Given more time, adding some polish to the multiplayer functionality would also be of benefit.
Chapter 6  Conclusion

Much of the main functionality for supporting a multiplayer experience has been reverse engineered and used from the existing code of Torchlight. The tools, reverse engineering process and design of the multiplayer extension have been described.

TLAPI and TLMP are freely available to download and develop on as they are open-source [31][32]. Since the design of the split functionality between TLAPI and TLMP was used, it is possible to apply TLAPI for other uses, such as post-processing graphic effects, adding more in-game settings, etc. The open-source nature also allows other developers to contribute new design ideas for improving either TLAPI or TLMP if they choose. Additionally, it allows others to view the code and use it for other applications outside of Torchlight. For instance, when a new game or application is produced the TLAPI code and design would give a tremendous jump start on implementing extended functionality.

This thesis detailed the process of reverse engineering to extend the functionality of a closed-source application. The process details the use of static and dynamic analysis tools to determine the location and logic of functions. A Torchlight API was built to interface with the game and allow programmers access to game elements at run-time. Multiplayer functionality was then described, along with how it was used with TLAPI to allow players to share a common game experience. Problems were encountered, many had solutions that are presented here. Some problems remain within the project, which were reported and discussed as future work.

Many of the techniques described can be applied to current and future applications if additional functionality is required. It has been shown that adding functionality to a closed-source application is possible to do and can extend the life of an application and replayability of a game.
Bibliography


