ARCHITECTING NETWORKED GAMES

"For any aspiring game programmer, this book is a must read! Glazer and Madhav are some of the best at explaining these critical multiplayer concepts. I look forward to their next book!"

—**ZACH METCALF,** Game Programmer at Rockstar Games and USC Games Alum

MULTIPLAYER GAME Programming

Joshua **GLAZER**
Sanjay **MADHAV**

Multiplayer Game Programming

The Addison-Wesley Game Design and Development Series

Visit **informit.com/series/gamedesign** for a complete list of available publications.

Essential References for Game Designers and Developers

These practical guides, written by distinguished professors and industry gurus, cover basic tenets of game design and development using a straightforward, common-sense approach. The books encourage readers to try things on their own and think for themselves, making it easier for anyone to learn how to design and develop digital games for both computers and mobile devices.

Make sure to connect with us!
informit.com/socialconnect

Multiplayer Game Programming

Architecting Networked Games

Joshua Glazer
Sanjay Madhav

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Control Number: 2015950053

ISBN-13: 978-013-403430-0
ISBN-10: 0-134-03430-9

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing: November 2015

Editor-in-Chief
Mark Taub

Acquisitions Editor
Laura Lewin

Development Editor
Michael Thurston

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Cenveo® Publisher Services

Indexer
Cenveo® Publisher Services

Proofreader
Cenveo® Publisher Services

Technical Reader
Alexander Boczar
Jeff Tucker
Jonathan Rucker

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Cenveo® Publisher Services

To Grilled Cilantro and the Jellybean. You know who you are.

–Joshua Glazer

To my family for their support, and to all of my TAs over the years.

–Sanjay Madhav

This page intentionally left blank

Contents

This page intentionally left blank

PREFACE

Networked multiplayer games are a huge part of the games industry today. The number of players and amount of money involved are staggering. As of 2014, *League of Legends* boasts 67 million active players each month. The 2015 *DoTA 2* world championship has a prize pool of over $16 million at the time of writing. The *Call of Duty* series, popular in part due to the multiplayer mode, regularly has new releases break $1 billion in sales within the first few days of release. Even games that have historically been single-player only, such as the *Grand Theft Auto* series, now include networked multiplayer components.

This book takes an in-depth look at all the major concepts necessary to program a networked multiplayer game. The book starts by covering the basics of networking—how the Internet works and how to send data to other computers. Once the fundamentals are established, the book discusses the basics of transmitting data for games—how to prepare game data to be sent over the network, how to update game objects over the network, and how to organize the computers involved in the game. The book next discusses how to compensate for unreliability and lag on the Internet, and how to design game code to scale and be secure. Chapters 12 and 13 cover integrating gamer services into and using cloud hosting for dedicated servers—two topics that are extremely important for networked games today.

This book takes a very practical approach. Most chapters not only discuss the concepts, they walk you through the actual code necessary to get your networked game working. The full source code for two different games is provided on the companion website—one game is an action game and the other is a real-time strategy (RTS). To help with the progression of topics, multiple versions of these two games are presented throughout the course of this book.

Much of the content in this book is based on curriculum developed for a multiplayer-game programming course at the University of Southern California. As such, it contains a proven method for learning how to develop multiplayer games. That being said, this book is not written solely for those in an academic setting. The approach taken by this book is just as valuable to any game programmer interested in learning how to engineer for a networked game.

Who Should Read This Book?

While Appendix A covers some aspects of modern C++ used in this book, it is assumed that the reader already is comfortable with C++. It is further assumed that the reader is familiar with

the standard data structures typically covered in a CS2 course. If you are unfamiliar with C++ or want to brush up on data structures, an excellent book to refer to is *Programming Abstractions in C++* by Eric Roberts.

It is further assumed that the reader already knows how to program single-player games. The reader should ideally be familiar with game loops, game object models, vector math, and basic game physics. If you are unfamiliar with these concepts, you will want to first start with an introductory game programming book such as *Game Programming Algorithms and Techniques* by Sanjay Madhav.

As previously mentioned, this book should be equally effective either in an academic environment or for game programmers who simply want to learn about networked games. Even game programmers in the industry who have not previously made networked games should find a host of useful information in this book.

Conventions Used in This Book

Code is always written in a fixed-point font. Small code snippets may be presented either `inline` or in standalone paragraphs:

```
std::cout << "Hello, world!" << std::endl;
```

Longer code segments are presented in code listings, as in Listing 0.1.

Listing 0.1 Sample Code Listing

```
// Hello world program!
int main()
{
   std::cout << "Hello, world!" << std::endl;
   return 0;
}
```

For readability, code samples are color coded much like in an IDE.

Throughout this book, you will see some paragraphs marked as notes, tips, sidebars, and warnings. Samples of each are provided for the remainder of this section.

> **note**
>
> Notes contain useful information that is separate from the flow of the normal text of the section. Notes should almost always be read.

tip

Tips are used to provide helpful hints when implementing specific systems in your game's code.

warning

Warnings are very important to read, as they contain common pitfalls or issues to watch out for, and ways to solve or work around these issues.

SIDEBAR

Sidebars contain lengthier discussions that usually are tangential to the main content of the chapter. These can provide some interesting insight to a variety of issues, but contain content that is deemed nonessential to the pedagogical goals of the chapter.

Why C++?

The vast majority of this book uses C++ because it is still the de facto language used in the game industry by game engine programmers. Although some engines allow a great deal of code for a game to be written in other languages, such as Unity in C#, it is important to remember that most of the lower-level code for these engines is still written in C++. Since this book is focused on writing a networked multiplayer game from the ground up, it makes the most sense to do so in the language that most game engines are written in. That being said, even if you are writing all your game's networking code in another language, all the core concepts will still largely be the same. Still, it is recommended that you be familiar with C++, otherwise the code samples may not make much sense.

Why JavaScript?

Since starting off life as a hastily hacked together scripting language to support the Netscape browser, JavaScript has evolved into a standardized, full-featured, somewhat functional language. Its popularity as a client-side language helped it make the leap to server side, where its first-class procedures, simple closure syntax, and dynamically typed nature make it very efficient for the rapid development of event-driven services. It's a little hard to refactor and it provides worse performance than C++, making it a bad choice for next-generation front-end development.

That's not an issue on the backend, where scaling up a service can mean nothing more than dragging a slider to the right. The backend examples in Chapter 13 use JavaScript, and understanding them will require a decent knowledge of the language. As of this writing, JavaScript is currently the number one most active language on GitHub by a margin of almost 50%. Following trends for the sake of trends is rarely a good idea, but being able to program in the world's most popular language definitely has its benefits.

Companion Website

The companion website for this book is at https://github.com/MultiplayerBook. The website has a link to the sample code used throughout the book. It also contains the errata, as well as links to PowerPoint slides and a sample syllabus for use in an academic setting.

ACKNOWLEDGMENTS

We would like to thank the entire team at Pearson for guiding this book through completion. This includes our Executive Editor, Laura Lewin, who convinced us to get the band together and write this book. Olivia Basegio, our Assistant Editor, has been great to ensure the process goes along smoothly. Michael Thurston, our Development Editor, has provided insight to help us improve the content. We would also like to thank the entire production team, including Andy Beaster, our production editor and Cenveo® Publisher Services.

Our technical reviewers, Alexander Boczar, Jonathan Rucker, and Jeff Tucker, were instrumental in ensuring the accuracy of this book. We would like to thank them for taking time out of their busy schedule to review the chapters. Finally, we'd like to thank Valve Software for allowing us to write about the Steamworks SDK, as well as reviewing Chapter 12.

Acknowledgments from Joshua Glazer

Thank you so much Lori and McKinney for your infinite understanding, support, love, and smiles. You are the best family ever. I lost a lot of time with you guys while writing this book, but hey, I'm done now! Wooh! Thank you mom and dad for raising and loving me and making sure I could write English at least within two orders of magnitude of how well I write code. Thank you Beth for the innumerable amazing things you've done for the world and also for watching my cats sometimes. Thank you all my extended family for the support and belief and for sounding impressed that I'm writing a textbook. Thank you Charles and all my Naked Sky Pros (short for programmers) for keeping me on my toes and pointing out whenever I'm being really daft. Thank you Tian and Sam for dragging me into this ludicrous industry. Thank you Sensei Copping for teaching me that the man who cleans his house by dirtying his closet has destroyed himself. And, of course, thank you Sanjay for bringing me on board at USC and tackling this mega undertaking with me! I never could have done this without your wisdom and cool-headedness, not to mention you writing half the stuff! (Oh yeah, and thank you to Lori again, just in case you missed the first one!)

Acknowledgments from Sanjay Madhav

There is a correlation between the number of books an author has written and the length of their acknowledgements. Since I wrote a lot of acknowledgements in my last book, I'll keep it short this time. I'd of course like to thank my parents and my sister. I'd also like to thank my colleagues in the Information Technology Program at USC. Finally, I'd like to thank Josh for agreeing to teach our "Multiplayer Game Programming" course, because this book would not have happened were it not for that course.

ABOUT THE AUTHORS

Joshua Glazer is a cofounder and CTO of Naked Sky Entertainment, the independent development studio behind console and PC games such as *RoboBlitz*, *MicroBot*, *Twister Mania*, and more recently, the mobile hits *Max Axe* and *Scrap Force*. As a leader of the Naked Sky team, he has consulted on several external projects including Epic Games' Unreal Engine, Riot Games' *League of Legends*, THQ's *Destroy All Humans* franchise, and numerous other projects for Electronic Arts, Midway, Microsoft, and Paramount Pictures.

Joshua is also a part-time lecturer at the University of Southern California, where he has enjoyed teaching courses in multiplayer game programming and game engine development.

Sanjay Madhav is a senior lecturer at the University of Southern California, where he teaches several programming and video game programming courses. His flagship course is an undergraduate-level game programming course that he has taught since 2008, but he has taught several other course topics, including game engines, data structures, and compiler development. He is also the author of *Game Programming Algorithms and Techniques*.

Prior to joining USC, Sanjay worked as a programmer at several video game developers, including Electronic Arts, Neversoft, and Pandemic Studios. His credited games include *Medal of Honor: Pacific Assault*, *Tony Hawk's Project 8*, *Lord of the Rings: Conquest*, and *The Saboteur*—most of which had networked multiplayer in one form or another.

CHAPTER 1

OVERVIEW OF NETWORKED GAMES

Although there are notable exceptions, the concept of networked multiplayer games didn't really catch on with mainstream gamers until the 1990s. This chapter first gives a brief history of how multiplayer games evolved from the early networked games of the 1970s to the massive industry today. Next, the chapter provides an overview of the architecture of two popular network games from the 1990s—*Starsiege: Tribes* and *Age of Empires*. Many of the techniques used in these games are still in use today, so this discussion gives insight into the overall challenges of engineering a networked multiplayer game.

A Brief History of Multiplayer Games

The progenitor of the modern networked multiplayer game began on university mainframe systems in the 1970s. However, this type of game didn't explode until Internet access became common in the mid-to-late 1990s. This section gives a brief overview of how networked games first started out, and the many ways these types of games have evolved in the nearly half century since the first such games.

Local Multiplayer Games

Some of the earliest video games featured **local multiplayer**, meaning they were designed for two or more players to play the game on a single computer. This included some very early games such as including *Tennis for Two* (1958) and *Spacewar!* (1962). For the most part, local multiplayer games can be programmed in the same manner as single-player games. The only differences typically are multiple viewpoints and/or supporting multiple input devices. Since programming local multiplayer games is so similar to single-player games, this book does not spend any time on them.

Early Networked Multiplayer Games

The first **networked multiplayer games** were run on small networks composed of mainframe computers. What distinguishes a networked multiplayer game from a local multiplayer game is that networked games have two or more computers connected to each other during an active game session. One such early mainframe network was the PLATO system, which was developed at the University of Illinois. It was on the PLATO system that one of the first networked games, the turn-based strategy game *Empire* (1973), was created. Around the same time as *Empire*, the first-person networked game *Maze War* was created, and there is not a clear consensus as to which of these two games was created first.

As personal computers started to gain some adoption in the latter part of the 1970s, developers figured out ways to have two computers communicate with each other over serial ports. A **serial port** allows for data to be transmitted one bit at a time, and its typical purpose was to communicate with external devices such as printers or modems. However, it was also possible to connect two computers to each other and have them communicate via this connection. This made it possible to create a game session that persisted over multiple personal computers, and led to some of the earliest networked PC games. The December 1980 issue of *BYTE Magazine* featured an article on how to program so-called Multimachine Games in BASIC (Wasserman and Stryker 1980).

One big drawback of using serial ports was that computers typically did not have more than two serial ports (unless an expansion card was used). This meant that in order to connect more than two computers via serial port, a **daisy chain** scheme where multiple computers are connected to each other in a ring had to be used. This could be considered a type of network topology, a topic that is covered in far more detail in Chapter 6, "Network Topologies and Sample Games."

So in spite of the technology being available in the early 1980s, most games released during the decade did not really take advantage of local networking in this manner. It wasn't until the 1990s that the idea of locally connecting several computers to play a game really gained traction, as discussed later in this chapter.

Multi-User Dungeons

A **multi-user dungeon** or MUD is a (usually text-based) style of multiplayer game where several players are connected to the same virtual world at once. This type of game first gained popularity on mainframes at major universities, and the term originates from the game *MUD* (1978), which was created by Rob Trushaw at Essex University. In some ways, MUDs can be thought of as an early computer version of the role-playing game *Dungeons and Dragons*, though not all MUDs are necessarily role-playing games.

Once personal computers became more powerful, hardware manufacturers began to offer modems that allowed two computers to communicate with each other over standard phone lines. Although the transmission rates were extraordinarily slow by modern standards, this allowed for MUDs to be played outside the university setting. Some ran MUD games on a **bulletin board system** (BBS), which allowed for multiple users to connect via modem to a system that could run many things including games.

Local Area Network Games

A **local area network** or LAN is a term used to describe several computers connected to each other within a relatively small area. The mechanism used for the local connection can vary—for example, the serial port connections discussed earlier in this chapter would be one example of a local area network. However, local area networks really took off with the proliferation of Ethernet (a protocol which is discussed in more detail in Chapter 2, "The Internet").

While by no means the first game to support LAN multiplayer, *Doom* (1993) was in many ways the progenitor of the modern networked game. The initial version of the id Software first-person shooter supported up to four players in a single game session, with the option to play cooperatively or in a competitive "deathmatch." Since *Doom* was a fast-paced action game, it required implementation of several of the key concepts covered in this book. Of course, these techniques have evolved a great deal since 1993, but the influence of *Doom* is widely accepted. For much greater detail on the history and creation of *Doom*, read *Masters of Doom* (2003), listed in the references at the conclusion of this chapter.

Many games that support networked multiplayer over a LAN also supported networked multiplayer in other ways—whether by modem connection or an online network. For many years, the vast majority of networked games also supported gaming on a LAN. This led to the rise of LAN parties where people would meet at a location and connect their computers to play networked games. Although some networked multiplayer games are still released with LAN play, the trend in recent years seems to have developers forgoing LAN play for exclusively online multiplayer.

Online Games

In an **online game**, players connect to each other over some large network with geographically distant computers. Today, online gaming is synonymous with Internet gaming, but the term "online" is a bit broader and can include some of the earlier networks such as CompuServe that, originally, did not connect to the Internet.

As the Internet started to explode in the late 1990s, online games took off alongside it. Some of the popular games in the earlier years included id Software's *Quake* (1996) and Epic Game's *Unreal* (1998).

Although it may seem like an online game could be implemented in much the same way as a LAN game, a major consideration is **latency**, or the amount of time it takes data to travel over the network. In fact, the initial version of *Quake* wasn't really designed to work over an Internet connection, and it wasn't until the *QuakeWorld* patch that the game was reliably playable over the Internet. Methods to compensate for latency are covered in much greater detail in Chapter 7, "Latency, Jitter, and Reliability" and Chapter 8, "Improved Latency Handling."

Online games took off on consoles with the creation of services such as Xbox Live and PlayStation Network in the 2000s, services that were direct descendants of PC-based services such as GameSpy and DWANGO. These console services now regularly have several million active users during peak hours (though with expansion of video streaming and other services to consoles, not all of these active users may be playing a game). Chapter 12, "Gamer Services," discusses how to integrate one such gamer service—Steam—into a PC game.

Massively Multiplayer Online Games

Even today, most online multiplayer games are limited to a small number of players per game session—somewhere from 4 to 32 is commonly the number of supported players. In a **Massively Multiplayer Online Game** (MMO), however, hundreds if not thousands of players can participate in a single game session. Most MMO games are role-playing games and thus called **MMORPGs**. However, there are certainly other styles of MMO games such as first-person shooters (MMOFPS).

In many ways, MMORPGs can be thought of as the graphical evolution of multi-user dungeons. Some of the earliest MMORPGs actually predated the widespread adoption of the Internet, and instead functioned over dial-in networks such as Quantum Link (later America Online) and CompuServe. One of the first such games was *Habitat* (1986) which implemented several pieces of novel technology (Morningstar and Farmer 1991). However, it wasn't until the Internet became more widely adopted that the genre gained more traction. One of the first big hits was *Ultima Online* (1997).

Other MMORPGs such as *EverQuest* (1999) were also successful, but the genre took the world by storm with the release of *World of Warcraft* (2004). At one point, Blizzard's MMORPG had over

12 million active subscribers worldwide, and the game became such a large part of popular culture that it was featured in a 2006 episode of the animated series *South Park*.

Architecting an MMO is a complex technical challenge, and some of these challenges are discussed in Chapter 9, "Scalability." However, most of the techniques necessary to create an MMO are well beyond the scope of this book. That being said, the foundations of creating a smaller-scale networked game are important to understand before it's possible to even consider creating an MMO.

Mobile Networked Games

As gaming has expanded to the mobile landscape, multiplayer games have followed right along. Many multiplayer games on these platforms are **asynchronous**—typically turn-based games that do not require real-time transmission of data. In this model, players are notified when it is their turn, and have a large amount of time to make their move. The asynchronous model has existed from the very beginning of networked multiplayer games. Some BBS only had one incoming phone line connection, which meant that only one user could be connected at any one time. Thus, a player would connect, take their turn, and disconnect. Then at some point in the future, another player would connect and be able to respond and take their own turn.

An example of a mobile game that uses asynchronous multiplayer is *Words with Friends* (2009). From a technical standpoint, an asynchronous networked game is simpler to implement than a real-time one. This is especially true on mobile platforms, because the platform APIs (application program interfaces) have built-in functionality for asynchronous communication. Originally, using an asynchronous model for mobile games was somewhat out of necessity because the reliability of mobile networks is comparatively poor to wired connections. However, with the proliferation of Wi-Fi–capable devices and improvements to mobile networks, more and more real-time networked games are appearing on these devices. An example of a mobile game that takes advantage of real-time network communication is *Hearthstone: Heroes of Warcraft* (2014).

Starsiege: Tribes

Starsiege: Tribes is a sci-fi first-person shooter that was released at the end of 1998. At the time of release, it was well regarded as a game featuring both fast-paced combat and a comparatively massive number of players. Some game modes supported 128 players over either a LAN or the Internet. To gain some perspective on the magnitude of the challenge in implementing such a game, keep in mind that during this time period, the vast majority of players with an Internet connection used a dial-up service. At best, these dial-up users had a modem capable of speeds up to 56.6 kbps. In the case of *Tribes*, it actually supported users with modem speeds of only 28.8 kbps. By modern standards, these are extremely slow connection speeds. Another factor was that dial-up connections also had relatively high latency—a latency of several hundred milliseconds was rather common.

It may seem that a networking model designed for a game with low bandwidth constraints would be irrelevant in the modern day. However, it turns out that the model used in *Tribes* still has a great deal of validity even today. This section summarizes the original *Tribes* networking model—for a more in-depth discussion, refer to the article by Frohnmayer and Gift referenced at the end of this chapter.

Do not be concerned if some of the concepts covered in this section don't entirely make sense right now. The intent is that by looking at a networked multiplayer game's architecture at a high level, you will gain an appreciation for the numerous technical challenges faced and decisions to be made. All the topics touched on in this section are covered in much greater detail throughout the remainder of this book. Furthermore, one of the sample games built throughout this book, *RoboCat Action*, ultimately uses a model similar to the *Tribes* networking model.

One of the first choices made when engineering a networked game is to choose a **communications protocol**, or an established convention by which data is exchanged between two computers. Chapter 2, "The Internet," covers how the Internet works and the commonly used protocols. Chapter 3, "Berkeley Sockets," covers a ubiquitous library used to facilitate communication via these protocols. For the sake of the current discussion, the only thing you need to know is that, for efficiency reasons, *Tribes* uses an *unreliable* protocol. This means that data sent over the network is *not* guaranteed to be received by the destination.

However, using an unreliable protocol can be problematic when a game needs to send information that is important to all the players in the game. Thus, the engineers needed to consider the different types data they wanted to send out. The developers of *Tribes* ultimately separated their data requirements into the following four categories:

1. **Non-guaranteed data.** As one might expect, this is data that the game designates as nonessential to the game. When bandwidth-starved, the game can choose to drop this data first.
2. **Guaranteed data.** This data guarantees both arrival and ordering of the data in question. This is used for data deemed critical by the game, such as an event signifying when a player has fired a weapon.
3. **"Most recent state" data.** This type of data is for cases where only the most recent version of the data is of importance. One example is the hit points of a particular player. A player's hit points 5 seconds ago are not terribly relevant if the game knows what their hit points are right now.
4. **Guaranteed quickest data.** This data is given the highest priority in order to transmit as quickly as possible *with* guaranteed delivery. An example of this type of data is player movement information, which is typically relevant for a very short period of time, and thus should be transmitted quickly.

Many of the implementation decisions made in the *Tribes* Networking Model center on providing these four types of data transmission.

Another important design decision was to utilize a client-server model instead of a peer-to-peer model. In a **client-server model**, players all connect to a central server, whereas in a **peer-to-peer model**, every player connects to every other player. As discussed in Chapter 6, "Network Topologies and Sample Games," a peer-to-peer model requires $O(n^2)$ bandwidth. This means that the bandwidth grows at a quadratic rate based on the number of users. In this case, with n being as high as 128, using peer-to-peer would lead to very little bandwidth per player. To avoid this issue, *Tribes* instead implemented a client-server model. In this configuration, the bandwidth requirements of each player remain constant, while the server must handle only $O(n)$ bandwidth. However, this meant that the server needed to be on a network that would allow for several incoming connections—the type of connection that only a company or university might have owned at the time.

Next, *Tribes* split up their networking implementation into several different layers—one can think of this as a "layer cake" of the *Tribes* Networking Model. This is illustrated in Figure 1.1. The remainder of this section briefly describes the composition of each of these layers.

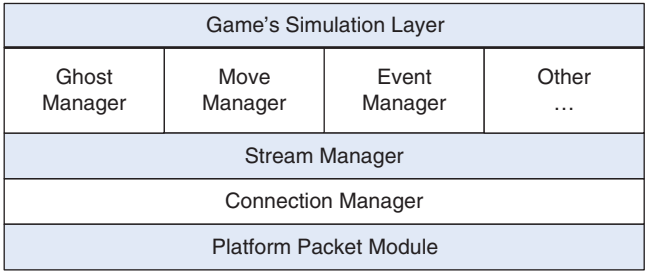

Figure 1.1 The main components of the *Tribes* Networking Model

Platform Packet Module

A **packet** is a formatted set of data sent over a network. In the *Tribes* model, the **platform packet module** is the lowest layer. It is the only layer in the model that is platform-specific. In essence, this layer is a wrapper for the standard socket APIs that can construct and send various packet formats. The implementation of this layer might look rather similar to the systems implemented in Chapter 3, "Berkeley Sockets."

Since *Tribes* utilized an unreliable protocol, the developers needed to add some mechanism to handle the data they decided needed to be guaranteed. Similar to the approach discussed in Chapter 7, "Latency, Jitter, and Reliability," *Tribes* implemented a custom reliability layer. However, this reliability layer is not handled by the platform packet module; instead the higher level managers such as the ghost manager, move manager, or event manager are responsible for adding any reliability.

Connection Manager

The job of the **connection manager** is to abstract the connection between two computers over the network. It receives data from the layer above it, the stream manager, and transmits data to the layer below it, the platform packet module.

The connection manager level is still unreliable. It *does not* guarantee delivery of data sent to it. However, the connection manager *does* guarantee a **delivery status notification**—that is to say, the status of a request passed to the connection manager can be verified. In this way, it is possible for the level above the connection manager (the stream manager) to know whether or not particular data was successfully delivered.

The delivery status notification is implemented with a sliding window bit field of acknowledgments. Although the original *Tribes* Networking Model paper does not contain a detailed discussion regarding the implementation of the connection manager, an implementation of a similar system is discussed in Chapter 7, "Latency, Jitter, and Reliability."

Stream Manager

The primary job of the **stream manager** is to send data to the connection manager. One important aspect of this is determining the maximum rate of data transmission that is allowed. This will vary depending on the quality of the Internet connection. An example given in the original paper is where a user on a 28.8-kbps modem might have their packet rate set to 10 packets per second with a maximum size of 200 bytes per packet, for approximately 2 kB of data per second. This rate and size is sent to the server upon connection of the client, in order to ensure that the server does not overwhelm the client's connection with too much data.

Since several other systems will ask the stream manager to send data, it is also the duty of the stream manager to prioritize these requests. The move, event, and ghost managers are given the highest priority when in a bandwidth-bound scenario. Once the stream manager decides on what data to send, the packets are dispatched to the connection manager. In turn, the higher-level managers will be informed by the stream manager regarding the status of delivery.

Because of the set interval and packet size enforced by the stream manager, it is very much possible for a packet to be dispatched with multiple types of data in it. For example, a packet may have some data from the move manager, some data from the event manager, and some data from the ghost manager.

Event Manager

The **event manager** maintains a queue of events that are generated by the game's simulation. These events can be thought of as a simple form of a **remote procedure call** or **RPC**, a function that can be executed on a remote machine. RPCs are discussed in Chapter 5, "Object Replication."

For example, when a player fires a weapon, this would likely cause a "player fired" event to be sent to the event manager. This event can then be sent to the server, which will actually validate and execute the weapon firing. It is also the purview of the event manager to prioritize the events—it will try to write as many of the highest priority events as possible until any of the following conditions are true: the packet is full, the event queue is empty, or there are currently too many active events.

The event manager also tracks the transmission records for each event marked as reliable. In this way, it is very simple for the event manager to enforce reliability. If a reliable event is unacknowledged, then the event manager can simply prepend the event to the event queue and try again. Of course, there will be some events that are marked as unreliable. For these unreliable events, there is no need to even track their transmission records.

Ghost Manager

The **ghost manager** is perhaps the most important system in terms of supporting up to 128 players. At a high level, the job of the ghost manager is to **replicate** or "ghost" *dynamic* objects that are deemed relevant to a particular client. In other words, the server sends information about dynamic objects to the clients, but only the objects that the server thinks the client needs to know about. The game's simulation layer is responsible for determining what a client absolutely *needs* to know and what a client ideally *should* know. This adds an inherent prioritization to game objects in the world: "need to know" objects are the highest priority, while "should know" objects are lower priority. In order to determine whether or not an object is relevant to a particular client, there are several different approaches that can be employed. Chapter 9, "Scalability," covers some of these approaches. In general, determining object relevancy is very game-specific.

Regardless of how the set of relevant objects is computed, the job of the ghost manager is to transmit object state from server to client for as many relevant objects as possible. It's very important that the ghost manager guarantees that the most recent data is always successfully transmitted to all of the clients. The reason for this is that the game object information that is ghosted will often contain information such as health, weapons, ammo count, and so on—all cases where the most recent data is the only information that matters.

When an object becomes **relevant** (or "in scope"), the ghost manager will assign some information to the object, which is appropriately called a **ghost record**. This record will include items such as a unique ID, a state mask, the priority, and status change (whether or not the object has been marked as in or out of scope).

For transmission of the ghost records, the objects are prioritized first by status change and then by the priority level. Once the ghost manager determines the objects that should be sent, their data can be added to the outgoing packet using an approach similar to what is covered in Chapter 5, "Object Replication."

Move Manager

The responsibility of the **move manager** is to transmit player movement data as quickly as possible. If you've ever played a fast-paced multiplayer game, you are likely cognizant of the fact that accurate movement information is extremely important. If the information regarding a player's position is slow to arrive, it could result in players shooting at where a player used to be instead of where a player is, which can result in frustrating gameplay. Quick movement updates can be an important way to reduce the perception of latency on the part of player.

The other reason the move manager is assigned a high priority is because input data is captured at 30 FPS. This means there is new input information available 30 times per second, so the latest data is sent as quickly as possible. This higher priority also means that, when move data is available, the stream manager will always first add any pending move manager data to an outgoing packet. Each client is responsible for transmitting their move information to the server. The server then applies this move information in its simulation of the game, and acknowledges the receipt of the move information to the client who sent it.

Other Systems

There are a few other systems in the *Tribes* model, though these are less critical to the overarching design. For example, there is a datablock manager, which handles transmission of game objects that are relatively static in nature. This differs from the relatively dynamic objects that are handled by the ghost manager. An example for this might be a static vehicle such as a turret—the object doesn't really move, but it exists to serve a purpose when a player interacts with it.

Age of Empires

As with *Tribes*, the real-time strategy (RTS) game *Age of Empires* was released in the late 1990s. This means that *Age of Empires* faced many of the same bandwidth and latency constraints of dial-up Internet access. *Age of Empires* uses a **deterministic lockstep** networking model. In this model, all the computers are connected to each other, meaning it is peer-to-peer. A guaranteed *deterministic* simulation of the game is concurrently performed by each of the peers. It is *lockstep* because peers use communication to ensure that they remain synchronized throughout the game. As with *Tribes*, even though the deterministic lockstep model has existed for many years, it is still commonly used in modern RTS games. The other sample game built during the course of this book, *RoboCat RTS*, implements a deterministic lockstep model.

One of the largest differences between implementing networked multiplayer for an RTS instead of an FPS is the number of relevant units. In *Tribes*, even though there are up to 128 players, at any particular point in time only a fraction of these players is going to be relevant to a particular client. This means that the ghost manager in *Tribes* rarely has to send information about more than 20 to 30 ghosts at a time.

Contrast this with an RTS such as *Age of Empires*. Although the player cap is much smaller (limited to eight simultaneous players in the original game), each player can control a large number of units. The original *Age of Empires* capped the number of units for each player at 50, whereas in later games the cap was as high as 200. Using the cap of 50, this means that in a massive eight-player battle, there could be up to 400 units active at a time. Although it is natural to wonder if some sort of relevancy system could reduce the number of units that need to be synchronized, it's important to consider the worst-case scenario. What if a battle toward the end of a game featured the armies of all eight players? In this case, there are going to be several hundred units that are relevant at the same time. It would be hard for the synchronization to keep up even if a minimal amount of information is sent per unit.

To alleviate this issue, the engineers for *Age of Empires* decided to synchronize the *commands* each player issued, rather than synchronizing the units. There's a subtle but important distinction in this implementation—even a professional RTS player may be able to issue no more than 300 commands per minute. This means that even in an extreme case, the game need only transmit a few commands per second per each player. This requires a much more manageable amount of bandwidth than transmitting information about several hundred units. However, given that the game is no longer transmitting unit information over the network, each instance of the game needs to independently apply the commands transmitted by each player. Since each game instance is performing an independent simulation, it is of the utmost importance that each game instance remains synchronized with the other game instances. This ends up being the largest challenge of implementing the deterministic lockstep model.

Turn Timers

Since every game instance is performing an independent simulation, it makes sense to utilize a peer-to-peer topology. As discussed in Chapter 6, "Network Topologies and Sample Games," one advantage of a peer-to-peer model is that data can reach every computer more quickly. This is because the server is not acting as a middleman. However, one disadvantage is that each player needs to send their information to every other player, as opposed to just a single server. So for example, if player A issues an attack command, then every game instance needs to be aware of this attack command, or their simulations would diverge from each other.

However, there is another key factor to consider. Different players are going to run the game at different frame rates, and different players are going to have different quality connections. Going back to the example where player A issues an attack command, it's just as important that player A does not immediately apply the attack command. Instead, player A should only apply the attack command once players B, C, and D are all ready to simultaneously apply the command. But this introduces a conundrum: If player A's game waits too long to execute the attack command, the game will seem very unresponsive.

The solution to this problem is to introduce a **turn timer** to queue up commands. With the turn timer approach, first a turn length is selected—in the case of *Age of Empires*, the default

duration was 200 ms. All commands during these 200 ms are saved into a buffer. When the 200 ms are over, all the commands for that player's turn are transmitted over the network to all other players. Another key aspect of this system is a turn execution delay of two turns. What this means is that, for example, commands that are issued by the player on turn 50 will not be executed by any game until turn 52. In the case of a 200-ms turn timer, this means that the **input lag**, the amount of time it takes for a player's command to be displayed on screen, could be as high as 600 ms. However, the two turns of slack allows for every other player to receive and acknowledge the commands for a particular turn. It may seem slightly counterintuitive for an RTS game to actually have turns, but you can see the hallmarks of the turn timer approach in many different RTS games, including *StarCraft II*. Of course, modern games can have the luxury of shorter turn timers since bandwidth and latency are much better for most users today in comparison to the late 1990s.

There is one important edge case to consider with the turn timer approach. What happens if one of the players experiences a lag spike and they can no longer keep up with the 200-ms timer? Some games might temporarily pause the simulation to see if the lag spike can be overcome—eventually, the game may decide to drop the player if they continue to slow down the game for everyone else. *Age of Empires* also tries to compensate for this scenario by dynamically adjusting the rendering frame rate based on network conditions—thus a computer with a particularly slow Internet connection might allocate more time to receive data over the network, with less time being allotted for rendering graphics. For more detail on the dynamic turn adjustment, consult the original Bettner and Terrano article listed in the references.

There's also an extra benefit of transmitting the commands issued by the clients. With such an approach, it does not take much extra memory or work to save the commands issued over the course of an entire match. This directly leads to the possibility of implementing savable match replays, as in *Age of Empires II*. Replays are very popular in RTS games because it allows players to evaluate matches to gain a deeper understanding of strategies. It would require significantly more memory and overhead to create replays in an approach that transmitted unit information instead of commands.

Synchronization

Turn timers alone are not enough to guarantee synchronization between each peer. Since each machine is receiving and processing commands independently, it is of the utmost importance that each machine arrives at an identical result. In their paper, Bettner and Terrano write that "the difficulty with finding out-of-sync errors is that very subtle differences would multiply over time. A deer slightly out of alignment when the random map was created would forage slightly differently—and minutes later a villager would path a tiny bit off, or miss with his spear and take home no meat."

One concrete example arises from the fact that most games have some amount of randomness in actions. For instance, what if the game performs a random check in order to determine

whether or not an archer hits an infantry? It would be conceivable that player A's instance decides the archer does hit the infantry, whereas player B's instance decides the archer doesn't hit the infantry. The solution to this problem is to exploit the "pseudo" prefix of the **pseudo-random number generator** (PRNG). Since all PRNGs use some sort of seeding, the way you can guarantee both players A and B arrive at the same random results is to synchronize the seed value across all game instances. One should keep in mind, however, that a seed only guarantees a particular sequence of numbers. So not only is it important that each game instance uses the same seed, it's equally important that each game instance makes the same number of calls to the random generation number—otherwise the PRNG numbers will become out of sync. PRNG synchronization in a peer-to-peer configuration is further elaborated in Chapter 6, "Network Topologies and Sample Games."

There is also an implicit advantage to checking for synchronization—it reduces the opportunity for players to cheat. For example, if one player gives themselves 500 extra resources, the other game instances could immediately detect the desynchronization in the game state. It would then be trivial to kick the offending player out of the game. However, as with any system, there are tradeoffs—the fact that each game state simulates each unit in the game means that it is possible to create cheats that reveal information that should not be visible. This means that the so-called "map hacks" that reveal the entire map are still a common issue in most RTS games. This and other security concerns are covered in Chapter 10, "Security."

Summary

Networked multiplayer games have a lengthy history. They began as games playable on networks of mainframe computers, such as *Empire* (1973), which was playable on the PLATO network. Networked games later expanded to text-based multi-user dungeon games. These MUDs later expanded to bulletin board systems which allowed for users to dial in over phone lines.

In the early 1990s, local area network games, led by *Doom* (1993), took the computer gaming world by storm. These games allowed for players to locally connect multiple computers and play with or against each other. As adoption of the Internet expanded in the late 1990s, online games such as *Unreal* (1998) became very popular. Online games also started to see adoption on consoles in the early 2000s. One type of online game is the massively multiplayer online game, which supports hundreds if not thousands of players in the same game session at once.

Starsiege: Tribes (1998) implemented a network architecture still relevant to a modern-day action game. It uses a client-server model, so each player in the game is connected to a server that coordinates the game. At the lowest level, the platform packet module abstracts sending packets over the network. Next, the connection manager maintains connections between the players and the server, and provides delivery status notifications. The stream manager takes data from the higher-level managers (including the event, ghost, and move managers), and based on priority, adds this data to outgoing packets. The event manager takes important events, such as "player fired" and ensures that this data is received by the relevant parties. The ghost manager

handles sending object updates for the set of objects deemed relevant for a particular player. The move manager sends the most recent movement information for each player.

Age of Empires (1997) implemented a deterministic lockstep model. All computers in the game connect to each other in a peer-to-peer manner. Rather than sending information about each unit over the network, the game instead sends commands to each peer. These commands are then independently evaluated by each peer. In order to ensure the machines stay synchronized, a turn timer is used to save up commands over a period of time before sending them over the network. These commands are not executed until two turns later, which gives enough time for each peer to send and receive turn commands. Additionally, it is important that each peer runs a deterministic simulation, which means, for example, pseudo-random number generators need to be synchronized.

Review Questions

1. What is the difference between a local multiplayer game and a networked multiplayer game?
2. What are three different types of local network connections?
3. What is a major consideration when converting a networked game that works over a LAN to work over the Internet?
4. What is an MUD, and what type of game did it evolve into?
5. How does an MMO differ from a standard online game?
6. In the *Tribes* model, which system(s) provide reliability?
7. Describe how the ghost manager in the *Tribes* model reconstructs the minimal necessary transmission in the event that a packet is dropped.
8. In the *Age of Empires* peer-to-peer model, what is the purpose of the turn timer? What information is transmitted over the network to the other peers?

Additional Readings

Bettner, Paul and Mark Terrano. "1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond." Presented at the Game Developer's Conference, San Francisco, CA, 2001.

Frohnmayer, Mark and Tim Gift. "The Tribes Engine Networking Model." Presented at the Game Developer's Conference, San Francisco, CA, 2001.

Koster, Raph. "Online World Timeline." *Raph Koster's Website*. Last modified February 20, 2002. http://www.raphkoster.com/gaming/mudtimeline.shtml.

Kushner, David. *Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture*. New York: Random House, 2003.

Morningstar, Chip and F. Randall Farmer. "The Lessons of Lucasfilm's Habitat." In *Cyberspace: First Steps*, edited by Michael Benedikt, 273-301. Cambridge: MIT Press, 1991.

Wasserman, Ken and Tim Stryker. "Multimachine Games." *Byte Magazine*, December 1980, 24-40.

CHAPTER 2

THE INTERNET

This chapter provides an overview of the TCP/IP suite and the associated protocols and standards involved in Internet communication, including a deep dive into those which are most relevant for multiplayer game programming.

Origins: Packet Switching

The Internet as we know it today is a far cry from the four-node network as which it started life in late 1969. Originally known as ARPANET, it was developed by the United States Advanced Research Projects Agency with the stated goal of providing geographically dispersed scientists with access to uniquely powerful computers, similarly geographically dispersed.

ARPANET was to accomplish its goal using a newly invented technology called **packet switching**. Before the advent of packet switching, long-distance systems transmitted information through a process known as **circuit switching**. Systems using circuit switching sent information via a consistent circuit, created by dedicating and assembling smaller circuits into a longer path that persisted throughout the duration of the transmission. For instance, to send a large chunk of data, like a telephone call, from New York to Los Angeles, the circuit switching system would dedicate several smaller lines between intermediary cities to this chunk of information. It would connect them into a continuous circuit, and the circuit would persist until the system was done sending the information. In this case, it might reserve a line from New York to Chicago, a line from Chicago to Denver, and a line from Denver to Los Angeles. In reality these lines themselves consisted of smaller dedicated lines between closer cities. The lines would remain dedicated to this information until the transmission was complete; that is, until the telephone call was finished. After that the system could dedicate the lines to other information transmissions. This provided a very high quality of service for information transfer. However, it limited the usability of the lines in place, as the dedicated lines could only be used for one purpose at a time, as shown in Figure 2.1.

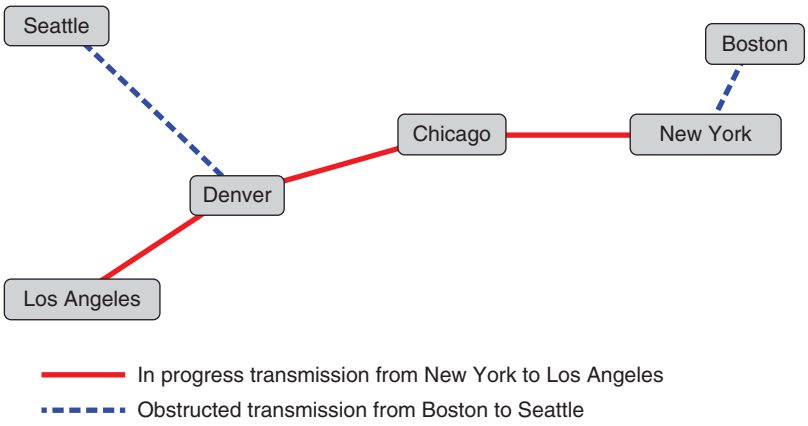

Figure 2.1 Circuit switching

Packet switching, however, provides increased usability by removing the requirement that a circuit be dedicated to a single transmission at a time. It achieves this by breaking up

transmissions into small chunks called **packets** and sending them down shared lines using a process called **store and forward**. Each node of the network is connected to other nodes in the network using a line that can carry packets between the nodes. Each node can store incoming packets and then forward them to a node closer to their final destination. For instance, in the call from New York to Los Angeles, the call would be broken up into very short packets of data. They would then be sent from New York to Chicago. When the Chicago node receives a packet, it examines the packet's destination and decides to forward the packet to Denver. The process continues until the packets arrive in Los Angeles and then the call receiver's telephone. The important distinction from circuit switching is that other phone conversations can happen at the same time, using the same lines. Other calls from New York to Los Angeles could have their packets forwarded along the same lines at the same time, as could a call from Boston to Seattle, or anywhere in between. Lines can hold packets from many, many transmissions at once, increasing usability, as shown in Figure 2.2.

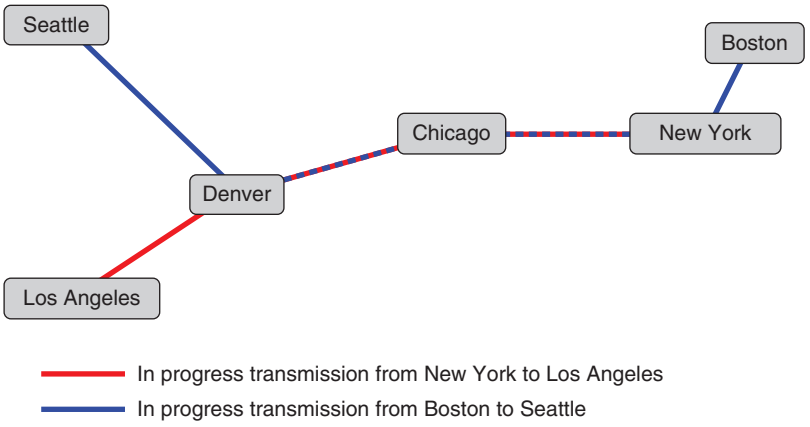

Figure 2.2 Packet switching

Packet switching itself is just a concept, though. Nodes on the network need a formal protocol collection to actually define how data should be packaged into packets and forwarded throughout the network. For the ARPANET, this protocol collection was defined in a paper known as the BBN Report 1822 and referred to as the 1822 protocol. Over many years, the ARPANET grew and grew and became part of the larger network now known as the **Internet**. During this time the protocols of the 1822 report evolved as well, becoming the protocols that drive the Internet of today. Together, they form a collection of protocols now known as the **TCP/IP suite**.

The TCP/IP Layer Cake

The TCP/IP suite is at once both a beautiful and frightening thing. It is beautiful because in theory it consists of a tower of independent and well-abstracted layers, each supported by

a variety of interchangeable protocols, bravely fulfilling their duties to support dependent layers and relay their data appropriately. It is frightening because these abstractions are often flagrantly violated by protocol authors in the name of performance, expandability, or some other worthwhile yet complexity-inducing excuse.

As multiplayer game programmers, our job is to understand the beauty and horror of the TCP/IP suite so that we can make our game functional and efficient. Usually this involves touching only the highest layers of the stack, but to do that effectively, it is useful to understand the underlying layers and how they affect the layers above them.

There are multiple models which explain the interactions of the layers used for Internet communication. *RFC 1122*, which defined early Internet host requirements, uses four layers: the link layer, the IP layer, the transport layer, and the application layer. The alternate Open Systems Interconnection (OSI) model uses seven layers: the physical layer, the data link layer, the network layer, the transport layer, the session layer, the presentation layer, and the application layer. To focus on matters relevant to game developers, this book uses a combined, five-model layer, consisting of the physical layer, the link layer, the network layer, the transport layer, and the application layer, as shown in Figure 2.3. Each layer has a duty, supporting the needs of the layer directly above it. Typically that duty includes

- Accepting a block of data to transmit from a higher layer
- Packaging the data up with a layer header and sometimes a footer
- Forwarding the data to a lower layer for further transmission
- Receiving transmitted data from a lower layer
- Unpackaging transmitted data by removing the header
- Forwarding transmitted data to a higher layer for further processing

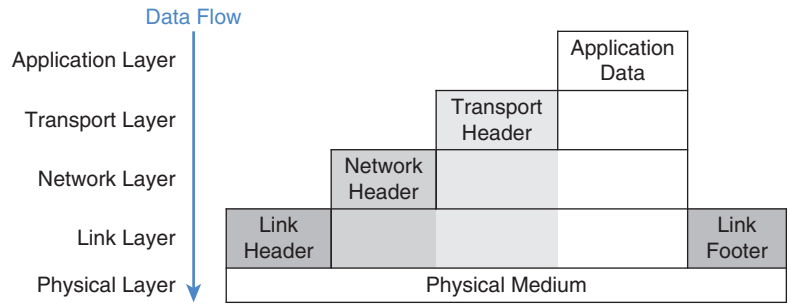

Figure 2.3 A game developer's view of the TCP/IP layer cake

The way a layer performs its duty, however, is not built into the definition of the layer. In fact, there are various protocols each layer can use to do its jobs, with some as old as the TCP/IP suite

and others currently being invented. For those familiar with object-oriented programming, it can be useful to think of each layer as an interface, and each protocol or collection of protocols as an implementation of that interface. Ideally, the details of a layer's implementation are abstracted away from the higher layers in the suite, but as mentioned previously that is not always true. The rest of this chapter presents an overview of the layers of the suite and some of the most common protocols employed to implement them.

The Physical Layer

At the very bottom of the layer cake is the most rudimentary, supporting layer: the **physical layer**. The physical layer's job is to provide a physical connection between networked computers, or hosts. A physical medium is necessary for the transmission of information. Twisted pair Cat 6 cable, phone lines, coaxial cable, and fiber optic cable are all examples of physical media that can provide the connection required by the physical layer.

Note that it is not necessary that the physical connection be tangible. As anyone with a mobile phone, tablet, or laptop can attest, radio waves also provide a perfectly good physical medium for the transmission of information. Some day soon, quantum entanglement may provide a physical medium for the transmission of information across great distances at instantaneous speeds, and when it does, the great layer cake of the Internet will be ready to accept it as a suitable implementation of its physical layer.

The Link Layer

The **link layer** is where the real computer science of the layer cake begins. Its job is to provide a method of communication between physically connected hosts. This means the link layer must provide a method through which a source host can package up information and transmit it through the physical layer, such that the intended destination host has a sporting chance of receiving the package and extracting the desired information.

At the link layer, a single unit of transmission is known as a **frame**. Using the link layer, hosts send frames to each other. Broken down more specifically, the duties of the link layer are to

- Define a way for a host to be identified such that a frame can be addressed to a specific destination.
- Define the format of a frame that includes the destination address and the data to be sent.
- Define the maximum size of a frame so that higher layers know how much data can be sent in a single transmission.
- Define a way to physically convert a frame into an electronic signal that can be sent over the physical layer and probably received by the intended host.

Note that delivery of the frame to the intended host is only probable, not guaranteed. There are many factors which influence whether the electronic signal actually arrives uncorrupted at its intended destination. A disruption in the physical medium, some kind of electrical interference, or an equipment failure could cause a frame to be dropped and never delivered. The link layer does not promise any effort will be made to determine if a frame arrives or resend it if it does not. For this reason, communication at the link layer level is referred to as unreliable. Any higher-layer protocol that needs guaranteed, or reliable, delivery of data must implement that guarantee itself.

For each physical medium which can be chosen to implement the physical layer, there is a corresponding protocol or list of protocols which provide the services necessary at the link layer. For instance, hosts connected by twisted pair cable can communicate using one of the Ethernet protocols such as 1000BASET. Hosts connected by radio waves can communicate using one of the short-range Wi-Fi protocols (e.g., 802.11g, 802.11n, 802.11ac) or one of the longer-range wireless protocols such as 3G or 4G. Table 2.1 lists some popular physical medium and link layer protocol combinations.

Table 2.1 Physical Medium and Link Layer Protocol Pairings

Physical Medium	Link Layer Protocol
Twisted pair	Ethernet 10BASET, Ethernet 100BASET, Ethernet 1000BASET
Twisted copper wire	Ethernet over copper (EoC)
2.4 GHz radio waves	802.11b, 802.11g, 802.11n
5 GHz radio waves	802.11n, 802.11ac
850 MHz radio waves	3G, 4G
Fiber optic cable	Fiber distributed data interface (FDDI), Ethernet 10GBASESR, Ethernet 10GBASELR
Coaxial cable	Ethernet over coax (also EoC), data over cable service interface specification (DOCSIS)

Because the link layer implementation and physical layer medium are so closely linked, some models group the two into a single layer. However, because some physical media support more than one link layer protocol, it can be useful to think of them as different layers.

It is important to note that an Internet connection between two distant hosts does not simply involve a single physical medium and a single link layer protocol. As will be explained in the following sections in the discussion of the remaining layers, several media and link layer protocols may be involved in the transmission of a single chunk of data. As such, many of the link layer protocols listed in the table may be employed while transmitting data for a networked computer game. Luckily, thanks to the abstraction of the TCP/IP suite, the details of the link

layer protocols used are mostly hidden from the game. Therefore, we will not explore in detail the inner workings of each of the existing link layer protocols. However, above all the rest, there is one link layer protocol group which both clearly illustrates the function of the link layer and is almost guaranteed to impact the working life of a networked game programmer in some way, and that is **Ethernet**.

Ethernet/802.3

Ethernet is not just a single protocol. It is a group of protocols all based on the original Ethernet blue book standard, published in 1980 by DEC, Intel, and Xerox. Collectively, modern Ethernet protocols are now defined under IEEE 802.3. There are varieties of Ethernet which run over fiber optic cable, twisted pair cable, or straight copper cable. There are varieties that run at different speeds: As of this writing, most desktop computers support gigabit speed Ethernet but 10 GB Ethernet standards exist and are growing in popularity.

To assign an identity to each host, Ethernet introduces the idea of the media access control address or **MAC address**. A MAC address is a theoretically unique 48-bit number assigned to each piece of hardware that can connect to an Ethernet network. Usually this hardware is referred to as a **network interface controller** or **NIC**. Originally, NICs were expansion cards, but due to the prevalence of the Internet, they have been built into most motherboards for the last few decades. When a host requires more than one connection to a network, or a connection to multiple networks, it is still common to add additional NICs as expansion cards, and such a host then has multiple MAC addresses, one for each NIC.

To keep MAC addresses universally unique, the NIC manufacturer burns the MAC address into the NIC during hardware production. The first 24 bits are an **organizationally unique identifier** or **OUI**, assigned by the IEEE to uniquely identify the manufacturer. It is then the manufacturer's responsibility to ensure the remaining 24 bits are uniquely assigned within the hardware it produces. In this way, each NIC produced should have a hardcoded, universally unique identifier by which it can be addressed.

The MAC address is such a useful concept that it is not used in just Ethernet. It is in fact used in most IEEE 802 link layer protocols, including Wi-Fi and Bluetooth.

> ### note
>
> Since its introduction, the MAC address has evolved in two significant ways. First, it is no longer reliable as a truly unique hardware identifier, as many NICs now allow software to arbitrarily change their MAC address. Second, to remedy a variety of pending issues, the IEEE has introduced the concept of a 64-bit MAC style address, called the extended unique identifier or EUI64. Where necessary, a 48-bit MAC address can be converted to an EUI64 by inserting the 2 bytes FFFE right after the OUI.

With a unique MAC address assigned to each host, Figure 2.4 specifies the format for an Ethernet packet, which wraps an Ethernet link layer frame.

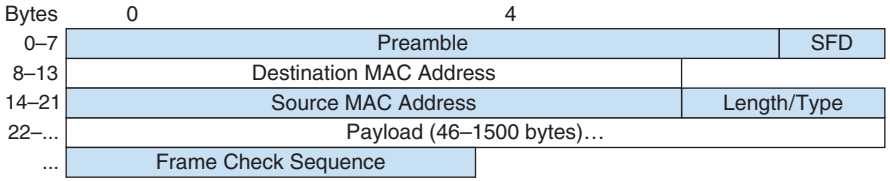

Figure 2.4 Ethernet packet structure

The **Preamble and start frame delimiter** (**SFD**) are the same for each packet and consist of the hex bytes 0×55 0×55 0×55 0×55 0×55 0×55 0×55 0×D5. This is a binary pattern that helps the underlying hardware sync up and prepare for the incoming frame. The Preamble and SFD are usually stripped from the packet by the NIC hardware, and the remaining bytes, comprising the frame, are passed to the Ethernet module for processing.

After the SFD are 6 bytes which represent the MAC address of the intended recipient of the frame. There is a special destination MAC address, FF:FF:FF:FF:FF:FF, known as the **broadcast address**, which indicates that the frame is intended for all hosts on the local area network.

The length/type field is overloaded and can be used to represent either length or type. When the field is used to represent length, it holds the size in bytes of the payload contained in the frame. However, when it is used to represent type, it contains an **EtherType** number which uniquely identifies the protocol that should be used to interpret the data inside the payload. When the Ethernet module receives this field, it must determine the correct way to interpret it. To assist with interpretation, the Ethernet standard defines the maximum length of the payload as 1500 bytes. This is known as the **maximum transmission unit**, or **MTU**, because it is the maximum amount of data that can be conveyed in a single transmission. The standard also defines the minimum EtherType value to be 0x0600, which is 1536. Thus, if the length/type field contains a number ≤1500, it represents a length, and if it contains a number ≥1536, it represents a protocol type.

> **note**
>
> Although not a standard, many modern Ethernet NICs support frames with MTUs higher than 1500 bytes. These **jumbo frames** can often have MTUs up to 9000 bytes. To support this, they specify an EtherType in the frame header and then rely on the underlying hardware to compute the size of the frame based on incoming data.

The payload itself is the data transmitted by this frame. Typically it is a network layer packet, having been passed onto the link layer for delivery to the appropriate host.

The **frame check sequence** (**FCS**) field holds a cyclic redundancy check (CRC32) value generated from the two address fields, the length/type field, the payload, and any padding. This way, as the Ethernet hardware reads in data, it can check for any corruption that occurred in transit and discard the frame if it did. Although Ethernet does not guarantee delivery of data, it makes a good effort to prevent delivery of corrupted data.

The specifics of the manner in which Ethernet packets are transmitted along the physical layer vary between media and are not relevant to the multiplayer game programmer. It suffices to say that each host on the network receives the frame, at which point the host reads the frame and determines if it is the intended recipient. If so, it extracts the payload data and processes it accordingly based on the value of the length/type field.

> **note**
>
> Initially, most small Ethernet networks used hardware known as **hubs** to connect multiple hosts together. Even older networks used a long coaxial cable strung between computers. In these style networks, the electronic signal for the Ethernet packet was literally sent to each host on the network, and it was up to the host to determine whether the packet was addressed to that host or not. This proved inefficient as networks grew. With the cost of hardware declining, most modern networks now use devices known as **switches** to connect hosts. Switches remember the MAC addresses, and sometimes the IPs, of the hosts connected to each of their ports, so most packets can be sent on the shortest path possible to their intended recipient, without having to visit every host on the network.

The Network Layer

The link layer provides a clear way to send data from an addressable host to one or more similarly addressable hosts. Therefore, it may be unclear why the TCP/IP suite requires any further layers. It turns out the link layer has several shortcomings which require a superior layer to address:

- Burned in MAC addresses limit hardware flexibility. Imagine you have a very popular webserver that thousands of users visit each day via Ethernet. If you were only using the link layer, queries to the server would need to be addressed via the MAC address of its Ethernet NIC. Now imagine that one day the NIC explodes in a very small ball of fire. When you install a replacement NIC, it will have a different MAC address, and thus your server will no longer receive requests from users. Clearly you need some easily configurable address system that lives on top of the MAC address.
- The link layer provides no support for segmenting the Internet into smaller, local area networks. If the entire Internet were run using just the link layer, all computers would have

to be connected in a single continuous network. Remember that Ethernet delivers each frame to every host on the network and allows the host to determine if it is the intended recipient. If the Internet used only Ethernet for communication, then each frame would have to travel to every single wired host on the planet. A few too many packets could bring the entire Internet to its knees. Also, there would be no ability to sanction different areas of the network into different security domains. It can be useful to easily broadcast a message to just the hosts in a local office, or just share files with the various computers in a house. With just the link layer there would be no ability to do this.

- The link layer provides no inherent support for communication between hosts using different link layer protocols. The fundamental idea behind allowing multiple physical and link layer protocols is that different networks can use the best implementation for their particular job. However, link layer protocols define no way of communicating from one link layer protocol to another. Again, you find yourself requiring an address system which sits on top of the hardware address system of the link layer.

The network layer's duty is to provide a logical address infrastructure on top of the link layer, such that host hardware can easily be replaced, groups of hosts can be segregated into subnetworks, and hosts on distant subnetworks, using different link layer protocols and different physical media can send messages to each other.

IPv4

Today, the most common protocol used to implement the required features of the network layer is **Internet protocol version 4** or **IPv4**. IPv4 fulfills its duties by defining a logical addressing system to name each host individually, a subnet system for defining logical subsections of the address space as physical subnetworks, and a routing system for forwarding data between subnets.

IP Address and Packet Structure

At the heart of IPv4 is the **IP address**. An IPv4 IP address is a 32-bit number, usually displayed to humans as four 8-bit numbers separated with periods. For example, the IP address of www.usc.edu is 128.125.253.146 and the IP address of www.mit.edu is 23.193.142.184. When read aloud, the periods are usually pronounced, "dot." With a unique IP address for each host on the Internet, a source host can direct a packet to a destination host simply by specifying the destination host's IP address in the header of the packet. There is an exception to IP address uniqueness, explained later in the section "Network Address Translation."

With the IP address defined, IPv4 then defines the structure of an IPv4 packet. The packet consists of a header, containing data necessary for implementing network layer functionality, and a payload, containing a higher layer's data to be transferred. Figure 2.5 gives the structure for an IPv4 packet.

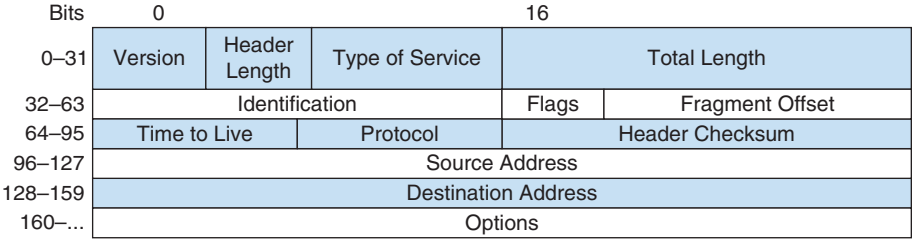

Figure 2.5 IPv4 header structure

Version (4 bits) specifies which version of the IP this packet supports. For IPv4, this is 4.

Header length (4 bits) specifies the length of the header in 32-bit words. Due to the optional fields at the end of an IP header, the header may be a variable length. The length field specifies exactly when the header ends and the encapsulated data begins. Because the length is specified in only 4 bits, it has a maximum value of 15, which means a header can be a maximum of 15 32-bit words, or 60 bytes. Because there are 20 bytes of mandatory information in the header, this field will never be less than 5.

Type of service (8 bits) is used to for a variety of purposes ranging from congestion control to differentiated services identification. For more information, see RFC 2474 and RFC 3168 in the "Additional Reading" section.

Packet length (16 bits) specifies the length in bytes of the entire packet, including header and payload. As the maximum number representable with 16 bits is 65535, the maximum packet size is clamped at 65535. As the minimum size of an IP header is 20 bytes, this means the maximum payload conveyable in an IPv4 packet is 65515 bytes.

Fragment identification (16 bits), **fragment flags** (3 bits), and **fragment offset** (13 bits), are used for reassembling fragmented packets, as explained later in the section "Fragmentation."

Time to live or **TTL** (8 bits) is used to limit the number of times a packet can be forwarded, as explained later in the section "Subnets and Indirect Routing."

Protocol (8 bits) specifies the protocol which should be used to interpret the contents of the payload. This is similar to the EtherType field in an Ethernet frame, in that it classifies a higher layer's encapsulated data.

Header checksum (16 bits) specifies a checksum that can be used to validate the integrity of the IPv4 header. Note that this is only for the header data. It is up to a higher layer to ensure integrity of the payload if required. Often, this is unnecessary, as many link layer protocols already contain a checksum to ensure integrity of their entire frame; for example, the FCS field in the Ethernet header.

Source address (32 bits) is the IP address of the packet's sender, and **destination address** (32 bits) is either the IP address of the packet's destination host, or a special address specifying delivery to more than one host.

> **note**
>
> The confusing manner of specifying header length in 32-bit words, but packet length in 8-bit words, suggests how important it is to conserve bandwidth. Because all possible headers are a multiple of 4-bytes long, their byte lengths are all evenly divisible by 4, and thus the last 2 bits of their byte lengths are always 0. Thus specifying the header length as units of 32-bit words saves 2 bits. Conserving bandwidth when possible is a golden rule of multiplayer game programming.

Direct Routing and Address Resolution Protocol

To understand how IPv4 allows packets to travel between networks with different link layer protocols, one must first understand how it delivers packets within a single network with a single link layer protocol. IPv4 allows packets to be targeted using an IP address. For the link layer to deliver a packet to the proper destination, it needs to be wrapped in a frame with an address the link layer can understand. Consider how Host A would send data to Host B in the network in Figure 2.6.

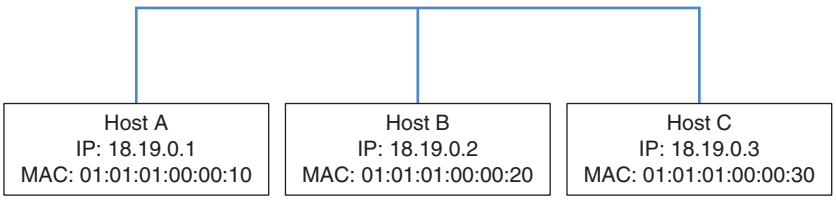

Figure 2.6 Three-host network

The sample network shown in Figure 2.6 contains three hosts, each with a single NIC, all connected by Ethernet. Host A wants to send a network layer packet to Host B at its IP address of 18.19.0.2. So, Host A prepares an IPv4 packet with a source IP address of 18.19.0.1 and a destination IP address of 18.19.0.2. In theory, the network layer should then hand off the packet to the link layer to perform the actual delivery. Unfortunately, the Ethernet module cannot deliver a packet purely by IP address, as IP is a network layer concept. The link layer needs some way to figure out the MAC address which corresponds to IP address 18.19.0.2. Luckily, there is a link layer protocol called the **address resolution protocol** (**ARP**), which provides a method for doing just that.

> **note**
>
> ARP is technically a link layer protocol because it sends out packets directly using link layer style addresses and does not require the routing between networks provided by the network layer. However, because the protocol violates some network layer abstractions by including network layer IP addresses, it can be useful to think of it more as a bridge between the layers than as a solely link layer protocol.

ARP consists of two main parts: a packet structure for querying the MAC address of the NIC associated with a particular IP address, and a table for keeping track of those pairings. A sample ARP table is shown in Table 2.2.

Table 2.2 An ARP Table Mapping from IP Address to MAC Address

IP Address	MAC Address
18.19.0.1	01:01:01:00:00:10
18.19.0.3	01:01:01:00:00:30

When the IP implementation needs to send a packet to a host using the link layer, it must first query the ARP table to fetch the MAC address associated with the destination IP address. If it finds the MAC address in the table, the IP module constructs a link layer frame using that MAC address and passes the frame to the link layer implementation for delivery. However, if the MAC address is not in the table, the ARP module attempts to determine the proper MAC address by sending out an ARP packet (Figure 2.7) to all reachable hosts on the link layer network.

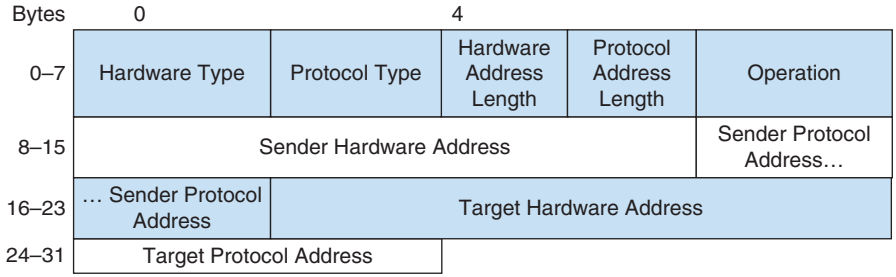

Figure 2.7 ARP packet structure

Hardware type (16 bits) defines the type of hardware on which the link layer is hosted. For Ethernet, this is 1.

Protocol type (16 bits) matches the EtherType value of the network layer protocol being used. For instance, IPv4 is 0×0800.

Hardware address length (8 bits) is the length in bytes of the link layer's hardware address. In most cases, this would be the MAC address size of 6 bytes.

Protocol address length (8 bits) is the length in bytes of the network layer's logical address. For IPv4, this is the IP address size of 4 bytes.

Operation (16 bits) is either 1 or 2, specifying whether this packet is a request for information (1) or a response (2).

Sender hardware address (variable length) is the hardware address of the sender of this packet and **sender protocol address** (variable length) is the network layer address of the sender of this packet. The lengths of these addresses match the lengths specified earlier in the packet.

Target hardware address (variable length) and **target protocol address** (variable length) are the corresponding addresses of the intended recipient of this packet. In the case of a request, the target hardware address is unknown and ignored by the receiver.

Continuing the previous example, if Host A doesn't know the MAC address of Host B, it prepares an ARP request packet with 1 in the Operation field, 18.19.0.1 in the sender protocol address field, 01:01:01:00:00:10 in the sender hardware field, and 18.19.0.2 in the target protocol address field. It then wraps this ARP packet in an Ethernet frame, which it sends to the Ethernet broadcast address FF:FF:FF:FF:FF:FF. Recall that this address specifies that the Ethernet frame should be delivered to and examined by each host on the network.

When Host C receives the packet, it does not respond because its IP address does not match the target protocol address in the packet. However, when Host B receives the packet, its IP does, so it responds with its own ARP packet containing its own addresses as the source and Host A's addresses as the target. When Host A receives the packet, it updates its ARP table with Host B's MAC address, and then wraps the waiting IP packet in an Ethernet frame and sends it off to Host B's MAC address.

note

When Host A broadcasts its initial ARP request to all hosts on the network, it includes both its MAC address and IP address. This gives all the other hosts on the network an opportunity to update their ARP tables with Host A's information even though they don't need it yet. This comes in handy if they ever have to talk to Host A, as they won't have to send out an ARP request packet first.

You may notice this system creates an interesting security vulnerability! A malicious host can send out ARP packets claiming to be any IP at all. Without a way to verify the authenticity of the ARP information, a switch might unintentionally route packets intended for one host to the malicious host. This not only allows sniffing packets, but could prevent intercepted packets from ever arriving at their intended host, thoroughly disrupting traffic on the network.

Subnets and Indirect Routing

Imagine two large companies, Company Alpha and Company Bravo. They each have their own large internal networks, Network Alpha and Network Bravo, respectively. Network Alpha contains 100 hosts, Host A1 to A100, and Network Bravo contains 100 hosts, Host B1 to B100. The two companies would like to connect their networks so they can send occasional messages back and forth, but simply connecting the networks with an Ethernet cable at the link layer presents a couple problems. Remember that an Ethernet packet must travel to each connected host on a network. Connecting Networks Alpha and Bravo at the link layer would cause each Ethernet packet to travel to 200 hosts instead of 100, effectively doubling the traffic on the entire network. It also presents a security risk, as it means all of Network Alpha's packets travel to Network Bravo, not just the ones intended for Network Bravo's hosts.

To allow Company Alpha and Company Bravo to connect their networks efficiently, the network layer introduces the ability to route packets between hosts on networks not directly connected at the link layer level. In fact, the Internet itself was originally conceived as a federation of such smaller networks throughout the country, joined by a few long-distance connections between them. The "inter" prefix on Internet, meaning, "between," represents these connections. It is the network layer's job to make this interaction between networks possible. Figure 2.8 illustrates a network layer connection between Networks Alpha and Bravo.

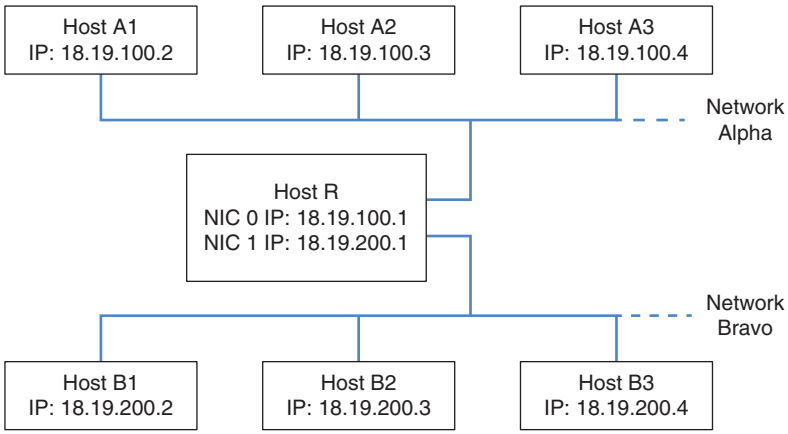

Figure 2.8 Connected networks Alpha and Bravo

Host R is a special type of host known as a **router**. A router has multiple NICs, each with its own IP address. In this case, one is connected to Network Alpha, and the other is connected to Network Bravo. Notice that all the IP addresses on Network Alpha share the prefix 18.19.100 and all the addresses on Network Bravo share the prefix 18.19.200. To understand why this is useful to our cause, we must now explore the subnet in more detail and define the concept of a subnet mask.

A **subnet mask** is a 32-bit number, usually written in the four-number, dotted notation typical of IP addresses. Hosts are said to be on the same subnet if their IP addresses, when bitwise ANDed with the subnet mask, yield the same result. For instance, if a subnet is defined as having a mask of 255.255.255.0, then 18.19.100.1 and 18.19.100.2 are both valid IP addresses on that subnet (Table 2.3). However, 18.19.200.1 is not on the subnet because it yields a different result when bitwise ANDed with the subnet mask.

Table 2.3 IP Addresses and Subnet Masks

Host	IP Address	Subnet Mask	IP Address ANDed with Subnet Mask
A1	18.19.100.1	255.255.255.0	18.19.100.0
A2	18.19.100.2	255.255.255.0	18.19.100.0
B1	18.19.200.1	255.255.255.0	18.19.200.0

In binary form, subnet masks are usually a string of 1s followed by a string of 0s, as this makes them easily human readable and human bitwise ANDable. Table 2.4 lists typical subnet masks and the number of unique hosts possible on the subnet. Note that two addresses on a subnet are always reserved and not usable by hosts. One is the **network address**, which is formed by bitwise ANDing the subnet mask with any IP address on the subnet. The other is the **broadcast address**, which is formed by bitwise ORing the network address with the bitwise complement of the subnet mask. That is, every bit in the network address that does not define the subnet should be set to 1. Packets addressed to the broadcast address for a subnet should be delivered to every host on the subnet.

Table 2.4 Sample Subnet Masks

Subnet Mask	Subnet Mask Binary	Significant Bits	Potential Host Count
255.255.255.248	11111111111111111111111111111000	29	6
255.255.255.192	11111111111111111111111111000000	26	62
255.255.255.0	11111111111111111111111100000000	24	254
255.255.0.0	11111111111111110000000000000000	16	65534
255.0.0.0	11111111000000000000000000000000	8	16777214

Because a subnet is, by definition, a group of hosts with IP addresses that yield the same result when bitwise ANDed with a subnet mask, a particular subnet can be defined simply by its subnet mask and network address. For instance, the subnet of Network Alpha is defined by network address 18.19.100.0 with subnet mask 255.255.255.0.

There is a common way to abbreviate this information, and that is known as **classless inter-domain routing** (**CIDR**) notation. A subnet mask in binary form is typically *n* ones followed by (32–*n*) zeroes. Therefore, a subnet can be notated as its network address followed by a forward slash and then the number of significant bits set in its subnet mask. For instance, the subnet of Network Alpha in Figure 2.8 is written using CIDR notation as 18.19.100.0/24.

note

The "classless" term in CIDR comes from the fact that inter-domain routing and address block assignment used to be based on three specifically sized classes of network. Class A networks had a subnet mask of 255.0.0.0, Class B networks had a subnet mask of 255.255.0.0, and Class C networks had a subnet mask of 255.255.255.0. For more on the evolution to CIDR, see RFC 1518 mentioned in the "Additional Reading" section.

With subnets defined, the IPv4 specification provides a way to move packets between hosts on different networks. This is made possible by the **routing table** present in the IP module of each host. Specifically, when the IPv4 module of a host is asked to send an IP packet to a remote host, it must decide whether to use the ARP table and direct routing, or some indirect route. To aid in this process, each IPv4 module contains a routing table. For each reachable destination subnet, the routing table contains a row with information on how packets should be delivered to that subnet. For the network in Figure 2.8, potential routing tables for Hosts A1, B1, and R are given in Tables 2.5, 2.6, and 2.7.

Table 2.5 Host A1 Routing Table

Row	Destination Subnet	Gateway	NIC
1	18.19.100.0/24		NIC 0 (18.19.100.2)
2	18.19.200.0/24	18.19.100.1	NIC 0 (18.19.100.2)

Table 2.6 Host B1 Routing Table

Row	Destination Subnet	Gateway	NIC
1	18.19.200.0/24		NIC 0 (18.19.200.2)
2	18.19.100.0/24	18.19.200.1	NIC 0 (18.19.200.2)

Table 2.7 Host R Routing Table

Row	Destination Subnet	Gateway	NIC
1	18.19.100.0/24		NIC 0 (18.19.100.1)
2	18.19.200.0/24		NIC 1 (18.19.200.1)

The destination subnet column refers to the subnet which contains the target IP address. The gateway column refers to the IP address of the next host, on the current subnet, which should be sent this packet via the link layer. It is required that this host be reachable through direct routing. If the gateway field is blank, it means the entire destination subnet is reachable through direct routing and the packet can be sent directly via the link layer. Finally, the NIC column identifies the NIC which should actually forward the packet. This is the mechanism by which a packet can be received from one link layer network and forwarded to another.

When Host A1 at 18.19.100.2 attempts to send a packet to Host B1 at 18.19.200.2, the following process occurs:

1. Host A1 builds an IP packet with source address 18.19.100.2 and destination address 18.19.200.2.
2. Host A1's IP module runs through the rows of its routing table from top to bottom, until it finds the first one with a destination subnet that contains the IP address 18.19.200.2. In this case, that is row 2. Note that the order of the rows is significant, as multiple rows might match a given address.
3. The gateway listed in row 2 is 18.19.100.1, so Host A1 uses ARP and its Ethernet module to wrap the packet in an Ethernet frame and send it to the MAC address that matches IP address 18.19.100.1. This arrives at Host R.
4. Host R's Ethernet module, running for its NIC 0 with IP address 18.19.100.1, receives the packet, detects the payload is an IP packet, and passes it up to its IP module.
5. Host R's IP module sees the packet is addressed to 18.19.200.1, so it attempts to forward the packet to 18.19.200.1.
6. Host R's IP module runs through its routing table until it finds a row whose destination subnet contains 18.19.200.1. In this case that is row 2.
7. Row 2 has no gateway, which means the subnet is directly routable. However, the NIC column specifies the use of the NIC 1 with IP address 18.19.200.1. This is the NIC connected to Network Bravo.
8. Host R's IP module passes the packet to the Ethernet module running for Host R's NIC 1. It uses ARP and the Ethernet module to wrap the packet in an Ethernet frame and send it to the MAC address that matches IP 18.19.200.1.
9. Host B1's Ethernet module receives the packet, detects the payload is an IP packet, and passes it up to its IP module.
10. Host B1's IP module sees that the destination IP address is its own. It sends the payload up to the next layer for more processing.

This example shows how two carefully configured networks communicate through indirect routing, but what if these networks need to send packets to the rest of the Internet? In that case, they first need a valid IP address and gateway from an **Internet Service Provider** (**ISP**).

For our purposes, assume they are assigned an IP address of 18.181.0.29 and a gateway of 18.181.0.1 by the ISP. The network administrator must then install an additional NIC into Host R and configure it with the IP address assigned. Finally, she must update the routing tables on Host R and all hosts on the network. Figure 2.9 shows the new network configuration and Tables 2.8, 2.9, and 2.10 show amended routing tables.

> **note**
>
> An ISP is not a special construct as far as the Internet is concerned. It's just a large organization, with its own very large block of IP addresses. What makes it interesting is that its main job is to take those IP addresses, break them into subnets, and then lease the subnets out to other organizations for use.

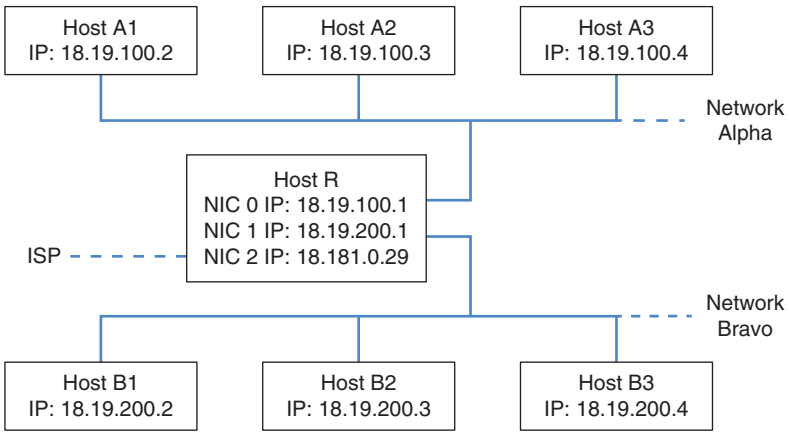

Figure 2.9 Networks Alpha and Bravo connected to the Internet

Table 2.8 Host A1 Routing Table with Internet Access

Row	Destination Subnet	Gateway	NIC
1	18.19.100.0/24		NIC 0 (18.19.100.2)
2	18.19.200.0/24	18.19.100.1	NIC 0 (18.19.100.2)
3	0.0.0.0/0	18.19.100.1	NIC 0 (18.19.100.2)

Table 2.9 Host B1 Routing Table with Internet Access

Row	Destination Subnet	Gateway	NIC
1	18.19.200.0/24		NIC 0 (18.19.200.2)
2	18.19.100.0/24	18.19.200.1	NIC 0 (18.19.200.2)
3	0.0.0.0/0	18.19.200.1	NIC 0 (18.19.200.2)

Table 2.10 Host R Routing Table with Internet Access

Row	Destination Subnet	Gateway	NIC
1	18.19.100.0/24		NIC 0 (18.19.100.1)
2	18.19.200.0/24		NIC 1 (18.19.200.1)
3	18.181.0.0/24	18.181.0.1	NIC 2 (18.181.0.29)
4	0.0.0.0/0	18.181.0.1	NIC 2 (18.181.0.29)

The destination 0.0.0.0/0 is known as the **default address**, because it defines a subnet which contains all IP addresses. If Host R receives a packet for a destination which does not match any of the first three rows, the destination will definitely match the subnet in the final row. In that case, the packet will be forwarded, via the new NIC, to the ISP's gateway, which should be able to set the packet on a path, from gateway to gateway, which will eventually terminate at the packet's intended destination. Similarly, Hosts A1 and B1 have new entries with the default address as their destination so that they can route Internet packets to Host R, which can then route them to the ISP.

Each time a packet is sent to a gateway and forwarded, the TTL field in the IPv4 header is decreased. When the TTL reaches 0, the packet is dropped by whichever host's IP module did the final decrementing. This prevents packets from circling the Internet forever if there happens to be cyclical routing information on the route. Changing the TTL requires recalculating the header checksum, which contributes to the time it takes hosts to process and forward a packet.

A TTL of 0 is not the only reason a packet might be dropped. For instance, if packets arrive at a router's NIC too rapidly for the NIC to process them, the NIC might just ignore them. Alternatively if packets arrive at a router on several NICs, but all need to be forwarded through a single NIC which isn't fast enough to handle them, some might be dropped. These are just some of the reasons an IP packet might be dropped on its journey from source to destination. As such, all protocols in the network layer, including IPv4, are unreliable. This means there is no guarantee that IPv4 packets, once sent, will arrive at their intended destination. Even if the packets do arrive, there is no guarantee they will arrive in their intended order, or that they will only arrive once. Network congestion may cause a router to route one packet onto one path and another packet with the same destination onto another path. These paths might be

different lengths and thus cause the latter packet to arrive first. Sometimes the same packet might get sent on multiple routes, causing it to arrive once and then arrive again a little later! Unreliability means no guarantee of delivery or delivery order.

IMPORTANT IP ADDRESSES

There are two special IP addresses worth mentioning. The first is the **loopback** or **localhost address**, 127.0.0.1. If an IP module is asked to send a packet to 127.0.0.1, it doesn't send it anywhere. It instead acts as if it just received the packet, and sends it up to the next layer for processing. Technically, the entire 127.0.0.0/8 address block should loopback, but some operating systems have firewall defaults which allow only packets addressed to 127.0.0.1 to do so completely.

The next is the **zero network broadcast address**, 255.255.255.255. This indicates the packet should be broadcast to all hosts on the current local link layer network but should not be passed through any routers. This is usually implemented by wrapping the packet in a link layer frame and sending it to the broadcast MAC address FF:FF:FF:FF:FF:FF.

Fragmentation

As mentioned earlier, the MTU, or maximum payload size, of an Ethernet frame is 1500 bytes. However, as noted previously, the maximum size of an IPv4 packet is 65535 bytes. This raises a question: If an IP packet must be transmitted by wrapping it in a link layer frame, how can it ever be larger than the link layer's MTU? The answer is **fragmentation**. If an IP module is asked to transmit a packet larger than the MTU of the target link layer, it can break the packet up into as many MTU-sized fragments as necessary.

IP packet fragments are just like regular IP packets, but with some specific values set in their headers. They make use of the fragment identification, fragment flags, and fragment offset fields of the header. When an IP module breaks an IP packet into a group of fragments, it creates a new IP packet for each fragment and sets the fields accordingly.

The fragment identification field (16 bits) holds a number which identifies the originally fragmented packet. Each fragment in a group has the same number in this field.

The fragment offset field (13 bits) specifies the offset, in 8-byte blocks, from the start of the original packet to the location in which this fragment's data belongs. This is necessarily a different number for each fragment within the group. The crazy numbering scheme is chosen so that any possible offset within a 65535-byte packet can be specified with only 13 bits. This requires that all offsets be even multiples of 8 bytes, because there is no ability to specify an offset with greater precision than that.

The fragment flags field (3 bits) is set to 0x4 for every fragment but the final fragment. This number is called the **more fragments flag**, representing that there are more fragments in the fragment group. If a host receives a packet with this flag set, it must wait until all fragments in the group are received before passing the reassembled packet up to a higher layer. This flag is not necessary on the final fragment, because it has a nonzero fragment offset field, similarly indicating that it is a member of a fragment group. In fact, the flag must be left off the final fragment to indicate that there are no further fragments in the original packet.

> **note**
>
> The fragment flags field has one other purpose. The original sender of an IP packet can set this to 0x2, a number known as the **do not fragment flag**. This specifies that the packet should not be fragmented under any circumstances. Instead, if an IP module must forward the packet on a link with an MTU smaller than the packet size, the packet should be dropped instead of fragmented.

Table 2.11 shows the relevant header fields for a large IP packet and the three packets into which it must be fragmented in order to forward it over an Ethernet link.

Table 2.11 IPv4 Packet Which Must Be Fragmented

Field	Original Packet Values	Fragment 1 Values	Fragment 2 Values	Fragment 3 Values
Version	4	4	4	4
Header length	20	20	20	20
Total length	3020	1500	1500	60
Identification	0	12	12	12
Fragment flags	0	0x4	0x4	0
Fragment offset	0	0	185	370
Time to live	64	64	64	64
Protocol	17	17	17	17
Source address	18.181.0.29	18.181.0.29	18.181.0.29	18.181.0.29
Destination address	181.10.19.2	181.10.19.2	181.10.19.2	181.10.19.2
Payload	3000 bytes	1480 bytes	1480 bytes	40 bytes

The fragment identification fields are all 12, indicating that the three fragments are all from the same packet. The number 12 is arbitrary, but it's likely this is the 12th fragmented packet this host has sent. The first fragment has the more fragments flag set and a packet offset of 0,

indicating that it contains the initial data from the unfragmented packet. Note that the packet length field indicates a total length of 1500. The IP module usually chooses to create fragments as large as possible to limit the number of fragments. Because the IP header is 20 bytes, this leaves 1480 for the fragment data. That suggests the second fragment's data should start at an offset of 1480. However, because the fragment offset field is represented in 8-byte blocks, and 1480/8 is 185, the actual number contained there is 185. The more fragments flag is also set on the second fragment. Finally, the third fragment has a data offset of 370 and does not have the more fragments flag set, indicating it is the final fragment. The total length of the third fragment is only 60, as the original packet had 3000 bytes of data inside its total length of 3020. Out of this 1480 bytes are in the first fragment, 1480 are in the second, and 40 are in the third.

After these fragment packets are sent out, it is conceivable that any or all of them could be further fragmented. This would happen if the route to the destination host involves traveling along a link layer with an even smaller MTU.

For the packet to be properly processed by the intended recipient, each of the packet fragments has to arrive at that final host and be reconstructed into the original, unfragmented packet. Because of network congestion, dynamically changing routing tables, or other reasons, it is possible that the packets arrive out of order, potentially interleaved with other packets from the same or other hosts. Whenever the first fragment arrives, the recipient's IP module has enough information to establish that the fragment is indeed a fragment and not a complete packet: This is evident from either the more fragments flag being set or the nonzero packet offset field. At this point, the recipient's IP module creates a 64-kB buffer (maximum packet size) and copies data from the fragment into the buffer at the appropriate offset. It tags the buffer with the sender's IP address and the fragment identification number, so that when future fragments come in with a matching sender and fragment identification, the IP module can fetch the appropriate buffer and copy in the new data. When a fragment arrives without the more fragments flag set, the recipient calculates the total length of the original packet by adding that fragment's data length to its packet offset. When all data for a packet has arrived, the IP module passes the fully reconstructed packet up to the next layer for further processing.

tip

Although IP packet fragmentation makes it possible to send giant packets, it introduces two large inefficiencies. First, it actually increases the amount of data which must be sent over the network. Table 2.11 illustrates that a 3020-byte packet gets fragmented into two 1500-bytes packets and a 60-byte packet, for a total of 3060 bytes. This isn't a terrible amount, but it can add up. Second, if a single fragment is lost in transit, the receiving host must drop the entire packet. This makes it more likely that larger packets with many fragments get dropped. For this reason, it is generally advisable to avoid fragmentation entirely by making sure all IP packets are smaller than the link layer MTU. This is not necessarily easy, because

there can be several different link layer protocols in between two hosts: Imagine a packet traveling from New York to Japan. It is very likely that at least one of the link layers between the two hosts will use Ethernet, so game developers make the approximation that the minimum MTU of the entire packet route will be 1500 bytes. This 1500 bytes must encapsulate the 20-byte IP header, the IP payload, and any additional data required by wrapper protocols like VPN or IPSec that may be in use. For this reason, it is wise to limit IP payloads to around 1300 bytes.

At first thought, it may seem better to limit packet size to something even smaller, like 100 bytes. After all, if a 1500-byte packet is unlikely to require fragmentation, a 100-byte packet is even less likely to require it, right? This may be true, but remember that each packet requires a header of 20 bytes. A game sending out packets that are only 100 bytes in length is spending 20% of its bandwidth on just IP headers, which is very inefficient. For this reason, once you've decided that there is a very good chance the minimum MTU is 1500, you want to send out packets that are as close to 1500 in size as possible. This would mean that only 1.3% of your bandwidth is wasted on IP headers, which is much better than 20%!

IPv6

IPv4, with its 32-bit addresses, allows for 4 billion unique IP addresses. Thanks to private networks and network address translation (discussed later in this chapter) it is possible for quite a few more hosts than that to actively communicate on the Internet. Nevertheless, due to the way IP addresses are allotted, and the proliferation of PCs, mobile devices, and the Internet of Things, the world is running out of 32-bit IP addresses. IPv6 was created to address both this problem, and some inefficiencies that have become evident throughout the long life of IPv4.

For the next few years, IPv6 will probably remain of low importance to game developers. As of July 2014, Google reports that roughly 4% of its users access its site through IPv6, which is probably a good indication of how many end users in general are using devices connecting to the Internet through IPv6. As such, games still have to handle all the idiosyncrasies and oddities of IPv4 that IPv6 was designed to fix. Nevertheless, as next gen platforms like the Xbox One gain in popularity, IPv6 will eventually replace IPv4, and it is worth briefly exploring what IPv6 is all about.

The most noticeable new feature of IPv6 is its new IP address length of 128 bits. IPv6 addresses are written as eight groups of 4-digit hex numbers, separated by colons. Table 2.12 shows a typical IPv6 address in three accepted forms.

Table 2.12 Typical IPv6 Address Forms

Form	Address
Unabbreviated	2001:4a60:0000:8f1:0000:0000:0000:1013
Leading zeroes dropped	2001:4a60:0:8f1:0:0:0:1013
Single run of zeroes removed	2001:4a60:0:8f1::1013

When written, leading zeroes in each hextet may be dropped. Additionally, a single run of zeroes may be abbreviated with a double colon. Because the address is always 16 bytes, it is simple to reconstruct the original form by replacing all missing digits with zeroes.

The first 64 bits of an IPv6 address typically represent the network and are called the network **prefix**, whereas the final 64 bits represent the individual host and are called the **interface identifier**. When it is important for a host to have a consistent IP address, such as when acting as a server, a network administrator may manually assign the interface identifier, similar to how IP addresses are manually assigned for IPv4. A host that does not need to be easy to find by remote clients can also chose its interface identifier at random and announce it to the network, as chances of a collision in the 64-bit space are low. Most often, the interface identifier is automatically set to the NIC's EUI-64, as this is already guaranteed to be unique.

Neighbor discovery protocol (**NDP**) replaces ARP as well as some of the features of DHCP, as described later in this chapter. Using NDP, routers advertise their network prefixes and routing table information, and hosts query and announce their IP addresses and link layer addresses. More information on NDP can be found in RFC 4861, referenced in the "Additional Reading" section.

Another nice change from IPv4 is that IPv6 no longer supports packet fragmentation at the router level. This enables the removal of all the fragmentation-related fields from the IP header and saves some bandwidth on each packet. If an IPv6 packet reaches a router and is too big for the outgoing link layer, the router simply drops the packet and responds to the sender that the packet was too big. It is up to the sender to try again with a smaller packet.

More information on IPv6 can be found in RFC 2460, referenced in the "Additional Reading" section.

The Transport Layer

While the network layer's job is to facilitate communication between distant hosts on remote networks, the **transport layer**'s job is to enable communication between individual processes on those hosts. Because multiple processes can be running on a single host, it is not always enough to know that Host A sent an IP packet to Host B: When Host B receives the IP packet,

it needs to know which process should be passed the contents for further processing. To solve this, the transport layer introduces the concept of **ports**. A port is a 16-bit, unsigned number representing a communication endpoint at a particular host. If the IP address is like a physical street address of a building, a port is a bit like a suite number inside that building. An individual process can then be thought of as a tenant who can fetch the mail from one or more suites inside that building. Using a transport layer module, a process can **bind** to a specific port, telling the transport layer module that it would like to be passed any communication addressed to that port.

As mentioned, all ports are 16-bit numbers. In theory, a process can bind to any port and use it for any communicative purpose it wants. However, problems arise if two processes on the same host attempt to bind to the same port. Imagine that both a webserver program and an email program bind to port 20. If the transport layer module receives data for port 20, should it deliver that data to both processes? If so, the webserver might interpret incoming email data as a web request, or the email program might interpret an incoming web request as email. This will end up making either a web surfer, or an emailer very confused. For this reason, most implementations require special flags for multiple processes to bind the same port.

To help avoid processes squabbling over ports, a department of the **Internet Corporation for Assigned Names and Numbers (ICANN)** known as the **Internet Assigned Numbers Authority** (**IANA**) maintains a port number registry with which various protocol and application developers can register the ports their applications use. There is only a single registrant per port number per transport layer protocol. Port numbers 1024-49151 are known as the **user ports** or **registered ports**. Any protocol and application developer can formally request a port number from this range from IANA, and after a review process, the port registration may be granted. If a user port number is registered with the IANA for a certain application or protocol, then it is considered bad form for any other application or protocol implementation to bind to that port, although most transport layer implementations do not prevent it.

Ports 0 to 1023 are known as the **system ports** or **reserved ports**. These ports are similar to the user ports, but their registration with IANA is more restricted and subject to more thorough review. These ports are special because most operating systems allow only root level processes to bind system ports, allowing them to be used for purposes requiring elevated levels of security.

Finally, ports 49152 to 65535 are known as **dynamic ports**. These are never assigned by IANA and are fair game for any process to use. If a process attempts to bind to a dynamic port and finds that it is in use, it should handle that gracefully by attempting to bind to other dynamic ports until an available one is found. As a good Internet citizen, you should use only dynamic ports while building your multiplayer games, and then register with IANA for a user port assignment if necessary.

Once an application has identified a port to use, it must employ a transport layer protocol to actually send data. Sample transport layer protocols, as well as their IP protocol number, are listed in Table 2.13. As game developers we deal primarily with UDP and TCP.

Table 2.13 Examples of Transport Layer Protocols

Name	Acronym	Protocol Number
Transmission control protocol	TCP	6
User datagram protocol	UDP	17
Datagram congestion control protocol	DCCP	33
Stream control transmission protocol	SCTP	132

> **tip**
>
> IP addresses and ports are often combined with a colon to indicate a complete source or destination address. So, a packet heading to IP 18.19.20.21 and port 80 would have its destination written as 18.19.20.21:80.

UDP

User datagram protocol (**UDP**) is a lightweight protocol for wrapping data and sending it from a port on one host to a port on another host. A UDP datagram consists of an 8-byte header followed by the payload data. Figure 2.10 shows the format of a UDP header.

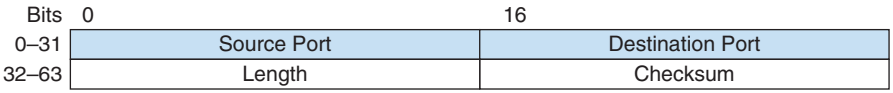

Figure 2.10 UDP header

Source port (16 bits) identifies the port from which the datagram originated. This is useful if the recipient of the datagram wishes to respond.

Destination port (16 bits) is the target port of the datagram. The UDP module delivers the datagram to whichever process has bound this port.

Length (16 bits) is the length of the UDP header and payload.

Checksum (16 bits) is an optional checksum calculated based on the UDP header, payload, and certain fields of the IP header. If not calculated, this field is all zeroes. Often this field is ignored because lower layers validate the data.

UDP is very much a no-frills protocol. Each datagram is a self-contained entity, relying on no shared state between the two hosts. It can be thought of as a postcard, dropped in the mail, and then forgotten. UDP provides no effort to limit traffic on a clogged network, deliver data in order, or guarantee that data is delivered at all. This is all very much in contrast to the next transport layer we will explore, TCP.

TCP

Whereas UDP allows the transfer of discreet datagrams between hosts, **transmission control protocol** (**TCP**) enables the creation of a persistent connection between two hosts followed by the reliable transfer of a stream of data. The key word here is reliable. Unlike every protocol discussed so far, TCP does its best to ensure all data sent is received, in its intended order, at its intended recipient. To effect this, it requires a larger header than UDP, and nontrivial connection state tracking at each host participating in the connection. This enables recipients to acknowledge received data, and senders to resend any transmissions that are unacknowledged.

A TCP unit of data transmission is called a TCP **segment**. This refers to the fact that TCP is built for transmitting a large stream of data and each lower layer packet wraps a single segment of that stream. A segment consists of a TCP header followed by the data for that segment. Figure 2.11 shows its structure.

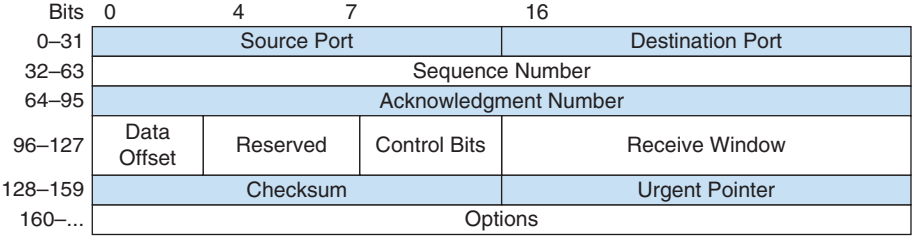

Figure 2.11 TCP header

Source port (16 bits) and **destination port** (16 bits) are transport layer port numbers.

Sequence number (32-bits) is a monotonically increasing identifier. Each byte transferred via TCP has a consecutive sequence number which serves as a unique identifier of that byte. This way, the sender can label data being sent and the recipient can acknowledge it. The sequence

number of a segment is typically the sequence number of the first byte of data in that segment. There is an exception when establishing the initial connection, as explained in the "Three-Way Handshake" section.

Acknowledgment number (32-bits) contains the sequence number of the next byte of data that the sender is expecting to receive. This acts as a de facto acknowledgment for all data with sequence numbers lower than this number: Because TCP guarantees all data is delivered in order, the sequence number of the next byte that a host expects to receive is always one more than the sequence number of the previous byte that it has received. Be careful to remember that the sender of this number is not actually acknowledging receipt of the sequence number with this value, but actually of all sequence numbers *lower* than this value.

Data offset (4 bits) specifies the length of the header in 32-bit words. TCP allows for some optional header elements at the end of its header, so there can be from 20 to 64 bytes between the start of the header and the data of the segment.

Control bits (9 bits) hold metadata about the header. They are discussed later where relevant.

Receive window (16 bits) conveys the maximum amount of remaining buffer space the sender has for incoming data. This is useful for maintaining flow control, as discussed later.

Urgent pointer (16 bits) holds the delta between the first byte of data in this segment and the first byte of urgent data. This is only relevant if the URG flag is set in the control bits.

> note
>
> Instead of using the loosely defined "byte" to refer to 8 bits, many RFCs, including those that define the major transport layer protocols, unambiguously refer to 8-bit sized chunks of data as **octets**. Some legacy platforms used bytes that contained more or fewer than 8 bits, and the standardization around an octet of bits helped ensure compatibility between platforms. This is less of an issue these days, as all platforms relevant to game developers treat a byte as 8 bits.

Reliability

Figure 2.12 illustrates the general manner in which TCP brings about reliable data transfer between two hosts. In short, the source host sends a uniquely identified packet to the destination host. It then waits for a response packet from the destination host, acknowledging receipt of the packet. If it does not receive the expected acknowledgment within a certain amount of time, it resends the original packet. This continues until all data has been sent and acknowledged.

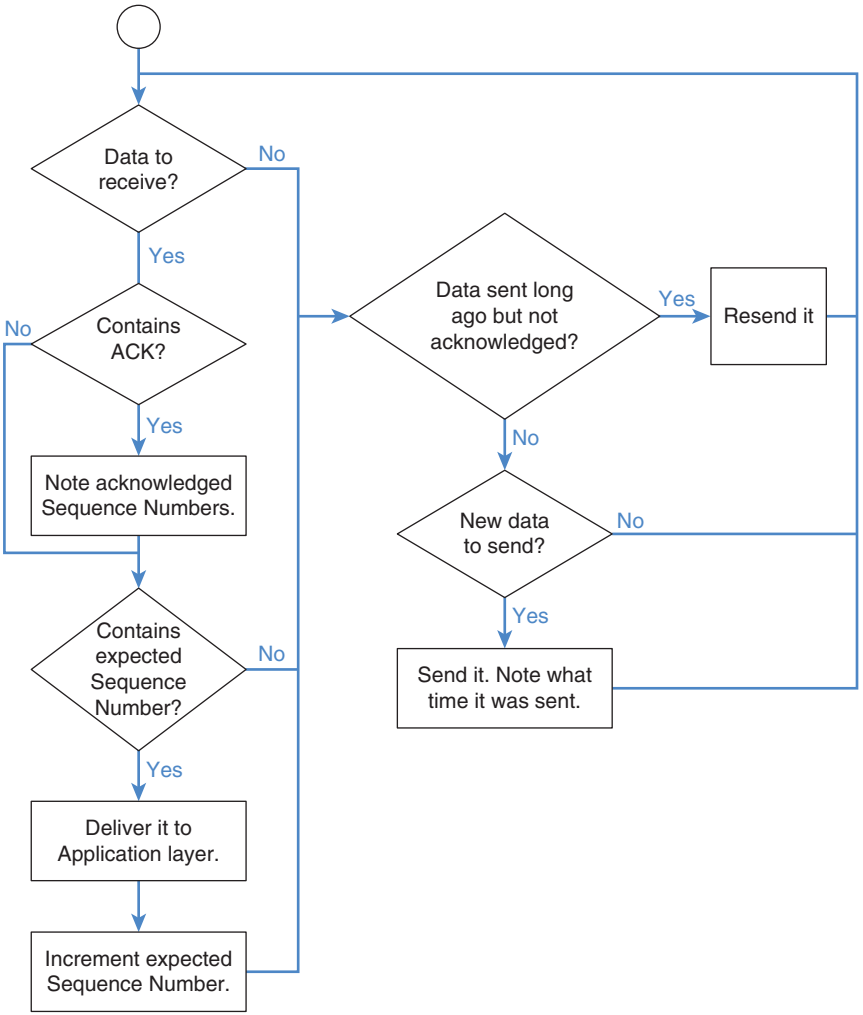

Figure 2.12 TCP reliable data transfer flow chart

The exact details of this process are slightly more complicated, but worth understanding in depth, as they provide an excellent case study of a reliable data transfer system. Because the TCP strategy involves resending data and tracking expected sequence numbers, each host must maintain state for all active TCP connections. Table 2.14 lists some of the state variables they must maintain and their standard abbreviations as defined by RFC 793. The process of initializing that state begins with a three-way handshake between the two hosts.

Table 2.14 TCP State Variables

Variable	Abbreviation	Definition
Send Next	SND.NXT	The sequence number of the next segment the host will send
Send Unacknowledged	SND.UNA	The sequence number of the oldest byte sent by the host that has not yet been acknowledged
Send Window	SND.WND	The current amount of data the host is allowed to send before receiving an acknowledgment for unacknowledged data
Receive Next	RCV.NXT	The next sequence number the host expects to receive
Receive Window	RCV.WND	The current amount of data the host is able to receive without overflowing its receive buffer

Three-Way Handshake

Figure 2.13 illustrates a three-way handshake between Hosts A and B. In the figure, Host A initiates the connection by sending the first segment. This segment has the SYN flag set and a randomly chosen initial sequence number of 1000. This indicates to Host B that Host A would like to initiate a TCP connection starting at sequence number 1000, and that Host B should initialize resources necessary to maintain the connection state.

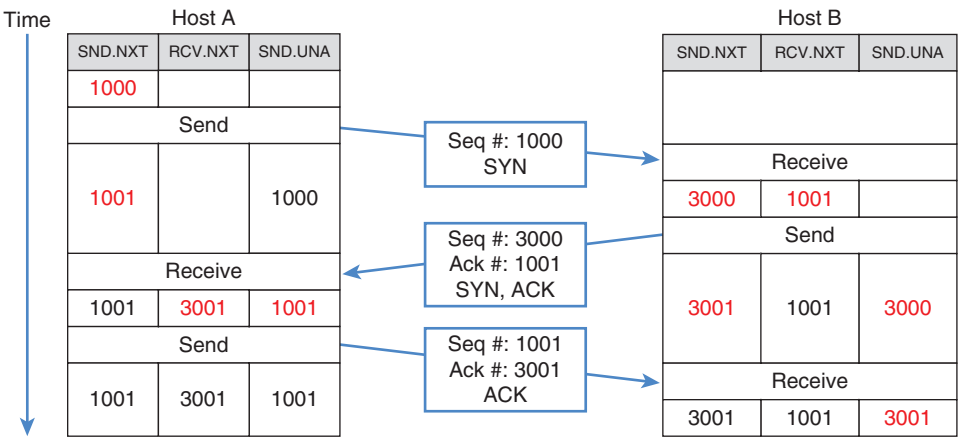

Figure 2.13 TCP three-way handshake

Host B, if it is willing and able to open the connection, then responds with a packet with both the SYN flag, and the ACK flag set. It acknowledges Host A's sequence number by setting the acknowledgment number on the segment to Host A's initial sequence number plus 1. This

means the next segment Host B is expecting from Host A should have a sequence number one higher than the previous segment. In addition, Host B picks its own random sequence number, 3000, to start its stream of data to Host A. It is important to note that Hosts A and B each picked their own random starting sequence numbers. There are two separate streams of data involved in the connection: One from Host A to Host B, which uses Host A's numbering, and one from Host B to Host A which uses Host B's numbering. The presence of the SYN flag in a segment means "Hey you! I'm going to start sending you a stream of data starting with a byte labeled one plus the sequence number mentioned in this segment." The presence of the ACK flag and the acknowledgment number in the second segment means "Oh by the way, I received all data you sent up until this sequence number, so this sequence number is what I'm expecting in the next segment you send me." When Host A receives this segment, all that's left is for it to ACK Host B's initial sequence number, so it sends out a segment with the ACK flag set and Host B's sequence number plus 1, 3001, in the acknowledgment field.

> **note**
>
> When a TCP segment contains a SYN or FIN flag, the sequence number is incremented by an extra byte to represent the presence of the flag. This is sometimes known as the **TCP phantom byte**.

Reliability is established through the careful sending and acknowledgment of data. If a timeout expires and Host A never receives the SYN-ACK segment, it knows that Host B either never received the SYN segment, or Host B's response was lost. Either way, Host A can resend the initial segment. If Host B did indeed receive the SYN segment and therefore receives it for a second time, Host B knows it is because Host A did not receive its SYN-ACK response, so it can resend the SYN-ACK segment.

Data Transmission

To transmit data, hosts can include a payload in each outgoing segment. Each segment is tagged with the sequence number of the first byte of data in the sequence. Remember that each byte has a consecutive sequence number, so this effectively means that the sequence number of a segment should be the sequence number of the previous segment plus the amount of data in the previous segment. Meanwhile, each time an incoming data segment arrives at its destination, the receiver sends out an acknowledgment packet with the acknowledgment field set to the next sequence number it expects to receive. This would typically be the sequence number of the most recently received segment plus the amount of data in that segment. Figure 2.14 shows a simple transmission with no dropped segments. Host A sends 100 bytes in its first segment, Host B acknowledges and sends 50 bytes of its own, Host A sends 200 bytes more, and then Host B acknowledges those 200 bytes without sending any additional data.

Things get slightly more complicated when a segment gets dropped or delivered out of order. In Figure 2.15, segment 1301 traveling from Host A to Host B is lost. Host A expects to receive

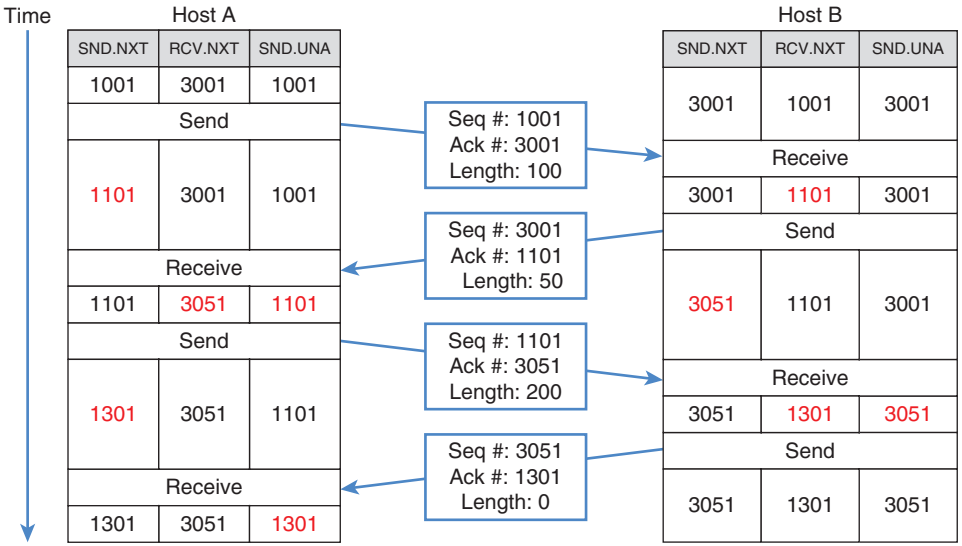

Figure 2.14 TCP transmission with no packet loss

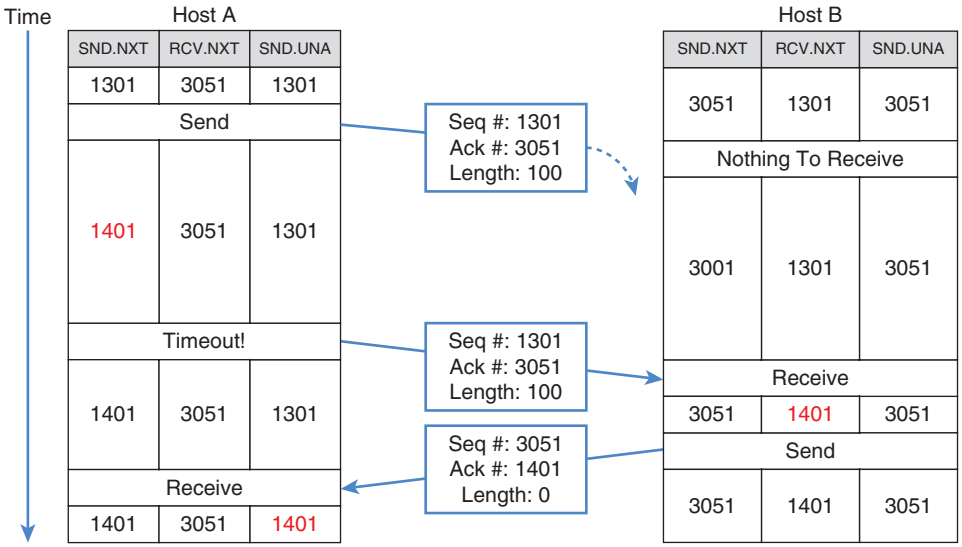

Figure 2.15 TCP packet lost and retransmitted

an ACK packet with 1301 in the acknowledgment field. When a certain time limit expires and Host A has not received the ACK, it knows something is wrong. Either segment 1301, or the ACK from Host B has been dropped. Either way, it knows it needs to redeliver segment 1301 until it receives an acknowledgment from Host B. To redeliver the segment, Host A needs to have a

copy of that segment's data on hand, and this is a key component of TCP's operation: The TCP module must store every byte it sends out until that byte is acknowledged by the recipient. Only once an acknowledgment for a segment is received can the TCP module purge that segment's data from its memory.

TCP guarantees that data is delivered in order, so if a host receives a packet with a sequence number it is not yet expecting, it has two options. The simple option is to just drop the packet and wait for it to be resent in order. An alternative option is to buffer it while neither ACKing it nor delivering it to the application layer for processing. Instead, the host copies it into its local stream buffer at the appropriate position based on the sequence number. Then, when all preceding sequence numbers have been delivered, the host can ACK the out of order packet and send it to the application layer for processing without requiring the sender to resend it.

In the preceding examples, Host A always waits for an acknowledgment before sending additional data. This is unusual and contrived just for the purpose of simplifying the examples. There is no requirement that Host A must stall its transmission, waiting for an acknowledgment after each segment it sends. In fact, if there were such a requirement, TCP would be a fairly unusable protocol over long distances.

Recall that the MTU for Ethernet is 1500 bytes. The IPv4 header takes up at least 20 of those bytes and the TCP header takes up at least another 20 bytes, which means the most data that can be sent in an unfragmented TCP segment that travels over Ethernet is 1460 bytes. This is known as the **maximum segment size** (**MSS**). If a TCP connection could only have one unacknowledged segment in flight at a time, then its bandwidth would be severely limited. In fact, it would be the MSS divided by the amount of time it takes for the segment to go from sender to receiver plus the time for the acknowledgment to return from receiver to sender (**round trip time** or **RTT**). Round trip times across the country can be on the order of 30 ms. This means the maximum cross-country bandwidth achievable with TCP, regardless of intervening link layer speed, would be 1500 bytes/0.03 seconds, or 50 kbps. That might be a decent speed for 1993, but not for today!

To avoid this problem, a TCP connection is allowed to have multiple unacknowledged segments in flight at once. However, it cannot have an unlimited number of segments in flight, as this would present another problem. When transport layer data arrives at a host, it is held in a buffer until the process which has bound the corresponding port consumes it. At that point, it is removed from the buffer. No matter how much memory is available on the host, the buffer itself is of some fixed size. It is conceivable that a complex process on a slow CPU may not consume incoming data as fast as it arrives. Thus, the buffer will fill up and incoming data will be dropped. In the case of TCP, this means the data will not be acknowledged, and the rapidly transmitting sender will then begin rapidly resending the data. In all likelihood, most of this resent data will be dropped as well, because the receiving host still has the same slow CPU and is still running the same complex process. This causes a big traffic jam and is a colossal waste of Internet resources.

To prevent this calamity, TCP implements a process known as **flow control**. Flow control prevents a rapidly transmitting host from overwhelming a slowly consuming one. Each TCP header contains a receive window field which specifies how much receive buffer space the sender of the packet has available. This equates to telling the other host the maximum amount of data it should send before stopping to wait for an acknowledgment. Figure 2.16 illustrates an exchange of packets between a rapidly transmitting Host A and a slowly consuming Host B.

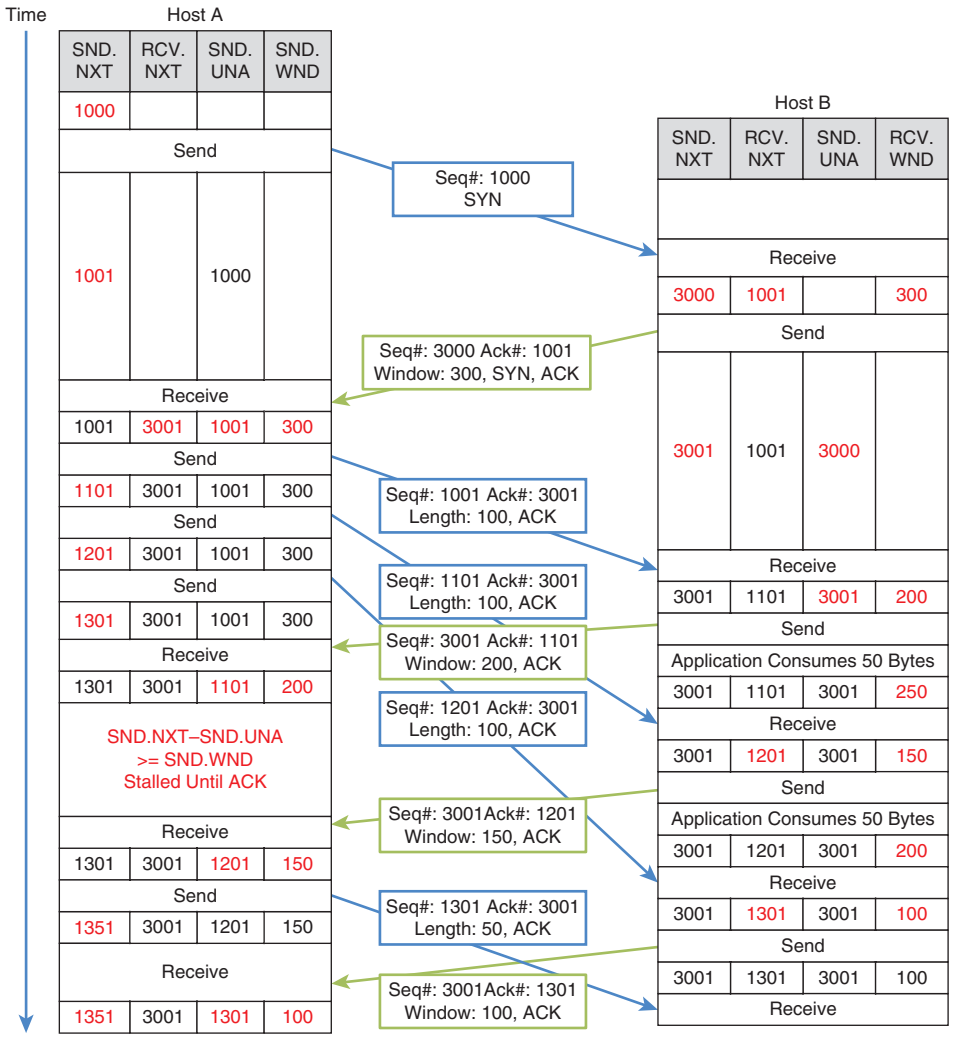

Figure 2.16 TCP flow control

For demonstration purposes, an MSS of 100 bytes is used. Host B's initial SYN-ACK flag specifies a receive window of 300 bytes, so Host A only sends out three 100-byte segments before pausing to wait for an ACK from Host B. When Host B finally sends an ACK, it knows it now has 100 bytes in its buffer which might not be consumed quickly, so it tells Host A to limit its receive window to 200 bytes. Host A knows 200 more bytes are already on their way to B, so it doesn't send any more data in response. It must stall until it receives an ACK from Host B. By the time Host B ACKs the second packet, 50 bytes of data from its buffer have been consumed, so it has a total of 150 bytes remaining in its buffer and 150 bytes free. When it sends an ACK to Host A, it tells Host A to limit the receive window to only 150 bytes. Host A knows at this point there are still 100 unacknowledged bytes in flight, but the receive window is 150 bytes, so it sends an additional 50-byte segment off to Host B.

Flow control continues in this way, with Host B always alerting Host A to how much data it can hold so that Host A never sends out more data than Host B can buffer. With that in mind, the theoretical bandwidth limit for a TCP stream of data is given by this equation:

$$BandwidthLimit \times \frac{ReceiveWindow}{RoundTripTime}$$

Having too small a receive window can create a bottleneck for TCP transmission. To avoid this, a large enough receive window should be chosen such that the theoretical bandwidth maximum is always greater than the maximum transmission rate of the link layer in between the hosts.

Notice that in Figure 2.16, Host B ends up sending two ACK packets in a row to Host A. This is not a very efficient use of bandwidth, as the acknowledgment number in the second ACK packet sufficiently acknowledges all the bytes that the first ACK packet acknowledges. Due to the IP and TCP headers alone, this wastes 40 bytes of bandwidth from Host B to Host A. When link layer frames are factored in, this wastes even more. To prevent this inefficiency, TCP rules allow for something called a **delayed acknowledgment**. According to the specification, a host receiving a TCP segment does not have to immediately respond with an acknowledgment. Instead, it can wait up to 500 ms, or until the next segment is received, whichever occurs first. In the previous example, if Host B receives the segment with sequence number 1101 within 500 ms of the segment with sequence number 1001, Host B only has to send an acknowledgment for segment 1101. For heavy data streams, this effectively cuts in half the number of required ACKs, and always gives a receiving host time to consume some data from its buffer and therefore include a larger receive window in its acknowledgments.

Flow control helps TCP protect slow endpoint consumers from being overwhelmed with data, but it does nothing to prevent slow networks and routers from being overwhelmed. Traffic builds up on networks just like it does on highways, with jams getting especially bad at popular routers, much like at popular entrances, exits, and interchanges. To avoid cluttering up networks unnecessarily, TCP implements **congestion control**, which is very similar to the stop light meters found at many highway entrances. To reduce congestion, the TCP module voluntarily limits the

amount of unacknowledged data it will allow in flight. This is similar to what it does for flow control, but instead of setting the limit to a window size dictated by the destination, it calculates the limit itself based on the number of packets that have been acknowledged or dropped. The exact algorithm is implementation dependent, but typically is some sort of additive increase, multiplicative decrease system. That is, when a connection is established, the TCP module sets the congestion avoidance limit to a low multiple of the MSS. Choosing two times the MSS is typical. Then, for every segment acknowledged, it increases the limit by an additional MSS. For an ideal connection, this means that a limit's worth of packets are acknowledged every RTT period, which causes the limit to double in size. However, if a packet is ever dropped, the TCP module quickly cuts the limit in half, suspecting that the drop was due to network congestion. In this way, an equilibrium is eventually reached such that a sender is transmitting as fast as it can without causing so much traffic that packets begin to drop.

TCP can also reduce network congestion by sending out packets as close in size to the MSS as necessary. Because each packet requires a 40-byte header, sending several small segments is much less efficient than coalescing the segments into a larger chunk and sending it when ready. This means the TCP module needs to keep an outgoing buffer to accumulate data that higher layers attempt to send. **Nagle's algorithm** is a set of rules that many TCP implementations use to decide when to accumulate data and when to finally send a segment. Traditionally, if there is already unacknowledged data in flight, it accumulates data until the amount is greater than the MSS or congestion control window, whichever is smaller. At that point it sends the largest segment allowed by those two limits.

> tip
>
> Nagle's algorithm is the bane of players whose games use TCP as a transport layer protocol. Although it decreases bandwidth used, it can significantly increase the delay before data is sent. If a real-time game needs to send small updates to a server, it might be many frames of gameplay before enough updates accumulate to fill an MSS. This can leave players feeling the game is laggy even though it's just Nagle's algorithm at work. For this reason, most TCP implementations provide an option to disable this congestion control feature.

Disconnecting

Shutting down a TCP connection requires a termination request and acknowledgment from each end. When one host has no more data to send, it sends a FIN packet, indicating that it is ready to cease sending data. All data pending in the outdoing buffer, including the FIN packet, will be transmitted and retransmitted until acknowledged. However, the TCP module will accept no new outgoing data from a higher layer. Data can still be received from the other host, though, and all incoming data will be ACK'd. When the other side has no more data to send, it too can send a FIN packet. When a closing host has received a FIN packet from the other

host and an ACK packet in response to its own FIN packet, or a timeout for the ACK has been exceeded, then the TCP module fully shuts down and deletes its connection state.

The Application Layer

At the very top of the TCP/IP layer cake is the application layer, and this is where our multiplayer game code lives. The application layer is also home to many fundamental protocols of the Internet that rely on the transport layer for end-to-end communication, and we will explore some here.

DHCP

Assigning unique IPv4 addresses to each host on a private subnet can be an administrative challenge, especially when laptops and smart phones are introduced into the mix. **Dynamic host configuration protocol** (**DHCP**) solves this problem by allowing a host to request configuration information automatically when it attaches to the network.

Upon connecting to the network, the host creates a DHCPDISCOVER message containing its own MAC address and broadcasts it using UDP to 255.255.255.255:67. Because this goes to every host on the subnet, any DHCP server present will receive the message. The DHCP server, if it has an IP address to offer the client, prepares a DHCPOFFER packet. This packet contains both the offered IP address and the MAC address of the client to be sent the offer. At this point, the client has no IP address assigned, so the server can't directly address a packet to it. Instead, the server broadcasts the packet to the entire subnet on UDP port 68. All DHCP clients receive the packet, and each checks the MAC address in the message to determine if it is the intended recipient. When the correct client receives the message, it reads the offered IP address and decides if it would like to accept the offer. If so, it responds, via broadcast, with a DHCPREQUEST message requesting the offered address. If the offer is still available, the server responds, again via broadcast, with a DHCPACK message. This message both confirms to the client that the IP address is assigned, and conveys any additional network information necessary, such as the subnet mask, router address, and any recommended DNS name servers to use.

The exact format of DHCP packets and extended information on DHCP can be found in RFC 2131, referenced in the "Additional Reading" section.

DNS

Domain name system (**DNS**) protocol enables the translation of domain and subdomain names into IP addresses. When an end user wants to perform a google search, she doesn't need to type 74.125.224.112 into her web browser, but can instead just type www.google.com. To translate the domain name into an IP address, her web browser sends a DNS query to the IP address of the name server which her computer has been configured to use.

A **name server** stores mappings from domain names to IP addresses. For instance, one might store that www.google.com should resolve to the IP address 74.125.224.112. There are many thousands of name servers in use on the Internet, and most are only authoritative for a small subset of the Internet's domains and subdomains. If a name server is queried about a domain for which it is not an authority, it usually has a pointer to a more authoritative name server which it queries in turn. The results of the second query are usually cached so that the next time the name server must answer a query for that domain, it has the answer on hand.

DNS queries and responses are usually sent via UDP on port 53. The format is defined in RFC 1035, referenced in the "Additional Reading" section.

NAT

Until now, every IP address discussed has been publically routable. An IP address qualifies as **publically routable** if any properly configured router on the Internet can set a packet on a route such that the packet eventually arrives at the host with that IP address. This necessitates that any publically routable address be uniquely assigned to a single host. If two or more hosts shared the same IP address, then a packet addressed to one might end up at another. If one of the hosts made a request to a webserver, the response could end up at the alternate host, thoroughly confusing it.

To keep publically routable addresses unique, ICANN and its subsidiaries allocate distinct blocks of IPs to large institutions like megacorporations, universities, and Internet service providers, who can then hand out those addresses to members and customers, ensuring that each address is assigned uniquely.

Because IPv4 supports only a 32-bit address space, there are a mere 4,294,967,296 potential public IP addresses available. Due to the incredible number of networked devices in use today, and the way IP addresses are distributed by ICANN, they have grown scarce. Oftentimes, a network administrator or user may find herself allocated fewer public IP addresses than she has hosts. For instance, as video game developers, we probably each have at least a smartphone, laptop, and gaming console, yet only pay for a single public IP address from our ISP. How annoying would it be if each device required its own dedicated public IP address? Each time we connected a new gadget to the Internet, we would end up having to fight with other users for a new IP address from our ISP, and then probably pay more for it as well.

Luckily, it is possible to connect an entire subnet of hosts to the Internet through a single shared public IP address. This is made possible by **network address translation** or **NAT**. To configure a network for NAT, each host on the network must be assigned a **privately routable** IP address. Table 2.15 lists some IP address blocks which IANA has reserved for private use, guaranteeing that no address from those blocks will ever be assigned as a public IP address. Thus, any user may set up their own private network using privately routable IP addresses,

without checking for uniqueness. Uniqueness between networks is not required because the addresses are not publically routable. That is, no public router on the Internet should have routing information regarding how to reach a private IP address, so it doesn't matter if multiple private networks employ the same private IP addresses internally.

Table 2.15 Private IP Address Blocks

IP Address Range	Subnet
10.0.0.0–10.255.255.255	10.0.0.0/8
172.16.0.0–172.31.255.255	172.16.0.0/12
192.168.0.0–192.168.255.255	192.168.0.0/16

To understand how NAT works, consider the video gamer's home network in Figure 2.17. The game console, smart phone, and laptop all have internally unique private IP addresses, assigned by the owner of the network, without the need to consult any outside service provider. The router also has a private IP address on its internal facing NIC, and it has a publically routable, ISP-assigned IP address on its outward facing NIC. Because the privately addressed NIC is connected to the local network, it is called a **local area network** (**LAN**) port, and because the publically addressable NIC is connected worldwide, it is called the **wide area network** (**WAN**) port.

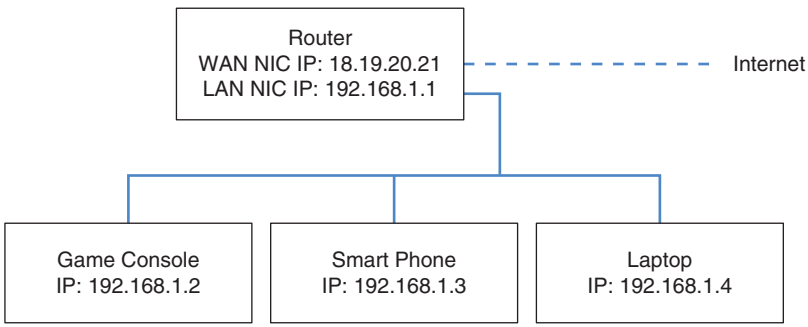

Figure 2.17 Private network behind a NAT

For this example, assume a host at the publically routable IP 12.5.3.2 is running a game server bound to port 200. The game console with private IP 192.168.1.2 is running a game bound to port 100. The game console needs to send a message to the server via UDP, so it builds a datagram as shown in Figure 2.18, with 192.168.1.2:100 as the source, and 12.5.3.2:200 as the destination. Without NAT enabled on the router, the console sends the datagram to the LAN port of the router, which then forwards it from the WAN port to the Internet. The packet eventually arrives at the server. At this point, though, there is a problem. Because the source address on the IP packet is 192.168.1.2, the server is unable to send a packet back in response. Remember that

192.168.1.2 is a private IP address, and thus no public router on the Internet can route to that address. Even if some router did nonsensically have routing information for that IP address, it is unlikely the packet would end up at our game console, as there are many thousands of hosts on the Internet with the private IP address 192.168.1.2.

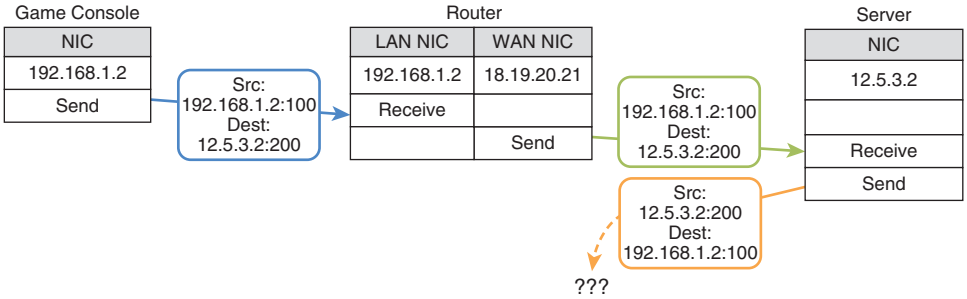

Figure 2.18 Router without NAT

To prevent this issue, the NAT module of the router can actually rewrite the IP packet as it routes it, replacing the private IP address, 192.168.1.2 with the router's own public IP address, 18.19.20.21. That solves part of the problem but not all of it: Rewriting only the IP address creates the situation depicted in Figure 2.19. The server sees the datagram as coming directly from the router's public IP address, so it can send a datagram back to the router successfully. However, the router has no record of who sent the original datagram, so it doesn't know where to direct the response.

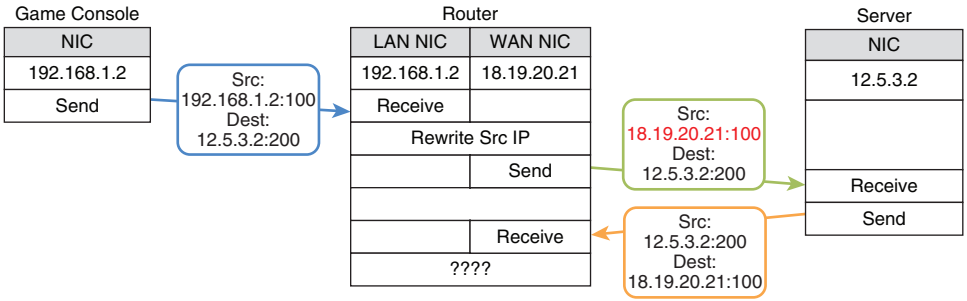

Figure 2.19 NAT router with address rewriting

To be able to return a reply to the proper internal host, the router needs some kind of mechanism to identify the intended internal recipient of an incoming packet. One naïve way is to build a table that records the source IP address of each outgoing packet. Then, when a response is received from an external IP address, the router could look up which internal host

sent a packet to that address and then rewrite the packet to use that internal host's IP address. This would break down, though, if multiple internal hosts sent traffic to the same external host. The router would not be able to identify which incoming traffic is for which internal host.

The solution employed by all modern NAT routers is to violently break the abstraction barrier between the network layer and the transport layer. By rewriting not only the IP addresses in the IP header, but also the port numbers in the transport layer header, the router can create a much more precise mapping and tagging system. It keeps track of these mapping in a **NAT table**. Consider Figure 2.20, which shows the traffic as a packet travels from the game console to the server and a reply returns successfully to the game console.

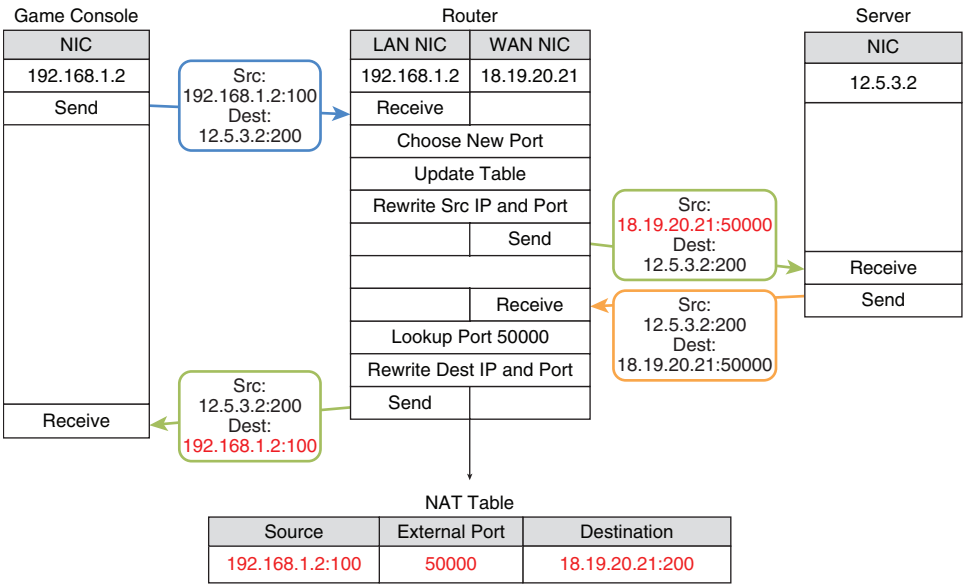

Figure 2.20 NAT router with address and port rewriting

When the game console's outgoing packet reaches the router, the NAT module records both the source IP address and the source port number into a new row in the NAT table. It then picks a random, previously unused port number that is used to identify that source address and source port combination, and writes that number into the same row. It rewrites the packet to use the router's own public IP address and the newly chosen port number. The rewritten packet travels to the server, at which point the server sends a response back, addressed to the router's public IP address and newly chosen port. The NAT module then uses that port number to look up the original source IP address and port. It rewrites the response packet and forwards it to the correct host.

> **note**
>
> For extra security, many routers add the original destination IP address and port to the NAT table entry. This way, when a response packet comes into the router, the NAT module can first look up the table entry using the source port of the packet and it can then make sure the source IP address and port of the response match the destination IP address and port of the original outgoing packet. If they do not match, something fishy is going on, and the packet is dropped instead of forwarded.

NAT Traversal

NAT is a fantastic boon for Internet users, but it can be a terrible headache for multiplayer game developers. Considering how many users have their own private networks at home and use NAT to connect their computers and game consoles to the Internet, it is not uncommon that the situation in Figure 2.21 arises. Player A owns Host A, behind NAT A. She wants to host a multiplayer game server on Host A. She wants her friend, Player B, to connect to her server. Player B uses Host B, behind NAT B. Because of the NAT, Player B has no way to initiate a connection with Host A. If Host B sends a packet to Host A's router in an attempt to connect, there will be no entry in Host A's NAT table, so the packet will simply be dropped.

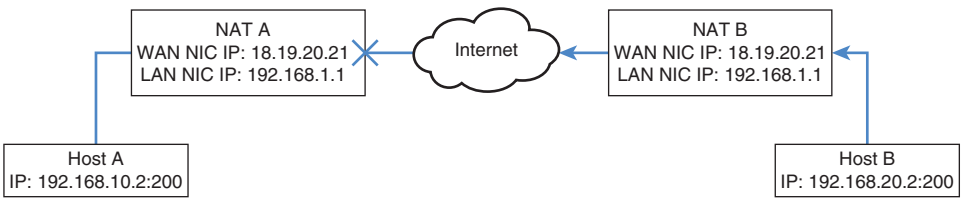

Figure 2.21 Typical user gaming setup

There are a few ways around this problem. One is to require Player A to manually configure port forwarding on her router. This is something that requires a small amount of technical skill and confidence and is not something nice to force players to do. The second way around the problem is much more elegant and much more sneaky. It is known as **simple traversal of UDP through NAT** or **STUN**.

When using STUN, hosts communicate with a third-party host, such as an Xbox Live or PlayStation Network server. That third party tells the hosts how to initiate connections with each other such that the required entries are made in their routers' NAT tables, and they can proceed to communicate directly. Figure 2.22 shows the flow of communication, Figure 2.23

details the packets exchanged, and the NAT tables generated. Let us assume our game runs on UDP port 200 so all communication to and from non-router hosts will be on port 200.

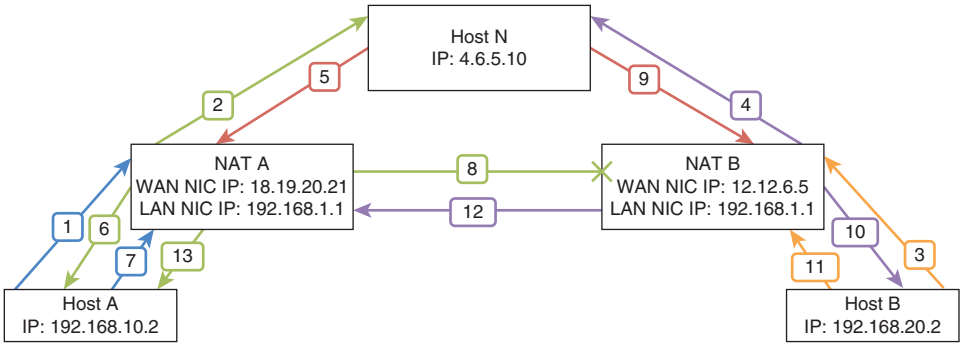

Figure 2.22 STUN data flow

Actions

Packet Number	Packet Sender	Source Address	Destination Address	Packet Receiver	Result
1	Host A	192.168.10.2:200	4.6.5.10:200	NAT A	Make row 1 in NAT A Table, NAT A rewrites packet
2	Host A	18.19.20.21:60000	4.6.5.10:200	Host N	Host N register Host A as game server at 18.19.20.21:60000
3	Host B	192.168.20.2:200	4.6.5.10:200	NAT B	Make row 1 in NAT B Table, NAT B rewrites packet,
4	Host B	12.12.6.5:62000	4.6.5.10:200	Host N	Host N registers Host B as client at 12.12.6.5:62000
5	Host N	4.6.5.10:200	18.19.20.21:60000	NAT A	Matches row 1 in NAT A Table, NAT A rewrites packet
6	NAT A	4.6.5.10:200	192.168.10.2:200	Host A	Host A learns Host B's public address and sends packet
7	Host A	192.168.10.2:200	12.12.6.5:62000	NAT A	Make row 2 in NAT A Table, reusing port from row 1, NAT A rewrites packet
8	Host A	18.19.20.21:60000	12.12.6.5:62000	NAT B	NAT B not expecting packet, Drops it
9	Host N	4.6.5.10:200	12.12.6.5:62000	NAT B	Matches row 1 in NAT B Table, NAT B rewrites packet.
10	NAT B	4.6.5.10:200	192.168.20.2:200	Host B	Host B learns Host A's public address, sends packet
11	Host B	192.168.20.2:200	18.19.20.21:60000	NAT B	Make row 2 in NAT B Table, reusing port from row 1, NAT B rewrites packet
12	NAT B	12.12.6.5:62000	18.19.20.21:60000	NAT A	Matches row 2 in NAT A Table. NAT A rewrites packet.
13	NAT A	12.12.6.5:62000	192.168.10.2:200	Host A	Successful transmission from Host B to Host A

NAT A Table

Row	Source	External Port	Destination
1	192.168.10.2:200	60000	4.6.5.10:200
2	192.168.10.2:200	60000	12.12.6.5:62000

NAT B Table

Row	Source	External Port	Destination
1	192.168.20.2:200	62000	4.6.5.10:200
2	192.168.20.2:200	62000	18.19.20.21:60000

Figure 2.23 STUN packet detail and NAT tables

First, Host A sends a packet from port 200 to the negotiator service at IP 4.6.5.10 (Host N) announcing it would like to be a server. When the packet passes through Router A, Router A makes an entry in its NAT table and rewrites the packet to use its own public IP address as the source and the random number 60000 as a source port. Router A then forwards the packet to Host N. Host N receives the packet and makes note of the fact that Player A, playing on Host A, at address 18.19.20.21:60000 wants to register as a server of a multiplayer game.

Host B then sends a packet to Host N, announcing that Player B would like to connect to Player A's game. When the packet passes through Router B, the NAT table at Router B is updated and the packet is rewritten, similar to how NAT occurred at Router A. The rewritten packet is then forwarded to Host N, who learns from the packet that Host B at 12.12.6.5:62000 would like to connect to Host A.

At this point, Host N knows Router A's public IP address, as well as the destination port which will result in Router A forwarding a packet to Host A. It could send this information to Host B in a reply packet, and request that Host B attempt to connect directly using it. However, recall that some routers check the origin of incoming packets to make sure they are expecting packets from that location. Router A is only expecting a packet from Host N. If Host B tries to connect to Host A at this point, Router A will block the packet because Router A is not expecting any response from Host B.

Luckily, Host N also knows Router B's public IP address and the port number which will cause a packet to be forwarded to Host B. So, it sends this information to Host A. Router A lets this information pass, because its NAT table indicates that Host A is expecting a response from Host N. Host A then sends an outgoing packet to Host B using the connection info received from Host N. This may seem crazy, as it is the server attempting to contact the client, whereas we would expect the reverse. It may seem even crazier, because we know that Router B is not expecting any incoming packets from Host A and will thus not allow the packet through anyway. Why would we waste a packet like that? We do it just to force an entry into Router A's NAT table!

As the packet travels from Host A to Host B, it passes through Router A. Router A sees in the NAT table that Host A's address, 192.168.1.2:200, already maps to external port 60000, so it chooses this port for the outgoing packet. It then makes an additional entry stating that 192.168.1.2:200 has sent traffic to 128.127.126.125:62000. This additional entry is the key. The packet will probably never arrive at Host B, but after this has happened, Host N can reply to Host B, telling it to connect directly to Host A at 18.19.20.21:51243. Host B does so, and when the packet arrives at Router A, Router A sees that it is indeed expecting an incoming packet from 128.127.126.125:62000. It rewrites the packet to be targeting 192.168.1.2:200 and sends it to Host A. From that point on, Hosts A and B can communicate directly by using the public IP address and port number they have exchanged.

note

There are a few more facts about NATs worth mentioning. First, the NAT traversal technique described earlier will not work for all NATs. There are some NATs which do not assign a consistent external port number to an internal host. These are known as **symmetric NATs**. In a symmetric NAT, each outgoing request receives a unique external port, even if originating from a source IP address and port already

in the NAT table. This breaks STUN because Router A will pick a new external port to use when Host A sends its first packet to Host B. When Host B contacts Router A on the original external port that Host A used to reach Host N, it will not match in the NAT table and the packet will be dropped.

Sometimes, less secure symmetric NATs assign external ports in a deterministic order, so clever programs can use **port assignment prediction** to make STUN-like techniques work on symmetric NATs. More secure symmetric NATs use randomized port assignments that cannot easily be predicted.

The STUN method works only for UDP. As described in Chapter 3, "The Berkeley Sockets," TCP uses a different system of port assignment and necessarily transmits data on a port different from the one on which it listens for incoming connections. When TCP is in use, there is a technique called **TCP hole punching** which may work if the NAT router acts in a way which supports it. RFC 5128, referenced in "Additional Reading" gives a good survey of NAT traversal techniques, including TCP hole punching.

Finally, there is yet another popular way to enable traversal of a NAT router. It is called **Internet gateway device protocol** (**IGDP**). This is a protocol that some **Universal Plug and Play** (**UPnP**) routers employ to allow LAN hosts to manually set up mappings between external and internal ports. It is not always supported and less academically interesting so it is not explained here. Its specification is also referenced in the "Additional Reading" section.

Summary

This chapter provided an overview of the inner workings of the Internet. Packet switching allows multiple transmissions to be sent simultaneously over the same line, giving rise to ARPANET and eventually the Internet. The TCP/IP suite, the layer cake that powers the Internet, consists of five layers, each of which provides a data channel for the layer above it.

The physical layer provides the medium along which the signal travels, and is sometimes considered part of the link layer above it. The link layer provides a method of communication between connected hosts. It requires a hardware addressing system so that each host can be uniquely addressed, and determines the MTU, the maximum amount of data which can be transmitted in a single chunk. There are many protocols which can provide the primary link layer services, but this chapter explored Ethernet in great depth, as it is the one most important to game developers.

The network layer, which provides a logical addressing system on top of the link layer's hardware addresses, allows hosts on different link layer networks to communicate. IPv4, the

primary network layer protocol of the day, provides direct and indirect routing systems, and fragments packets too large for the link layer. IPv6, rising in prominence, solves the problem of a limited address space and optimizes several of the biggest bottlenecks in IPv4 data transmission.

The transport layer and its ports provide end-to-end communication between processes on remote hosts. TCP and UDP are the primary protocols in the transport layer, and fundamentally different: UDP is lightweight, connectionless, and unreliable, whereas TCP has a heavier footprint, requires stateful connections, and guarantees reliable, in-order delivery of all data. TCP implements flow control and congestion control mechanisms to decrease packet loss.

At the top of the cake is the application layer, containing DHCP, DNS, and your game code.

To facilitate the creation of private networks with minimal oversight, NAT allows a single public IP address to be shared by an entire network. A drawback of NAT is that it blocks unsolicited incoming connections that a server might desire, but there are techniques such as STUN and TCP hole punching which provide workarounds for this.

This chapter has provided a theoretical basis for the workings of the Internet. This will prove useful in Chapter 3, which covers the functions and data structures used to write code that actually communicates between hosts.

Review Questions

1. List the five layers of the TCP/IP stack and briefly describe each. Which layer is not considered a separate layer in some models?
2. For what is ARP used? How does it work?
3. Explain how a host with multiple NICs (i.e., a router) routes packets between different subnets. Explain how a routing table works.
4. What does MTU stand for? What does it mean? What is the MTU of Ethernet?
5. Explain how packet fragmentation works. Assuming a link layer with an MTU of 400, give the header of a packet which would be fragmented into two fragments, and then give the headers of those fragments.
6. Why is it good to avoid IP fragmentation?
7. Why is it good to send packets that are as large as possible without fragmenting?
8. What is the difference between unreliable and reliable data transfer?
9. Describe the TCP handshake process to establish a connection. What important pieces of data are exchanged?
10. Describe how TCP effects reliable data transfer.
11. What is the difference between a publically routable IP address and a privately routable one?

12. What is NAT? What are some benefits of using a NAT? What are some costs?
13. Explain how a client behind a NAT can send a packet to a publically routable server and receive a response.
14. What is STUN? Why would you need it? How does it work?

Additional Readings

Bell, Gordon. (1980, September). *The Ethernet—A Local Area Network*. Retrieved from http://research.microsoft.com/en-us/um/people/gbell/ethernet_blue_book_1980.pdf. Accessed September 12, 2015.

Braden, R. (Ed). (1989, October). *Requirements for Internet Hosts—Application and Support*. Retrieved from http://tools.ietf.org/html/rfc1123. Accessed September 12, 2015.

Braden, R. (Ed). (1989, October). *Requirements for Internet Hosts—Communication Layers*. Retrieved from http://tools.ietf.org/html/rfc1122. Accessed September 12, 2015.

Cotton, M., L. Eggert, J. Touch, M. Westerlund, and S. Cheshire. (2011, August). *Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry*. Retrieved from http://tools.ietf.org/html/rfc6335. Accessed September 12, 2015.

Deering, S., and R. Hinden. (1998, December). *Internet Protocol, Version 6 (IPv6) Specification*. Retrieved from https://www.ietf.org/rfc/rfc2460.txt. Accessed September 12, 2015.

Drom, R. (1997, March). *Dynamic Host Configuration Protocol*. Retrieved from http://tools.ietf.org/html/rfc2131. Accessed September 12, 2015.

Google IPv6 Statistics. (2014, August 9). Retrieved from https://www.google.com/intl/en/ipv6/statistics.html. Accessed September 12, 2015.

Information Sciences Institute. (1981, September). *Transmission Control Protocol*. Retrieved from http://www.ietf.org/rfc/rfc793.txt. Accessed September 12, 2015.

Internet Gateway Device Protocol. (2010, December). Retrieved from http://upnp.org/specs/gw/igd2/. Accessed September 12, 2015.

Mockapetris, P. (1987, November). *Domain Names—Concepts and Facilities*. Retrieved from http://tools.ietf.org/html/rfc1034. Accessed September 12, 2015.

Mockapetris, P. (1987, November). *Domain Names—Implementation and Specification*. Retrieved from http://tools.ietf.org/html/rfc1035. Accessed September 12, 2015.

Nagle, John. (1984, January 6). *Congestion Control in IP/TCP Internetworks*. Retrieved from http://tools.ietf.org/html/rfc896. Accessed September 12, 2015.

Narten, T., E. Nordmark, W. Simpson, and H. Soliman. (2007, September). *Neighbor Discovery for IP version 6 (IPv6)*. Retrieved from http://tools.ietf.org/html/rfc4861. Accessed September 12, 2015.

Nichols, K., S. Blake, F. Baker, and D. Black. (1998, December). *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. Retrieved from http://tools.ietf.org/html/rfc2474. Accessed September 12, 2015.

Port Number Registry. (2014, September 3). Retrieved from http://www.iana.org/assignments /service-names-port-numbers/service-names-port-numbers.xhtml. Accessed September 12, 2015.

Postel, J., and R. Reynolds. (1988, February). *A Standard for the Transmission of IP Datagrams over IEEE 802 Networks*. Retrieved from http://tools.ietf.org/html/rfc1042. Accessed September 12, 2015.

Ramakrishnan, K., S. Floyd, and D. Black. (September 2001). *The Addition of Explicit Congestion Notification (ECN) to IP.* Retrieved from http://tools.ietf.org/html/rfc3168. Accessed September 12, 2015.

Rekhter, Y., and T. Li. (1993, September). *An Architecture for IP Address Allocation with CIDR*. Retrieved from http://tools.ietf.org/html/rfc1518. Accessed September 12, 2015.

Rosenberg, J., J. Weinberger, C. Huitema, and R. Mahy. (2003, March). *STUN—Simple Traversal of User Datagram Protocol (UDP).* Retrieved from http://tools.ietf.org/html/rfc3489. Accessed September 12, 2015.

Socolofsky, T., and C. Kale. (1991, January). *A TCP/IP Tutorial*. Retrieved from http://tools.ietf.org /html/rfc1180. Accessed September 12, 2015.

This page intentionally left blank

CHAPTER 3

BERKELEY SOCKETS

This chapter introduces the most commonly used networking construct for multiplayer game development, the Berkeley Socket. It presents the most common functions for creating, manipulating, and disposing sockets, discusses differences between platforms, and explores a type-safe, C++ friendly wrapper for socket functionality.

Creating Sockets

Originally released as part of BSD 4.2, the **Berkeley Sockets API** provides a standardized way for processes to interface with various levels of the TCP/IP stack. Since its release, the API has been ported to every major operating system and most popular programming languages, so it is the veritable standard in network programming.

Processes use the API by creating and initializing one or more **sockets**, and then reading data from or writing data to those sockets. To create a socket, use the aptly named `socket` function:

```
SOCKET socket(int af, int type, int protocol);
```

The `af` parameter, standing for address family, indicates the network layer protocol which the socket should employ. Potential values are listed in Table 3.1.

Table 3.1 Address Family Values for Socket Creation

Macro	Meaning
`AF_UNSPEC`	Unspecified
`AF_INET`	Internet Protocol Version 4
`AF_IPX`	Internetwork Packet Exchange: An early network layer protocol popularized by Novell and MS-DOS
`AF_APPLETALK`	Appletalk: An early network suite popularized by apple computer for use with its Apple and Macintosh computers
`AF_INET6`	Internet Protocol Version 6

Most games written these days support IPv4, so your code will most likely use `AF_INET`. As more users switch to IPv6 Internet connections, it becomes more worthwhile to support `AF_INET6` sockets as well.

The `type` parameter indicates the meaning of packets sent and received through the socket. Each transport layer protocol that the socket can use has a corresponding way in which it groups and uses packets. Table 3.2 lists the most commonly supported values for this parameter.

Table 3.2 Type Values for Socket Creation

Macro	Meaning
`SOCK_STREAM`	Packets represent segments of an ordered, reliable stream of data
`SOCK_DGRAM`	Packets represent discrete datagrams
`SOCK_RAW`	Packet headers may be custom crafted by the application layer
`SOCK_SEQPACKET`	Similar to `SOCK_STREAM` but packets may need to be read in their entirety upon receipt

Creating a socket of type SOCK_STREAM informs the operating system that the socket will require a stateful connection. It then allocates the necessary resources to support a reliable, ordered stream of data. This is the appropriate socket type to use when creating a TCP socket. SOCK_DGRAM, on the other hand, provides for no stateful connection and allocates only the minimal resources necessary to send and receive individual datagrams. The socket should make no effort to maintain reliability or ordering of packets. This is the appropriate socket type for a UDP socket.

The protocol parameter indicates the specific protocol that the socket should use to send data. This can include transport layer protocols, or various utility network layer protocols that are part of the Internet protocol suite. Typically, the value passed in as the protocol is copied directly into the protocol field of the IP header for each outgoing packet. This signifies to the receiving operating system how to interpret data wrapped by the packet. Table 3.3 gives typical values for the protocol parameter.

Table 3.3 Protocol Values for Socket Creation

Macro	Required Type	Meaning
IPPROTO_UDP	SOCK_DGRAM	Packets wrap UDP datagrams
IPPROTO_TCP	SOCK_STREAM	Packets wrap TCP segments
IPPROTO_IP / 0	Any	Use the default protocol for the given type

Note that passing 0 as the protocol tells the OS to pick the default implemented protocol for the given socket type. This means you can create an IPv4 UDP socket by calling

```
SOCKET udpSocket = socket(AF_INET, SOCK_DGRAM, 0);
```

You can create a TCP socket by calling

```
SOCKET tcpSocket = socket(AF_INET, SOCK_STREAM, 0);
```

To close a socket, regardless of type, use the closesocket function:

```
int closesocket( SOCKET sock );
```

When disposing of a TCP socket, it is important to do so in a manner that ensures all outgoing and incoming data are transmitted and acknowledged. It is best to first cease transmitting on the socket, then wait for all data to be acknowledged and all incoming data to be read, and then to close the socket.

To cease transmitting or receiving before closing, use the shutdown function:

```
int shutdown(SOCKET sock, int how)
```

For how, pass SD_SEND to cease sending, SD_RECEIVE to cease receiving, or SD_BOTH to cease sending and receiving. Passing SD_SEND will cause a FIN packet to transmit once all data

has been sent, which will notify the other end of the connection it can safely close its socket. That will result in a FIN packet being sent back in response. Once your game receives the FIN packet, it is safe to actually close the socket.

This closes the socket and returns any associated resources to the operating system. Make sure to close all sockets when they are no longer needed.

> **note**
>
> In most cases, the operating system creates the IP layer header and transport layer header for each packet sent out over a socket. However, by creating a socket of type `SOCK_RAW` and protocol 0, you can directly write each of the header values for those two layers. This allows you to set header fields directly which are not normally editable. For instance, you could easily specify a custom TTL for each outgoing packet: That is exactly what the Traceroute utility does. Manually writing the values for various header fields is often the only way to insert illegal values in those fields, which can be particularly useful when fuzz testing your servers, as mentioned in Chapter 10, "Security."
>
> Because raw sockets allow illegal values in header fields, they are a potential security risk, and most operating systems allow the creation of raw sockets only in programs with elevated security credentials.

API Operating System Differences

Although Berkeley Sockets are the standard low-level way to interface with the Internet on various platforms, the API is not perfectly uniform across all operating systems. There are several idiosyncrasies and differences worth understanding before jumping into cross-platform socket development.

The first of these is the data type used to represent the socket itself. The `socket` function as listed earlier returns a result of type `SOCKET`, but this type actually exists only on Windows-based platforms like Windows 10 and Xbox. A little digging into the Windows headers files shows that `SOCKET` is a `typedef` for a `UINT_PTR`. That is, it points to an area of memory that holds state and data about the socket.

Contrariwise, on POSIX-based platforms like Linux, Mac OS X, and PlayStation, a socket is represented by a single `int`. There is no socket data type per se: The socket function returns an integer. This integer represents an index into the operating system's list of open files and sockets. In this way, a socket is very similar to a POSIX file descriptor, and in fact can be passed to many OS functions that take file descriptors. Using sockets in this way limits some of the flexibility provided by the dedicated socket functions, but in some cases provides an easy

path to porting a non-network based process to a network compatible one. One significant drawback of the socket function returning an `int` is the lack of type safety, as the compiler will not balk at code which passes any integral expression (e.g., 5×4) to a function that takes a socket parameter. Several code examples in this chapter address this problem, as it is a general weakness of the Berkeley Socket API on all platforms.

Regardless of whether your platform represents a socket as an `int` or a `SOCKET`, it's worth noting that sockets should always be passed by value to functions in the socket library.

The second major difference between platforms is the header file which contains the declarations for the library. The Windows version of the socket library is known as Winsock2, and thus files which use socket functionality must `#include` the file WinSock2.h. There is an older version of the Winsock library called Winsock, and this version is actually included by default in the overarching Windows.h file used in most Windows programs. The Winsock library is an earlier, limited, less optimized version of the WinSock2 library, but it does contain several basic library functions, such as the `socket` creation one discussed earlier. This creates a name conflict when both Windows.h and WinSock2.h are included in the same translation unit: Multiple declarations for the same functions cause the compiler to choke and spew errors confusing to those who are unaware of this conflict. To avoid this, you must make sure to either `#include` WinSocket2.h before Windows.h, or to `#define` the macro `WIN32_LEAN_AND_MEAN` before including Windows.h. The macro causes the preprocesser to omit, among other things, the inclusion of Winsock from the list of files contained in Windows.h, thus preventing the conflict.

WinSock2.h only contains declarations for the functions and data types directly related to sockets. For tangential functionality, you will have to include other files. For instance, to use address conversion functionality discussed in this chapter, you will also need to include Ws2tcpip.h.

On POSIX platforms, there is only one version of the socket library and it is usually accessed by including the file sys/socket.h. To use IPv4-specific functionality you may also have to include netinet/in.h. To use address conversion functionality, include arpa/inet.h. To perform name resolution you may have to include netdb.h.

Initialization and shutdown of the socket library also differ between platforms. On POSIX platforms, the library is active by default and nothing is required to enable socket functionality. Winsock2, however, requires explicit startup and cleanup and allows the user to specify what version of the library to use. To activate the socket library on Windows, use `WSAStartup`:

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);
```

`wVersionRequested` is a 2-byte word in which the low-order byte specifies the major version and the high-order byte specifies the minor version of the Winsock implementation desired. The highest version supported as of this printing is 2.2, so typically you will pass `MAKEWORD(2, 2)` for this parameter.

`lpWSAData` points to a Windows-specific data structure which the `WSAStartup` function fills in with data about the activated library, including the version of the implementation provided. Typically this will match the version requested and you will not usually need to check this data.

`WSAStartup` returns either a 0, indicating success, or an error code, indicating why the library could not be started up. Note that no Winsock2 functions will work properly unless your process first invokes `WSAStartup` successfully.

To shut down the library, call `WSACleanup`:

```
int WSACleanup();
```

`WSACleanup` takes no parameters and returns an error code. When a process invokes `WSACleanup`, all pending socket operations are terminated and all socket resources are deallocated, so it is a good idea to make sure all sockets are closed and truly unused before shutting down Winsock. `WSAStartup` is reference counted, so you must call `WSACleanup` exactly as many times you called `WSAStartup` to make sure that anything is actually cleaned up.

Error reporting is handled slightly differently between platforms. Most functions on all platforms return –1 in the case of an error. On Windows, you can use the macro `SOCKET_ERROR` instead of the magic number –1. A single –1 does little to reveal the source of the error, though, so Winsock2 provides the function `WSAGetLastError` to fetch an additional code that expands on the cause of the error:

```
int WSAGetLastError();
```

This function returns only the latest error code generated on the currently running thread, so it is important to check it immediately after any socket library function returns a –1. Calling a successive socket function after an error could cause a secondary error due to the initial one. This would change the result returned by `WSAGetLastError` and mask the true cause of the problem.

POSIX-compatible libraries similarly provide a method to retrieve specific error information. However, these use the C standard library global variable `errno` to report specific error codes. To check the value of `errno` from code, you must include the file errno.h. After that, you can read from `errno` like any other variable. Just like the result returned by `WSAGetLastError`, `errno` can change after every function call, so it is important to check it at the first sign of error.

> **tip**
>
> Most platform-independent functions in the socket library use purely lowercase letters, like `socket`. Most Windows-specific Winsock2 functions, however, begin with capital letters, and sometimes the `WSA` prefix, to mark them as nonstandard. When developing for Windows, try to keep capital letter Winsock2 functions isolated from the cross-platform ones so that porting to POSIX platforms will be simpler.

There are additional Winsock2-specific functions that are not supported by the POSIX version of the Berkeley Socket library, just like most POSIX-compatible operating systems have their own platform-specific networking functions in addition to the POSIX standard ones. The standard socket functions provide adequate functionality for a typical multiplayer networked game, so for the rest of this chapter, we will explore only the standard, cross-platform functions. The sample code for this book targets the Windows Operating System but uses Winsock2-specific functions only when necessary, to start up, shut down, and check for errors. The text will call out multiple versions whenever a function differs across platforms.

Socket Address

Every network layer packet requires a source address and a destination address. If the packet wraps transport layer data, it also requires a source port and a destination port. To pass this address information in and out of the socket library, the API provides the `sockaddr` data type:

```
struct sockaddr {
    uint16_t  sa_family;
    char    sa_data[14];
};
```

`sa_family` holds a constant identifying the type of the address. When using this socket address with a socket, the `sa_family` field should match the `af` parameter used to create the socket. `sa_data` is 14 bytes which hold the actual address. The `sa_data` field is a necessarily generic array of bytes because it must be able to hold the address format appropriate for whatever address family is specified. Technically, you could fill in the bytes manually, but this would require knowing the memory layout for various address families. To remedy this, the API provides dedicated data types to help initialize addresses for common address families. Because there were no classes or polymorphic inheritance at the time of the socket API's creation, these data types must be manually cast to the `sockaddr` type when passed into any socket API function that requires an address. To create an address for an IPv4 packet, use the `sockaddr_in` type:

```
struct sockaddr_in {
    short    sin_family;
    uint16_t sin_port;
    struct   in_addr sin_addr;
    char     sin_zero[8];
};
```

`sin_family` overlaps sockaddr's `sa_family` and thus has the same meaning.

`sin_port` holds the 16-bit port section of the address.

`sin_addr` holds the 4-byte IPv4 address. The `in_addr` type varies between socket libraries. On some platforms, it is a simple 4-byte integer. IPv4 addresses are not usually written as 4-byte integers, but instead as 4 individual bytes separated with dots. For this reason, other

platforms provide a structure that wraps a union of structs that can be used to set the address in different formats:

```
struct in_addr {
  union {
    struct {
      uint8_t s_b1,s_b2,s_b3,s_b4;
    } S_un_b;
    struct {
      uint16_t s_w1,s_w2;
    } S_un_w;
    uint32_t S_addr;
  } S_un;
};
```

By setting the `s_b1`, `s_b2`, `s_b3`, and `s_b4` fields of the `S_un_b` struct inside the `S_un` union, you can enter the address in a human readable form.

`sin_zero` is unused and merely exists to pad the size of `sockaddr_in` to match the size of `sockaddr`. For consistency, it should be set to all zeroes.

> **tip**
>
> In general, when instancing any BSD socket struct, it is a wise idea to use `memset` to zero out all its members. This can help prevent cross-platform errors from uninitialized fields that arise when one platform uses fields that another platform ignores.

When setting the IP address as a 4-byte integer, or when setting the port number, it is important to account for that fact the TCP/IP suite and the host computer may use different standards for the ordering of bytes within multibyte numbers. Chapter 4, "Object Serialization," provides an in-depth look at platform-dependent byte ordering, but for now, it is sufficient to know that any multibyte numbers set in a socket address structure must be converted from host byte order to network byte order. To facilitate this, the socket API provides the functions `htons` and `htonl`:

```
uint16_t htons( uint16_t hostshort );
uint32_t htonl( uint32_t hostlong );
```

The `htons` function takes any unsigned, 16-bit integer in the host's native byte order and converts it to the same integer represented in the network's native byte order. The `htonl` function performs the same operation on 32-bit integers.

On platforms where the host byte order and network byte order are the same, these functions will do nothing. When optimizations are turned on, the compiler will recognize this fact and

omit the function calls without generating any extra code. On platforms where the host byte order does not match the network byte order, the returned values will have the same bytes as the input parameters, but their order will be swapped. This means that if you are on such a platform, and you use the debugger to examine the `sa_port` field of a properly initialized `sockaddr_in`, the decimal value represented there will not match that of your intended port. Instead it will be the decimal value of a byte-swapped version of your port.

Sometimes, as in the case of receiving a packet, the socket library fills in the `sockaddr_in` structure for you. When this happens, the `sockadd_in` fields will still be in network byte order, so if you wish to extract them and make sense of them, you should use the functions `ntohs` and `ntohl` to convert the values from network byte order to host byte order:

```
uint16_t ntohs(uint16_t networkshort);
uint32_t ntohl(uint32_t networklong);
```

These two functions work the same way as their host-to-network counterparts.

Putting all these techniques together, Listing 3.1 shows how to create a socket address that represents port 80 at IP address 65.254.248.180.

Listing 3.1 Initializing a `sockaddr_in`

```
sockaddr_in myAddr;
memset(myAddr.sin_zero, 0, sizeof(myAddr.sin_zero));
myAddr.sin_family = AF_INET;
myAddr.sin_port = htons(80);
myAddr.sin_addr.S_un.S_un_b.s_b1 = 65;
myAddr.sin_addr.S_un.S_un_b.s_b2 = 254;
myAddr.sin_addr.S_un.S_un_b.s_b3 = 248;
myAddr.sin_addr.S_un.S_un_b.s_b4 = 180;
```

> **note**
>
> Some platforms add an extra field in the `sockaddr` to store the length of the structure used. This is to enable longer-length `sockaddr` structures in the future. On these platforms, just set the length to the `sizeof` the structure used. For instance, on Mac OS X, initialize a `sockaddr_in` named myAddr by setting `myAddr.sa_len = sizeof(sockaddr_in)`.

Type Safety

Because there was very little consideration for type safety when the initial socket library was created, it can be useful to wrap the basic socket data types and functions with custom object-oriented ones, implemented at the application level. This also helps isolate the socket API from

your game, in case you decide to change out the socket library for some alternative networking library at a later date. In this book, we will be wrapping many structs and functions both as a way to demonstrate proper use of the underlying API and to provide a more type-safe framework on which you can build your own code. Listing 3.2 presents a wrapper for the `sockaddr` structure.

Listing 3.2 Type-Safe `SocketAddress` Class

```
class SocketAddress
{
public:
    SocketAddress(uint32_t inAddress, uint16_t inPort)
    {
        GetAsSockAddrIn()->sin_family = AF_INET;
        GetAsSockAddrIn()->sin_addr.S_un.S_addr = htonl(inAddress);
        GetAsSockAddrIn()->sin_port = htons(inPort);
    }
    SocketAddress(const sockaddr& inSockAddr)
    {
        memcpy(&mSockAddr, &inSockAddr, sizeof( sockaddr) );
    }

    size_t GetSize() const {return sizeof( sockaddr );}

private:
    sockaddr mSockAddr;

    sockaddr_in* GetAsSockAddrIn()
         {return reinterpret_cast<sockaddr_in*>( &mSockAddr );}
};
typedef shared_ptr<SocketAddress> SocketAddressPtr;
```

`SocketAddress` has two constructors. The first takes a 4-byte IPv4 address and port and assigns the value to an internal `sockaddr`. The constructor automatically sets the address family to `AF_INET` because the parameters are only sensible for an IPv4 address. To support IPv6, you could extend this class with another constructor.

The second constructor takes a native `sockaddr` and copies it into the internal `mSockAddr` field. This is useful when the network API returns a `sockaddr` and you wish to wrap it with a `SocketAddress`.

The `GetSize` helper method of `SocketAddress` keeps the code clean when dealing with functions that need the size of the `sockaddr`.

Finally, the shared pointer type to a socket address ensures there is an easy way to share socket addresses without having to worry about cleaning up the memory. At the moment, `SocketAddress` wraps very little, but it provides a good base on which to add more functionality as future examples require it.

Initializing `sockaddr` from a String

It takes a bit of work just to feed an IP address and port into a socket address, especially considering that the address information will probably be passed to your program as a string in a config file or on a commandline. If you do have a string to turn into a `sockaddr`, you can skip this work by using the `inet_pton` function on POSIX-compatible systems or the `InetPton` function on Windows.

```
int inet_pton(int af, const char* src, void* dst);
int InetPton(int af, const PCTSTR src  void* dst);
```

Both functions take an address family, either `AF_INET` or `AF_INET6`, and convert a string representation of an IP address into an `in_addr` representation. `src` should point to a null terminated character string containing the address in dotted notation and `dst` should point to the `sin_addr` field of the `sockaddr` to be set. The functions return 1 on success, 0 if the source string is malformed, or –1 if some other system error occurred. Listing 3.3 shows how to initialize a `sockaddr` using one of these presentation-to-network functions.

Listing 3.3 Initializing `sockaddr` with `InetPton`

```
sockaddr_in myAddr;
myAddr.sin_family = AF_INET;
myAddr.sin_port = htons( 80 );
InetPton(AF_INET, "65.254.248.180", &myAddr.sin_addr);
```

Although `inet_pton` converts a human readable string to a binary IP address, the string must be an IP address. It cannot be a domain name, as no DNS lookup is performed. If you wish to perform a simple DNS query to translate a domain name into an IP address, use `getaddrinfo`:

```
int getaddrinfo(const char *hostname, const char *servname, const addrinfo
*hints, addrinfo **res);
```

`hostname` should be a null terminated string holding the name of the domain to look up. For instance, "live-shore-986.herokuapp.com."

`servname` should be a null terminated string containing either a port number, or the name of a service which maps to a port number. For instance, you can send either "80" or "http" to request a `sockaddr_in` containing port 80.

`hints` should be a pointer to an `addrinfo` structure containing information about the results you wish to receive. You can specify a desired address family or other requirement using this parameter, or you can just pass `nullptr` to get all matching results.

Finally, `res` should be a pointer to a variable that the function will set to point to the head of a linked list of newly allocated `addrinfo` structures. Each `addrinfo` represents a section of the response from the DNS server:

```
struct addrinfo {
  int           ai_flags;
  int           ai_family;
  int           ai_socktype;
  int           ai_protocol;
  size_t        ai_addrlen;
  char          *ai_canonname;
  sockaddr      *ai_addr;
  addrinfo      *ai_next;
}
```

`ai_flags`, `ai_socktype`, and `ai_protocol` are used to request certain types of responses when you pass an `addrinfo` into `getaddrinfo` as a hint. They can be ignored in the response.

`ai_family` identifies the address family to which this `addrinfo` pertains. A value of `AF_INET` indicates an IPv4 address and a value of `AF_INET6` indicates an IPv6 address.

`ai_addrlen` gives the size of the `sockaddr` pointed to by `ai_addr`.

`ai_canonname` holds the canonical name of the resolved hostname, if the `AI_CANONNAME` flag is set in the `ai_flags` field of the `addrinfo` passed as `hints` in the original call.

`ai_addr` contains a `sockaddr` of the given address family, which addresses the host specified by the `hostname` and the port specified by the `servname` parameters of the original call.

`ai_next` points to the next `addrinfo` in the linked list. Because a domain name can map to multiple IPv4 and IPv6 addresses, you should iterate through the linked list until you find a `sockaddr` that suits your needs. Alternatively, you can specify the `ai_family` in the `addrinfo` passed as a hint and you will receive results for only the desired family. The final `addrinfo` in the list will have `nullptr` as its `ai_next` to indicate it is the tail.

Because `getaddrinfo` allocates one or more `addrinfo` structures, you should call `freeaddrinfo` to release the memory once you have copied the desired `sockaddr` out of the linked list:

```
void freeaddrinfo(addrinfo* ai);
```

In `ai`, pass only the very first `addrinfo` returned by `getaddrinfo`. The function will walk the linked list freeing up all `addrinfo` nodes and all associated buffers.

To resolve a host name into an IP address, `getaddrinfo` creates a DNS protocol packet and sends it using either UDP or TCP to one of the DNS servers configured in the operating system. It then waits for a response, parses the response, constructs the linked list of `addrinfo` structures, and returns this to the caller. Because this process is dependent on sending information to and receiving information from a remote host, it can take a significant amount of time. Sometimes this is on the order of milliseconds, but more often it is on the order of

seconds. `getaddrinfo` has no provisions for asynchronous operation built in, so it will block the calling thread until it receives a response. This can cause an undesirable experience for the user, so if you need to resolve hostnames into IP addresses, you should consider calling `getaddrinfo` on a thread other than the main thread of your game. On Windows, you can alternatively call the Windows-specific `GetAddrInfoEx` function, which does allow for asynchronous operation without manually creating a different thread.

You can encapsulate the functionality of `getaddrinfo` nicely in the `SocketAddressFactory` given in Listing 3.4.

Listing 3.4 Name Resolution Using the `SocketAddressFactory`

```
class SocketAddressFactory
{
public:
    static SocketAddressPtr CreateIPv4FromString(const string& inString)
    {
        auto pos = inString.find_last_of(':');
        string host, service;
        if(pos != string::npos)
        {
            host = inString.substr(0, pos);
            service = inString.substr(pos + 1);
        }
        else
        {
            host = inString;
            //use default port...
            service = "0";
        }
        addrinfo hint;
        memset(&hint, 0, sizeof(hint));
        hint.ai_family = AF_INET;

        addrinfo* result;
        int error = getaddrinfo(host.c_str(), service.c_str(),
                                &hint, &result);
        if(error != 0 && result != nullptr)
        {
            freeaddrinfo(result);
            return nullptr;
        }

        while(!result->ai_addr && result->ai_next)
        {
            result = result->ai_next;
        }
```

```
        if(!result->ai_addr)
        {
            freeaddrinfo(result);
            return nullptr;
        }
        auto toRet = std::make_shared< SocketAddress >(*result->ai_addr);

        freeaddrinfo(result);

        return toRet;
    }
};
```

`SocketAddressFactory` has a single static method to create a `SocketAddress` from a string representing a host name and port. The function returns a `SocketAddressPtr` so that it has the option of returning `nullptr` if anything goes wrong with the name conversion. This is a nice alternative to making a `SocketAddress` constructor do the conversion because, without requiring exception handling, it makes sure there is never an incorrectly initialized `SocketAddress` in existence: If `CreateIPv4FromString` returns a non-null pointer, then it is guaranteed to be a valid `SocketAddress`.

The method first separates the port from the name by searching for a colon. It then creates a hint `addrinfo` to ensure that only IPv4 results are returned. It feeds all this into `getaddrinfo` and iterates through the resulting list until a non-null address is found. It copies this address into a new `SocketAddress` using the appropriate constructor and then frees the linked list. If anything goes wrong, it returns null.

Binding a Socket

The process of notifying the operating system that a socket will use a specific address and transport layer port is known as **binding**. To manually bind a socket to an address and port, use the `bind` function:

```
int bind(SOCKET sock, const sockaddr *address, int address_len);
```

`sock` is the socket to bind, previously created by the `socket` function.

`address` is the address to which the socket should bind. Note that this has nothing to do with the address to which the socket will send packets. You can think of this as defining the return address of any packets sent from the socket. It may seem curious that you must specify a return address at all, since any packets sent from this host are clearly coming from this host's address. However, remember that a host can have multiple network interfaces, and each interface can have its own IP address.

Passing a specific address to bind allows you to determine which interface the socket should use. This is especially useful for hosts that serve as routers or bridges between networks,

as their different interfaces may be connected to entirely different sets of computers. For multiplayer game purposes, it is usually not important to specify a network interface, and in fact often desirable to bind a given port for *all* available network interfaces and IP addresses that the host has. To do this, you can assign the macro `INADDR_ANY` to the `sin_addr` field of the `sockaddr_in` that you pass to bind.

`address_len` should contain the size of the `sockaddr` passed as the `address`.

`bind` returns 0 on success, or –1 in case of an error.

Binding a socket to a `sockaddr` serves two functions. First, it tells the OS that this socket should be the target recipient for any incoming packet with a destination matching the socket's bound address and port. Second, it dictates the source address and port that the socket library should use when creating network and transport layer headers for packets sent out from the socket.

Typically you can only bind a single socket to a given address and port. `bind` will return an error if you try to bind to an address and port already in use. In that case, you can repeatedly try binding different ports until you find one that is not in use. To automate this process, you can specify 0 for the port to bind. This tells the library to find an unused port and bind that.

A socket must be bound before it can be used to transmit or receive data. Because of this, if a process attempts to send data using an unbound socket, the network library will automatically bind that socket to an available port. Therefore, the only reason to manually call `bind` is to specify the bound address and port. This is necessary when building a server that must listen for packets on a publically announced address and port, but usually not necessary for a client. A client can automatically bind to any available port: When it sends its first packet to the server, the packet will contain the automatically chosen source address and port, and the server can use those to address any return packets correctly.

UDP Sockets

You can send data on a UDP socket as soon as the socket is created. If it is not bound, the network module will find a free port in the dynamic port range and automatically bind it. To send data, use the `sendto` function:

```
int sendto(SOCKET sock, const char *buf, int len, int flags,
const sockaddr *to, int tolen);
```

`sock` is the socket from which the datagram should send. If the socket is unbound, the library will automatically bind it to an available port. The socket's bound address and port will be used as the source address in the headers of the outgoing packet.

`buf` is a pointer to the starting address of the data to send. It does not have to be an actual `char*`. It can be any type of data as long as it is cast appropriately to a `char*`. Because of this,

`void*` would have been a more appropriate data type for this parameter, so it is useful to think of it that way.

`len` is the length of data to send. Technically the maximum length of a UDP datagram including its 8-byte header is 65535 bytes, because the length field in the header holds only 16 bits. However, remember that the link layer's MTU determines the largest packet that can be sent without fragmentation. The MTU for Ethernet is 1500 bytes, but this must include not only the game's payload data, but also multiple headers and potentially any packet wrappers. As a game programmer trying to avoid fragmentation, a good rule of thumb is to avoid sending datagrams with data larger than 1300 bytes.

`flags` is a bitwise OR collection of flags controlling the sending of data. For most game play code, this should be 0.

`to` is the `sockaddr` of the intended recipient. This `sockaddr`'s address family must match the one used to create the socket. The address and port from the `to` parameter are copied into the IP header and UDP header as the destination IP address and destination port.

`len` is the length of the `sockaddr` passed as the `to` parameter. For IPv4, just pass `sizeof(sockaddr_in)`.

If the operation is successful, `sendto` returns the length of the data queued to send. Otherwise it returns –1. Note that a nonzero return value doesn't actually mean the datagram was sent, just that it was successfully queued to be sent.

Receiving data on a UDP socket is a simple matter of using the `recvfrom` function:

```
int recvfrom(SOCKET sock, char *buf, int len, int flags, sockaddr *from,
int *fromlen);
```

`sock` is the socket to query for data. By default, if no unread datagrams have been sent to the socket, the thread will block until a datagram arrives.

`buf` is the buffer into which the received datagram should be copied. By default, once a datagram has been copied into a buffer through a `recvfrom` call, the socket library no longer keeps a copy of it.

`len` should specify the maximum number of bytes the `buf` parameter can hold. To avoid a buffer overflow error, `recvfrom` will never copy more than this number of bytes into `buf`. Any remaining bytes in the incoming datagram will be lost for good, so make sure to always use a receiving buffer as large as the largest datagram you expect to receive.

`flags` is a bitwise OR collection of flags controlling the receiving of data. For most game play code, this should be 0. One occasionally useful flag is the `MSG_PEEK` flag. This will copy a received datagram into the `buf` parameter without removing any data from the input queue. That way, the next `recvfrom` call, potentially with a larger buffer, can refetch the same datagram.

`from` should be a pointer to a `sockaddr` structure that the `recvfrom` function can fill in with the sender's address and port. Note that this structure does not need to be initialized ahead of time with any address information. It is a common misconception that one can specifically request a packet from a particular address by filling in this parameter, but no such thing is possible. Instead, datagrams are delivered to the `recvfrom` function in the order received, and the `from` variable is set to the corresponding source address for each datagram.

`fromlen` should point to an integer holding the length of the `sockaddr` passed in as `from`. `recvfrom` may reduce this value if it doesn't need all the space to copy the source address.

After successful execution, `recvfrom` returns the number of bytes that were copied into `buf`. If there was an error, it returns –1.

Type-Safe UDP Sockets

Listing 3.5 shows the type-safe `UDPSocket` class, capable of binding an address and sending and receiving datagrams.

Listing 3.5 Type-Safe `UDPSocket` Class

```
class UDPSocket
{
public:
    ~UDPSocket();
    int Bind(const SocketAddress& inToAddress);
    int SendTo(const void* inData, int inLen, const SocketAddress& inTo);
    int ReceiveFrom(void* inBuffer, int inLen, SocketAddress& outFrom);
private:
    friend class SocketUtil;
    UDPSocket(SOCKET inSocket) : mSocket(inSocket) {}
    SOCKET mSocket;
};
typedef shared_ptr<UDPSocket> UDPSocketPtr;

int UDPSocket::Bind(const SocketAddress& inBindAddress)
{
    int err = bind(mSocket, &inBindAddress.mSockAddr,
           inBindAddress.GetSize());
    if(err != 0)
    {
        SocketUtil::ReportError(L"UDPSocket::Bind");
        return SocketUtil::GetLastError();
    }
    return NO_ERROR;
}
```

```
int UDPSocket::SendTo(const void* inData, int inLen,
                      const SocketAddress& inTo)
{
    int byteSentCount = sendto( mSocket,
                                static_cast<const char*>( inData),
                                inLen,
                                0, &inTo.mSockAddr, inTo.GetSize());
    if(byteSentCount >= 0)
    {
        return byteSentCount;
    }
    else
    {
        //return error as negative number
        SocketUtil::ReportError(L"UDPSocket::SendTo");
        return -SocketUtil::GetLastError();
    }
}

int UDPSocket::ReceiveFrom(void* inBuffer, int inLen,
                           SocketAddress& outFrom)
{
    int fromLength = outFromAddress.GetSize();
    int readByteCount = recvfrom(mSocket,
                                 static_cast<char*>(inBuffer),
                                 inMaxLength,
                                 0, &outFromAddress.mSockAddr,
                                 &fromLength);
    if(readByteCount >= 0)
    {
        return readByteCount;
    }
    else
    {
        SocketUtil::ReportError(L"UDPSocket::ReceiveFrom");
        return -SocketUtil::GetLastError();
    }
}

UDPSocket::~UDPSocket()
{
    closesocket(mSocket);
}
```

The `UDPSocket` class has three main methods: `Bind`, `SendTo`, and `ReceiveFrom`. Each makes use of the `SocketAddress` class previously defined. To make this possible, `SocketAddress` must declare `UDPSocket` a friend class so that the methods can access the private `sockaddr` member variable. Treating `SocketAddress` this way makes sure no code outside of this socket wrapper module can edit `sockaddr` directly, which reduces dependencies and prevents potential errors.

A nice benefit of the object-oriented wrapper is the ability to create destructors. In this case, `~UDPSocket` automatically closes the internally wrapped socket to prevent sockets from leaking.

The `UDPSocket` code in Listing 3.5 introduces a dependency on the `SocketUtil` class for reporting errors. Isolating error reporting code this way makes it easy to change error handling behavior and cleanly wraps the fact that some platforms take their errors from `WASGetLastError` and some from `errno`.

The code does not provide a way to create a `UDPSocket` from scratch. The only constructor on `UDPSocket` is private. Similarly to the `SocketAddressFactory` pattern, this is so that there is no way to create a `UDPSocket` with an invalid `mSocket` inside it. Instead, the `SocketUtil::CreateUDPSocket` function in Listing 3.6 will create a `UDPSocket` only after the underlying `socket` call succeeds.

Listing 3.6 Creating a UDP Socket

```
enum SocketAddressFamily
{
    INET = AF_INET,
    INET6 = AF_INET6
};
UDPSocketPtr SocketUtil::CreateUDPSocket(SocketAddressFamily inFamily)
{
    SOCKET s = socket(inFamily, SOCK_DGRAM, IPPROTO_UDP);
    if(s != INVALID_SOCKET)
    {
        return UDPSocketPtr(new UDPSocket(s));
    }
    else
    {
        ReportError(L"SocketUtil::CreateUDPSocket");
        return nullptr;
    }
}
```

TCP Sockets

UDP is stateless, connectionless, and unreliable, so it needs only a single socket per host to send and receive datagrams. TCP, on the other hand is reliable, and requires a connection to be established between two hosts before data transmission can take place. In addition, it must maintain state to resend dropped packets and it has to store that state somewhere. In the Berkeley Socket API, the socket itself stores the connection state. This means a host needs an additional, unique socket for each TCP connection it maintains.

TCP requires a three-stage handshake to initiate a connection between a client and a server. For the server to receive the initial stage of the handshake, it must first create a socket, bind it to a designated port, and then listen for any incoming handshakes. Once it has created and bound the socket using `socket` and `bind`, it begins listening using the `listen` function:

```
int listen(SOCKET sock, int backlog);
```

`sock` is the socket to set into listen mode. Each time a socket in listen mode receives the first stage of a TCP handshake, it stores the request until the owning process makes a call to accept the connection and continue the handshake.

`backlog` is the maximum number of incoming connections that should be allowed to queue up. Once the maximum number of handshakes are pending, any further incoming connection is dropped. Pass `SOMAXCONN` to use the default backlog value.

The function returns 0 on success, or –1 in case of error.

To accept an incoming connection and continue the TCP handshake, call `accept`:

```
SOCKET accept(SOCKET sock, sockaddr* addr, int* addrlen);
```

`sock` is the listening socket on which an incoming connection should be accepted.

`addr` is a pointer to a `sockaddr` structure that will be filled in with the address of the remote host requesting the connection. Similarly to the address passed into `recvfrom`, this `sockaddr` does not need to be initialized and it does not control which connection is accepted. It merely receives the address of the accepted connection.

`addrlen` should be a pointer to the size in bytes of the `addr` buffer. It will be updated by `accept` with the size of the address actually written.

If `accept` succeeds, it creates and returns a new socket which can be used to communicate with the remote host. This new socket is bound to the same port as the listening socket. When the OS receives a packet destined for the bound port, it uses the source address and source port to determine which socket should receive the packet: Remember that TCP requires a host to have a unique socket for each remote host to which it is connected.

The new socket returned by `accept` is associated with the remote host which initiated the connection. It stores the remote host's address and port, and tracks all outgoing packets so they can be resent if dropped. It is also the only socket which can communicate with the remote host: A process should never attempt to send data to a remote host using the initial socket in listen mode. That will fail, as the listen socket is never connected to anything. It only acts as a dispatcher to help create new sockets in response to incoming connection requests.

By default, if there are no connections ready to accept, `accept` will block the calling thread until an incoming connection is received or the attempt times out.

The process of listening for and accepting connections is an asymmetrical one. Only the passive server needs a listen socket. A client wishing to initiate a connection should instead create a socket and use the `connect` function to begin the handshake process with a remote server:

```
int connect(SOCKET sock, const sockaddr *addr, int addrlen);
```

`sock` is the socket on which to connect.

`addr` is a pointer to the address of the desired remote host.

`addrlen` is the length of the `addr` parameter.

On success, `connect` returns 0. If there is an error, it returns –1.

Calling `connect` initiates the TCP handshake by sending the initial SYN packet to a target host. If that host has a listen socket bound to the appropriate port, it can proceed with the handshake by calling `accept`. By default, a call to `connect` will block the calling thread until the connection is accepted or the attempt times out.

Sending and Receiving via Connected Sockets

A connected TCP socket stores the remote host's address information. Because of this, a process does not need to pass an address with each call to transmit data. Instead of using `sendto`, send data through a connected TCP socket using the `send` function:

```
int send(SOCKET sock, const char *buf, int len, int flags)
```

`sock` is the socket on which the data should be sent.

`buf` is a buffer of data to write to the stream. Note that unlike for UDP, `buf` is not a datagram and not guaranteed to be transferred as a single data unit. Instead, the data is just appended to the socket's outgoing buffer, and transferred sometime in the future at the socket library's whim. If the Nagle algorithm is active, as described in Chapter 2, this may not happen until an MSS worth of data has accumulated.

`len` is the number of bytes to transmit. Unlike for UDP, there is no reason to keep this value below the expected MTU of the link layer. As long as there is room in the socket's send buffer, the network library will append the data and then send it out in whatever chunk sizes it deems appropriate.

`flags` is a bitwise OR collection of flags controlling the sending of data. For most game play code, this should be 0.

If the `send` call succeeds, it returns the amount of data sent. This may be less than the `len` parameter, if the socket's send buffer had some space free but not enough to hold the entire `buf`. If there is no room at all, then by default the calling thread will block until the call times out,

or enough packets are sent for there to be room. If there is an error, `send` returns –1. Note that a nonzero return value does not imply any data was sent, just that data was queued to be sent.

To receive data on a connected TCP socket, call `recv`:

```
int recv(SOCKET sock, char *buf, int len, int flags);
```

`sock` is the socket to check for data.

`buf` is the buffer into which the data should be copied. The copied data is removed from the socket's receive buffer.

`len` is the maximum amount of received data to copy into `buf`.

`flags` is a bitwise OR collection of flags controlling the receiving of data. Any flags usable with `recvfrom` are also usable with `recv`. For most game play code, this should be 0.

If the `recv` call is successful, it returns the number of bytes received. This will be less than or equal to `len`. It is not possible to predict the amount of data received based on remote calls to `send`: The network library on the remote host accumulates the data and sends out segments sized as it sees fit. If `recv` returns zero when `len` is nonzero, it means the other side of the connection has sent a `FIN` packet and promises to send no more data. If `recv` returns zero when `len` is zero, it means there is data on the socket ready to be read. With many sockets in use, this can be a handy way to check for the presence of data without having to dedicate a buffer to the task. Once `recv` has indicated there is data available, you can reserve a buffer and then call `recv` again, passing the buffer and a nonzero `len`.

If there is an error, `recv` returns –1.

By default, if there is no data in the socket's receive buffer, `recv` blocks the calling thread until the next segment in the stream arrives or the call times out.

> **note**
>
> You can actually use `sendto` and `recvfrom` on a connected socket if you want. However, the address parameters will be ignored and this can be confusing. Similarly, on some platforms it is possible to call `connect` on a UDP socket to store a remote host's address and port in the socket's connection data. This doesn't establish a reliable connection, but it does allow the use of `send` to transmit data to the stored host without having to specify the address each time. It also causes the socket to discard incoming datagrams from any host other than the stored one.

Type-Safe TCP Sockets

The type-safe `TCPSocket` looks similar to `UDPSocket`, but with additional connection-oriented functionality wrapped. Listing 3.7 gives the implementation.

Listing 3.7 Type-Safe `TCPSocket` Class

```
class TCPSocket
{
public:
    ~TCPSocket();
    int                     Connect(const SocketAddress& inAddress);
    int                     Bind(const SocktetAddress& inToAddress);
    int                     Listen(int inBackLog = 32);
    shared_ptr< TCPSocket > Accept(SocketAddress& inFromAddress);
    int                     Send(const void* inData, int inLen);
    int                     Receive(void* inBuffer, int inLen);
private:
    friend class SocketUtil;
    TCPSocket(SOCKET inSocket) : mSocket(inSocket) {}
    SOCKET mSocket;
};
typedef shared_ptr<TCPSocket> TCPSocketPtr;

int TCPSocket::Connect(const SocketAddress& inAddress)
{
    int err = connect(mSocket, &inAddress.mSockAddr, inAddress.GetSize());
    if(err < 0)
    {
        SocketUtil::ReportError(L"TCPSocket::Connect");
        return -SocketUtil::GetLastError();
    }
    return NO_ERROR;
}
int TCPSocket::Listen(int inBackLog)
{
    int err = listen(mSocket, inBackLog);
    if(err < 0)
    {
        SocketUtil::ReportError(L"TCPSocket::Listen");
        return -SocketUtil::GetLastError();
    }
    return NO_ERROR;
}

TCPSocketPtr TCPSocket::Accept(SocketAddress& inFromAddress)
{
    int length = inFromAddress.GetSize();
    SOCKET newSocket = accept(mSocket, &inFromAddress.mSockAddr, &length);

    if(newSocket != INVALID_SOCKET)
    {
        return TCPSocketPtr(new TCPSocket( newSocket));
    }
```

```
    else
    {
        SocketUtil::ReportError(L"TCPSocket::Accept");
        return nullptr;
    }
}

int TCPSocket::Send(const void* inData, int inLen)
{
    int bytesSentCount = send(mSocket,
                              static_cast<const char*>(inData ),
                              inLen, 0);
    if(bytesSentCount < 0 )
    {
        SocketUtil::ReportError(L"TCPSocket::Send");
        return -SocketUtil::GetLastError();
    }
    return bytesSentCount;
}

int TCPSocket::Receive(void* inData, int inLen)
{
    int bytesReceivedCount = recv(mSocket,
                                  static_cast<char*>(inData), inLen, 0);
    if(bytesReceivedCount < 0)
    {
        SocketUtil::ReportError(L"TCPSocket::Receive");
        return -SocketUtil::GetLastError();
    }
    return bytesReceivedCount;
}
```

`TCPSocket` contains the TCP-specific methods: `Send`, `Receive`, `Connect`, `Listen`, and `Accept`. `Bind` and the destructor are no different from the `UDPSocket` versions, so they are not shown. `Accept` returns a `TCPSocketPtr` to ensure the new socket closes automatically when no longer referenced. `Send` and `Receive` do not require addresses because they automatically use the address stored in the connected socket.

To enable creation of a `TCPSocket`, you must add a `CreateTCPSocket` function to `SocketUtils`.

Blocking and Non-Blocking I/O

Receiving from a socket is typically a blocking operation. If there is no data ready to be received, the thread will block until data comes in. This is undesirable if you are polling for packets on the main thread. Sending, accepting, and connecting can also block if the socket is

not ready to perform the operation. This raises issues for a real-time application, like a game, that needs to check for incoming data without reducing the frame rate. Imagine a game server with TCP connections to five clients. If the server calls `recv` on one of its sockets to check for new data from the corresponding client, the server's thread will pause until that client sends some data. This prevents the server from checking on its other sockets, accepting new connections on its listen socket, and running the game simulation. Clearly a game cannot ship that way. Luckily there are three common ways to work around this issue: multithreading, non-blocking I/O, and the `select` function.

Multithreading

One way to work around the problem of blocking I/O is to put each potentially blocking call on its own thread. In the example mentioned earlier, the server would need at least seven threads total: one for each client connection, one for the listen socket, and one or more for the simulation. Figure 3.1 shows the process.

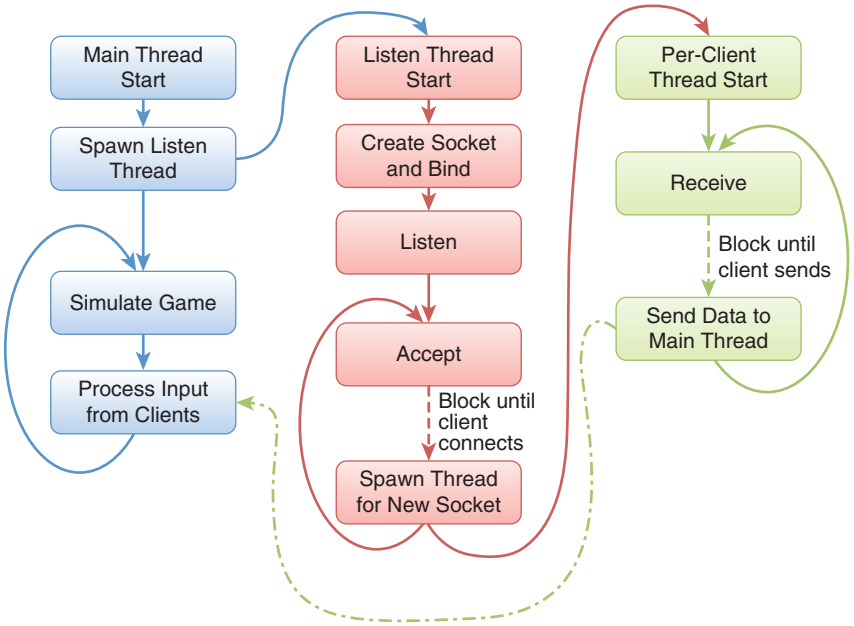

Figure 3.1 Multithreading process

At startup, the listen thread creates a socket, binds it, calls `listen`, and then calls `accept`. The accept call blocks until a client tries to connect. When a client does connect, the `accept` call returns a new socket. The server process spawns a new thread for this socket, which loops

calling `recv`. The `recv` call blocks until the client sends data. When the client sends data, the `recv` call unblocks and the unblocked thread uses some callback mechanism to send the new client data to the main thread before looping back and calling `recv` again. Meanwhile, the listen socket keeps blocking while accepting new connections, and the main thread runs the simulation.

This works, but has the drawback of requiring one thread per client, which does not scale well as the number of clients grows. It also can get hard to manage, as all client data comes in on parallel threads and needs to be passed to the simulation in a safe manner. Finally, if the simulation thread tries to send data on a socket at the same time the receive thread is receiving on that socket, it will still block the simulation. These are not insurmountable problems, but there are simpler alternatives.

Non-Blocking I/O

By default, sockets operate in blocking mode, as previously mentioned. However, sockets also support **non-blocking** mode. When a socket in non-blocking mode is asked to perform an operation that would otherwise require blocking, it instead returns immediately, with a result of –1. It also sets the system error code, `errno` or `WSAGetLastError`, to return a value of EAGAIN or WASEWOULDBLOCK, respectively. This code signifies that the previous socket action would have blocked and was aborted without taking place. The calling process can then react accordingly.

To set a socket into non-blocking mode on Windows, use the `ioctlsocket` function:

```
int ioctlsocket(SOCKET sock, long cmd, u_long *argp);
```

`sock` is the socket to place in non-blocking mode.

`cmd` is the socket parameter to control. In this case, pass `FIONBIO`.

`argp` is the value to set for the parameter. Any nonzero value will enable non-blocking mode, and zero will disable it.

On a POSIX-compatible operating system, use the `fcntl` function:

```
int fcntl(int sock, int cmd, . . .);
```

`sock` is the socket to place in non-blocking mode.

`cmd` is the command to issue to the socket. On newer POSIX systems, you must first use `F_GETFL` to fetch the flags currently associated with the socket, bitwise OR them with the constant `O_NONBLOCK`, and then use the `F_SETFL` command to update the flags on the socket. Listing 3.8 shows how to add a method to enable non-blocking mode on the `UDPSocket`.

Listing 3.8 Enabling Non-Blocking Mode for a Type-Safe Socket

```
int UDPSocket::SetNonBlockingMode(bool inShouldBeNonBlocking)
{
#if _WIN32
    u_long arg = inShouldBeNonBlocking ? 1 : 0;
    int result = ioctlsocket(mSocket, FIONBIO, &arg);
#else
    int flags = fcntl(mSocket, F_GETFL, 0);
    flags = inShouldBeNonBlocking ?
            (flags | O_NONBLOCK):(flags & ~O_NONBLOCK);
    fcntl(mSocket, F_SETFL, flags);
#endif

    if(result == SOCKET_ERROR)
    {
        SocketUtil::ReportError(L"UDPSocket::SetNonBlockingMode");
        return SocketUtil::GetLastError();
    }
    else
    {
        return NO_ERROR;
    }
}
```

When a socket is in non-blocking mode, it is safe to call any usually blocking function and know that it will return immediately if it cannot complete without blocking. A typical game loop using a non-blocking socket might look something like Listing 3.9.

Listing 3.9 Game Loop Using a Non-Blocking Socket

```
void  DoGameLoop()
{
    UDPSocketPtr mySock = SocketUtil::CreateUDPSocket(INET);
    mySock->SetNonBlockingMode(true);

    while(gIsGameRunning)
    {
        char data[1500];
        SocketAddress socketAddress;

        int bytesReceived = mySock->ReceiveFrom(data, sizeof(data),
                socketAddress);
        if(bytesReceived> 0)
        {
            ProcessReceivedData(data, bytesReceived, socketAddress);
        }
        DoGameFrame();
    }
}
```

With the socket set to non-blocking mode, the game can check in each frame to see if any data is ready to be received. If there is data, the game processes the first pending datagram. If there is none, the game immediately moves on to the rest of the frame without waiting. If you want to process more than just the first datagram, you can add a loop which reads pending datagrams until it has read a maximum number, or there are no more present. It is important to limit the number of datagrams read per frame. If you do not, a malicious client could send a slew of single-byte datagrams faster than the server can process them, effectively halting the server from simulating the game.

Select

Polling non-blocking sockets each frame is a simple and straightforward way to check for incoming data without blocking a thread. However, when the number of sockets to poll is large, this can become inefficient. As an alternative, the socket library provides a way to check many sockets at once, and take action as soon as any one of them becomes ready. To do this, use the `select` function:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
const timeval *timeout);
```

On POSIX platforms, `nfds` should be the socket identifier of the highest numbered socket to check. On POSIX, each socket is just an integer, so this is simply the maximum of all sockets passed in to this function. On Windows, where sockets are represented by pointers instead of integers, this parameter does nothing and can be ignored.

`readfds` is a pointer to a collection of sockets, known as an `fd_set`, which should contain sockets to check for readability. Information on how to construct an `fd_set` follows. When a packet arrives for a socket in the `readfds` set, `select` returns control to the calling thread as soon as it is able to. It first removes all sockets from the set that have not received a packet. Thus, when `select` returns, a read from any socket still in the `readfds` set is guaranteed not to block. Pass `nullptr` for `readfds` to skip checking any sockets for readability.

`writefds` is a pointer to an `fd_set` filled with sockets to check for writability. When `select` returns, any sockets that remain in the `writefds` are guaranteed to be writable without causing the calling thread to block. Pass `nullptr` for `writefds` to skip checking any sockets for writability. Typically a socket will block on writing only when its outgoing send buffer is too full of data.

`exceptfds` is a pointer to an `fd_set` filled with sockets to check for errors. When `select` returns, any sockets that remain in `exceptfds` have had errors occur. Pass `nullptr` for `exceptfds` to skip checking any sockets for errors.

`timeout` is a pointer to the maximum time to wait before timing out. If the timeout expires before any socket in the `readfds` becomes readable, any socket in the `writefds` becomes

writable, or any socket in the `exceptfds` experiences an error, all the sets are emptied and `select` returns control to the calling thread. Pass `nullptr` for `timeout` to never time out.

`select` returns the number of sockets which remain in `readfds`, `writefds`, and `exceptfds` after its execution completes. In the case of a timeout, this value is 0.

To initialize an empty `fd_set`, declare one on the stack and zero it out with the `FD_ZERO` macro:

```
fd_set myReadSet;
FD_ZERO(&myReadSet);
```

To add a socket to a set, use the `FD_SET` macro:

```
FD_SET(mySocket, &myReadSet);
```

To check if a socket is in a set after `select` returns, use the `FD_ISSET` macro:

```
FD_ISSET(mySocket, &myReadSet);
```

`select` is not a function of a single socket, so it does not fit as a method of the type-safe socket. It belongs more correctly as a utility method in the `SocketUtils` class. Listing 3.10 shows a `Select` function to work with the type-safe `TCPSocket`.

Listing 3.10 Using `select` with a Type-Safe `TCPSocket`

```
fd_set* SocketUtil::FillSetFromVector(fd_set& outSet,
                                      const vector<TCPSocketPtr>*
                                      inSockets)
{
    if(inSockets)
    {
        FD_ZERO(&outSet);
        for(const TCPSocketPtr& socket : *inSockets)
        {
            FD_SET(socket->mSocket, &outSet);
        }
        return &outSet;
    }
    else
    {
        return nullptr;
    }
}

void SocketUtil::FillVectorFromSet(vector<TCPSocketPtr>* outSockets,
                                   const vector<TCPSocketPtr>*
                                   inSockets,
                                   const fd_set& inSet)
{
```

```
    if(inSockets && outSockets)
    {
        outSockets->clear();
        for(const TCPSocketPtr& socket : *inSockets)
        {
            if(FD_ISSET(socket->mSocket, &inSet))
            {
                outSockets->push_back(socket);
            }
        }
    }
}

int SocketUtil::Select(const vector<TCPSocketPtr>* inReadSet,
                       vector<TCPSocketPtr>* outReadSet,
                       const vector<TCPSocketPtr>* inWriteSet,
                       vector<TCPSocketPtr>* outWriteSet,
                       const vector<TCPSocketPtr>* inExceptSet,
                       vector<TCPSocketPtr>* outExceptSet)
{
    //build up some sets from our vectors
    fd_set read, write, except;

    fd_set *readPtr = FillSetFromVector(read, inReadSet);
    fd_set *writePtr = FillSetFromVector(read, inWriteSet);
    fd_set *exceptPtr = FillSetFromVector(read, inExceptSet);

    int toRet = select(0, readPtr, writePtr, exceptPtr, nullptr);

    if(toRet > 0)
    {
        FillVectorFromSet(outReadSet, inReadSet, read);
        FillVectorFromSet(outWriteSet, inWriteSet, write);
        FillVectorFromSet(outExceptSet, inExceptSet, except);
    }
    return toRet;
}
```

The helper functions `FillSetFromVector` and `FillVectorFromSet` convert between a vector of sockets and an `fd_set`. They allow null to be passed for the vector to support cases where the user would pass null for the `fd_set`. This can be mildly inefficient but is probably not an issue compared to the time required to block on sockets. For slightly better performance, wrap `fd_set` with a C++ data type that provides a good way of iterating through any sockets that remain after the `select` call returns. Keep all relevant sockets in an instance of that data type and remember to pass a duplicate of it to `select` so that `select` does not alter the original set.

Using this `Select` function, Listing 3.11 shows how to set up a simple TCP server loop to listen for and accept new clients while receiving data from old clients. This could run either on the main thread or on a single dedicated thread.

Listing 3.11 Running a TCP Server Loop

```
void DoTCPLoop()
{
    TCPSocketPtr listenSocket = SocketUtil::CreateTCPSocket(INET);
    SocketAddress receivingAddres(INADDR_ANY, 48000);
    if( listenSocket->Bind(receivingAddres ) != NO_ERROR)
    {
        return;
    }
    vector<TCPSocketPtr> readBlockSockets;
    readBlockSockets.push_back(listenSocket);

    vector<TCPSocketPtr> readableSockets;

    while(gIsGameRunning)
    {
        if(SocketUtil::Select(&readBlockSockets, &readableSockets,
                              nullptr, nullptr,
                              nullptr, nullptr))
        {
            //we got a packet—loop through the set ones...
            for(const TCPSocketPtr& socket : readableSockets)
            {
                if(socket == listenSocket)
                {
                    //it's the listen socket, accept a new connection
                    SocketAddress newClientAddress;
                    auto newSocket = listenSocket->Accept(newClientAddress);
                    readBlockSockets.push_back(newSocket);
                    ProcessNewClient(newSocket, newClientAddress);
                }
                else
                {
                    //it's a regular socket—process the data...
                    char segment[GOOD_SEGMENT_SIZE];
                    int dataReceived =
                      socket->Receive( segment, GOOD_SEGMENT_SIZE );
                    if(dataReceived > 0)
                    {
                        ProcessDataFromClient(socket, segment,
                                              dataReceived);
```

```
                    }
                }
            }
        }
    }
}
```

The routine begins by creating a listen socket and adding it into the list of sockets to check for readability. Then it loops until the application requests it do otherwise. The loop uses `Select` to block until a packet comes in on any socket in the `readBlockSockets` vector. When a packet does come in, `Select` ensures that `readableSockets` contains only sockets that have incoming data. The function then loops over each socket `Select` has identified as readable. If the socket is the listen socket, it means a remote host has called `Connect`. The function accepts the connection, adds the new socket to `readBlockSockets`, and notifies the application via `ProcessNewClient`. If the socket is not a listen socket, however, the function calls `Receive` to obtain a chunk of the newly arrived data and passes it to the application via `ProcessDataFromClient`.

> **note**
>
> There are other ways to handle incoming data on multiple sockets, but they are platform specific and less commonly used. On Windows, I/O completion ports are a viable choice when supporting many thousands of concurrent connections. More on I/O completion ports can be found in the "Additional Reading" section.

Additional Socket Options

Various configuration options control the sending and receiving behavior of the sockets. To set these values for these options, call `setsockopt`:

```
int setsockopt(SOCKET sock, int level, int optname, const char *optval, int
optlen);
```

`sock` is the socket to configure.

`level` and `optname` describe the option to be set. `level` is an integer identifying the level at which the option is defined and `optname` defines the option.

`optval` is a pointer to the value to set for the option.

`optlen` is the length of the data. For instance, if the particular option takes an integer, `optlen` should be 4.

`setsockopt` returns 0 if successful or –1 if an error occurs.

Table 3.4 lists some useful options available at the SOL_SOCKET level.

Table 3.4 SOL_SOCKET Options

Macro	Value Type (Windows/POSIX)	Description
SO_RCVBUF	int	Specifies the buffer space this socket allocates for received packets. Incoming data accumulates in the receive buffer until the owning process calls recv or recvfrom to receive it. Remember that TCP bandwidth is limited by the receive window's size, which can never be larger than the receive buffer of the receiving socket. Thus, controlling this value can have a significant impact on bandwidth.
SO_REUSEADDR	BOOL/int	Specifies that the network layer should allow this socket to bind an IP address and port already bound by another socket. This is useful for debugging or packet-sniffing applications. Some operating systems require the calling process to have elevated privileges.
SO_RECVTIMEO	DWORD/timeval	Specifies the time (in milliseconds on Windows) after which a blocking call to receive should time out and return.
SO_SNDBUF	int	Specifies the buffer space this socket allocates for outgoing packets. Outgoing bandwidth is limited based on the link layer. If the process sends data faster than the link layer can accommodate, the socket stores it in its send buffer. Sockets using reliable protocols, like TCP, also use the send buffer to store outgoing data until it is acknowledged by the receiver. When the send buffer is full, calls to send and sendto block until there is room.
SO_SNDTIMEO	DWORD/timeval	Specifies the time (in milliseconds on Windows) after which a blocking call to send should time out and return.
SO_KEEPALIVE	BOOL/int	Valid only for sockets using connection-oriented protocols, like TCP; this option specifies that the socket should automatically send periodic keep alive packets to the other end of the connection. If these packets are not acknowledged, the socket raises an error state, and the next time the process attempts to send data using the socket, it is notified that the connection has been lost. This is not only useful for detecting dropped connections, but also for maintaining connections through firewalls and NATs that might time out otherwise.

Table 3.5 describes the `TCP_NODELAY` option available at the `IPPROTO_TCP` level. This option is only settable on TCP sockets.

Table 3.5 `IPPROTO_TCP` Options

Macro	Value Type (Windows/POSIX)	Description
`TCP_NODELAY`	`BOOL/int`	Specifies whether the Nagle algorithm should be ignored for this socket. Setting this to true will decrease the delay between the process requesting data to be sent and the actual sending of that data. However, it may increase network congestion as a result. For more on the Nagle algorithm, see Chapter 2, "The Internet."

Summary

The Berkeley Socket is the most commonly used construct for transmitting data over the Internet. While the library interface differs across platforms, the core fundamentals are the same.

The core address data type is the `sockaddr`, and it can represent addresses for a variety of network layer protocols. Use it any time it is necessary to specify a destination or source address.

UDP sockets are connectionless and stateless. Create them with a call to `socket` and send datagrams on them with `sendto`. To receive UDP packets on a UDP socket, you must first use `bind` to reserve a port from the operating system, and then `recvfrom` to retrieve incoming data.

TCP sockets are stateful and must connect before they can transmit data. To initiate a connection, call `connect`. To listen for incoming connections, call `listen`. When a connection comes in on a listening socket, call `accept` to create a new socket as the local endpoint of the connection. Send data on connected sockets using `send` and receive it using `recv`.

Socket operations can block the calling thread, creating problems for real-time applications. To prevent this, either make potentially blocking calls on non–real-time threads, set sockets to non-blocking mode, or use the `select` function.

Configure socket options using `setsockopt` to customize socket behavior. Once created and configured, sockets provide the communication pathway that makes networked gaming possible. Chapter 4, "Object Serialization" will begin to deal with the challenge of making the best use of that pathway.

Review Questions

1. What are some differences between POSIX-compatible socket libraries and the Windows implementation?
2. To what two TCP/IP layers does the socket enable access?

3. Explain how and why a TCP server creates a unique socket for each connecting client.
4. Explain how to bind a socket to a port and what it signifies.
5. Update `SocketAddress` and `SocketAddressFactory` to support IPv6 addresses.
6. Update `SocketUtils` to support creation of a TCP socket.
7. Implement a chat server that uses TCP to allow a single host to connect and relays messages back and forth.
8. Add support for multiple clients to the chat server. Use non-blocking sockets on the client and `select` on the server.
9. Explain how to adjust the maximum size of the TCP receive window.

Additional Readings

Information Sciences Institute. (1981, September). *Transmission Control Protocol*. Retrieved from http://www.ietf.org/rfc/rfc793. Accessed September 12, 2015.

I/O Completion Ports. Retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx. Accessed September 12, 2015.

Porting Socket Applications to WinSock. Retrieved from http://msdn.microsoft.com/en-us/library/ms740096.aspx. Accessed September 12, 2015.

Stevens, W. Richard, Bill Fennerl, and Andrew Rudoff. (2003, November 24) *Unix Network Programming Volume 1: The Sockets Networking API, 3rd ed*. Addison-Wesley.

WinSock2 Reference. Retrieved from http://msdn.microsoft.com/en-us/library/windows/desktop/ms740673%28v=vs.85%29.aspx. Accessed September 12, 2015.

This page intentionally left blank

CHAPTER 4

OBJECT SERIALIZATION

To transmit objects between networked instances of a multiplayer game, the game must format the data for those objects such that it can be sent by a transport layer protocol. This chapter discusses the need for and uses of a robust serialization system. It explores ways to handle the issues of self-referential data, compression, and easily maintainable code, while working within the runtime performance requirements of a real-time simulation.

The Need for Serialization

Serialization refers to the act of converting an object from its random access format in memory into a linear series of bits. These bits can be stored on disk or sent across a network and later restored to their original format. Assume that in the *Robo Cat* game, a player's `RoboCat` is represented by the following code:

```
class RoboCat: public GameObject
{
public:
    RoboCat(): mHealth(10), mMeowCount(3) {}

private:
    int32_t mHealth;
    int32_t mMeowCount;
};
```

As mentioned in Chapter 3, "Berkeley Sockets," the Berkeley Socket API uses the `send` and `sentdo` functions to send data from one host to another. Each of these functions takes a parameter which points to the data to transmit. Therefore, without writing any special serialization code, the naïve way to send and receive a `RoboCat` from one host to another would look something like this:

```
void NaivelySendRoboCat(int inSocket, const RoboCat* inRoboCat)
{
    send(nSocket,
        reinterpret_cast<const char*>(inRoboCat),
        sizeof(RoboCat), 0 );
}

void NaivelyReceiveRoboCat(int inSocket, RoboCat* outRoboCat)
{
    recv(inSocket,
          reinterpret_cast<char*>(outRoboCat),
          sizeof(RoboCat), 0);
}
```

`NaivelySendRoboCat` casts the `RoboCat` to a `char*` so that it can pass it to `send`. For the length of the buffer, it sends the size of the `RoboCat` class, which in this case is eight. The receiving function again casts the `RoboCat` to a `char*`, this time so that it can receive directly into the data structure. Assuming a TCP connection using the sockets exists between two hosts, the following process will send the state of a `RoboCat` between those hosts:

1. Call `NaivelySendRoboCat` on the source host, passing in the `RoboCat` to be sent.
2. On the destination host, create or find an existing `RoboCat` object that should receive the state.
3. On the destination host, call `NaivelyReceiveRoboCat`, passing in a pointer to the `RoboCat` object chosen in Step 2.

Chapter 5, "Object Replication" deals in depth with Step 2, explaining how and when to find or create a destination `RoboCat`. For now, assume that a system is in place to locate or spawn the target `RoboCat` on the receiving host.

Once the transfer procedure completes, assuming the hosts are running on identical hardware platforms, the state from the source `RoboCat` successfully copies into the destination `RoboCat`. The memory layout for a sample `RoboCat`, as displayed in Table 4.1, demonstrates why the naïve send and receive functions are effective for a class that is this simple.

The `RoboCat` on the destination host has an `mHealth` of 10 and an `mMeowCount` of 3, as set by the `RoboCat` constructor. The `RoboCat` on the source host has lost half its health, leaving it at 5, and has used up one of its meows, due to whatever game logic has run on that host. Because `mHealth` and `mMeowCount` are primitive data types, the naïve send and receive works correctly, and the `RoboCat` on the destination host ends up with the proper values.

Table 4.1 Sample `RoboCat` in Memory

Address	Field	Source Value	Destination Initial Value	Destination Final Value
Bytes 0–3	`mHealth`	0x00000005	0x0000000A	0x00000005
Bytes 4–7	`mMeowCount`	0x00000002	0x00000003	0x00000002

However, objects representing key elements of a game are rarely as simple as the `RoboCat` in Table 4.1. Code for a more likely version of `RoboCat` presents challenges that cause the naïve process to break down, introducing the need for a more robust serialization system:

```
class RoboCat: public GameObject
{
public:
    RoboCat(): mHealth(10), mMeowCount(3),
                mHomeBase(0)
    {
        mName[0] = '\0';
    }
    virtual void Update();

    void Write(OutputMemoryStream& inStream) const;
    void Read(InputMemoryStream& inStream);

private:
    int32_t                 mHealth;
    int32_t                 mMeowCount;
    GameObject*             mHomeBase;
    char                    mName[128];
    std::vector<int32_t>    mMiceIndices;

};
```

These additions to `RoboCat` create complications that must be considered when serializing. Table 4.2 shows the memory layout before and then after the transfer.

Table 4.2 A Complicated `RoboCat` in Memory

Address	Field	Source Value	Destination Initial Value	Destination Final Value
Bytes 0–3	`vTablePtr`	0x0A131400	0x0B325080	0x0A131400
Bytes 4–7	`mHealth`	0x00000005	0x0000000A	0x00000005
Bytes 8–11	`mMeowCount`	0x00000002	0x00000003	0x00000002
Bytes 12–15	`mHomeBase`	0x0D124008	0x00000000	0x0D124008
Bytes 16–143	`mName`	"Fuzzy\0"	"\0"	"Fuzzy\0"
Bytes 144–167	`mMiceIndices`	??????	??????	??????

The first 4 bytes of `RoboCat` are now a virtual function table pointer. This assumes compilation for a 32-bit architecture—on a 64-bit system this would be the first 8 bytes. Now that `RoboCat` has the virtual method `RoboCat::Update()`, each `RoboCat` instance needs to store a pointer to the table that contains the locations of the virtual method implementations for `RoboCat`. This causes a problem for the naïve serialization implementation because the location of that table can be different for each instance of the process. In this case, receiving replicated state into the destination `RoboCat` replaces the correct virtual function table pointer with the value 0x0B325080. After that, invoking the `Update` method on the destination `RoboCat` would at best result in a memory access exception and at worst result in the invocation of random code.

The virtual function table pointer is not the only pointer overwritten in this instance. Copying the `mHomeBase` pointer from one process to another provides a similarly nonsensical result. Pointers, by their nature, refer to a location in a particular process's memory space. It is not safe to blindly copy a pointer field from one process to another process and hope that the pointer references relevant data in the destination process. Robust replication code must either copy the referenced data and set the field to point to the copy or find an existing version of the data in the destination process and set the field to point there. The section "Referenced Data" later in this chapter discusses these techniques further.

Another issue evident in the naïve serialization of the `RoboCat` is the mandatory copying of all 128 bytes of the `mName` field. Although the array holds up to 128 characters, it may sometimes hold fewer, as it does in the sample `RoboCat` with `mName` equal to "Fuzzy." To fulfill the multiplayer game programmer's mandate of optimized bandwidth usage, a good serialization system should avoid serializing unnecessary data when possible. In this case, that requires the

system to understand that the `mName` field is a null-terminated c string and to only serialize the characters up to and including the null termination. This is one of several techniques for compressing runtime data during serialization, more of which are discussed in detail later in this chapter in the section "Compression."

The final serialization issue illustrated in the new version of `RoboCat` occurs when copying the `std::vector<int32_t> mMiceIndices`. The internals of the STL's `vector` class are not specified by the C++ standard, and thus it is not clear whether it is safe to naïvely copy the field's memory from one process to another. In all likelihood, it is not: There are probably one or more pointers inside the `vector` data structure referencing the `vector`'s elements, and there may be initialization code that must be run once these pointers are set up. It is almost certain that naïve serialization would fail to copy the `vector` properly. In fact, it should be assumed that naïve serialization would fail when copying any black box data structure: Because it is not specified what's inside the structure, it is not safe to copy it bit for bit. Properly handling the serialization of complex data structures is addressed throughout this chapter.

The three problems enumerated earlier suggest that instead of sending a single blob of `RoboCat` data to the socket, each field should be serialized individually to ensure correctness and efficiency. It is possible to create one packet per field, calling a unique send function for each field's data, but this would cause chaos for the network connection, wasting scads of bandwidth for all the unnecessary packet headers. Instead, it is better to gather up all the relevant data into a buffer and then send that buffer as a representation of the object. To facilitate this process, we introduce the concept of the stream.

Streams

In computer science, a **stream** refers to a data structure that encapsulates an ordered set of data elements and allows the user to either read or write data into the set.

A stream can be an **output stream**, **input stream**, or both. An output stream acts as an output sink for user data, allowing the user of the stream to insert elements sequentially, but not to read them back. Contrariwise, an input stream acts as a source of data, allowing the user to extract elements sequentially, but does not expose functionality for inserting them. When a stream is both an input and output stream, it exposes methods for inserting and extracting data elements, potentially concurrently.

Often, a stream is an interface to some other data structure or computing resource. For instance, a **file output stream** could wrap a file that has been opened for writing, providing a simple method of sequentially storing different data types to disk. A **network stream** could wrap a socket, providing a wrapper for the `send()` and `recv()` functions, tailored for specific data types relevant to the user.

Memory Streams

A **memory stream** wraps a memory buffer. Typically, this is a buffer dynamically allocated on the heap. The **output memory stream** has methods for writing sequentially into the buffer, as well as an accessor that provides read access to the buffer itself. By calling the buffer accessor, a user can take all data written into the stream at once and pass it to another system, such as the `send` function of a socket. Listing 4.1 shows an implementation of an output memory stream.

Listing 4.1 Output Memory Stream

```
class OutputMemoryStream
{
public:
    OutputMemoryStream():
    mBuffer(nullptr), mHead(0), mCapacity(0)
    {ReallocBuffer(32);}
    ~OutputMemoryStream()    {std::free(mBuffer);}

    //get a pointer to the data in the stream
    const     char*     GetBufferPtr()    const    {return mBuffer;}
              uint32_t  GetLength()       const    {return mHead;}

    void      Write(const void* inData, size_t inByteCount);
    void      Write(uint32_t inData) {Write(&inData, sizeof( inData));}
    void      Write(int32_t inData) {Write(&inData, sizeof( inData));}

private:
    void        ReallocBuffer(uint32_t inNewLength);

    char*       mBuffer;
    uint32_t    mHead;
    uint32_t    mCapacity;
};

void OutputMemoryStream::ReallocBuffer(uint32_t inNewLength)
{
    mBuffer = static_cast<char*>(std::realloc( mBuffer, inNewLength));
    //handle realloc failure
    //...
    mCapacity = inNewLength;
}

void OutputMemoryStream::Write(const void* inData,
                               size_t inByteCount)
{
    //make sure we have space...
    uint32_t resultHead = mHead + static_cast<uint32_t>(inByteCount);
    if(resultHead > mCapacity)
    {
```

```
        ReallocBuffer(std::max( mCapacity * 2, resultHead));
    }

    //copy into buffer at head
    std::memcpy(mBuffer + mHead, inData, inByteCount);

    //increment head for next write
    mHead = resultHead;
}
```

The `Write(const void* inData, size_t inByteCount)` method is the primary way to send data to the stream. The overloads of the `Write` method take specific data types so that the byte count does not always need to be sent as a parameter. To be more complete, a templated `Write` method could allow all data types, but it would need to prevent nonprimitive types from being serialized: Remember that nonprimitive types require special serialization. A static assert with type traits provides one way to safely template the `Write` method:

```
template<typename T> void Write(T inData)
{
    static_assert(std::is_arithmetic<T>::value ||
                  std::is_enum<T>::value,
                  "Generic Write only supports primitive data types");
    Write(&inData, sizeof(inData));
}
```

Regardless of the method chosen, building a helper function to automatically select byte count helps to prevent errors by reducing the chance that a user will pass the incorrect byte count for a data type.

Whenever there is not enough capacity in the `mBuffer` to hold new data being written, the buffer automatically expands to the maximum of either double the current capacity or to the amount necessary to contain the write. This is a common memory expansion technique, and the multiple can be adjusted to suit a specific purpose.

warning

Although the `GetBufferPtr` function provides a read-only pointer to the stream's internal buffer, the stream retains ownership of the buffer. This means the pointer is invalid once the stream is deallocated. If a use case calls for the pointer returned by `GetBufferPtr` to persist once the stream is deallocated, the buffer could be implemented as `std::shared_ptr<std::vector<uint8_t> >`, but this is left as an exercise at the end of the chapter.

Using the output memory stream, it is possible to implement more robust `RoboCat` send functions:

```
void RoboCat::Write(OutputMemoryStream& inStream) const
{
    inStream.Write(mHealth);
    inStream.Write(mMeowCount);
    //no solution for mHomeBase yet
    inStream.Write(mName, 128);
    //no solution for mMiceIndices yet
}

void SendRoboCat(int inSocket, const RoboCat* inRoboCat)
{
    OutputMemoryStream stream;
    inRoboCat->Write(stream);
    send(inSocket, stream.GetBufferPtr(),
          stream.GetLength(), 0);
}
```

Adding a `Write` method to the `RoboCat` itself allows access to private internal fields and abstracts the task of serialization away from the task of sending data over the network. It also allows a caller to write a `RoboCat` instance as one of many elements inserted into the stream. This proves useful when replicating multiple objects, as described in Chapter 5.

Receiving the `RoboCat` at the destination host requires a corresponding input memory stream and `RoboCat::Read` method, as shown in Listing 4.2.

Listing 4.2 Input Memory Stream

```
class InputMemoryStream
{
public:
    InputMemoryStream(char* inBuffer, uint32_t inByteCount):
    mCapacity(inByteCount), mHead(0),
    {}

    ~InputMemoryStream()    {std::free( mBuffer);}

    uint32_t GetRemainingDataSize() const {return mCapacity - mHead;}

    void     Read(void* outData, uint32_t inByteCount);
    void     Read(uint32_t& outData) {Read(&outData, sizeof(outData));}
    void     Read(int32_t& outData) {Read(&outData, sizeof(outData));}

private:
    char*       mBuffer;
    uint32_t    mHead;
    uint32_t    mCapacity;
};
```

```
void RoboCat::Read(InputMemoryStream& inStream)
{
    inStream.Read(mHealth);
    inStream.Read(mMeowCount);
    //no solution for mHomeBase yet
    inStream.Read(mName, 128);
    //no solution for mMiceIndices
}

const uint32_t kMaxPacketSize = 1470;

void ReceiveRoboCat(int inSocket, RoboCat* outRoboCat)
{
    char* temporaryBuffer =
        static_cast<char*>(std::malloc(kMaxPacketSize));
    size_t receivedByteCount =
        recv(inSocket, temporaryBuffer, kMaxPacketSize, 0);

    if(receivedByteCount > 0)
    {
        InputMemoryStream stream(temporaryBuffer,
                                 static_cast<uint32_t> (receivedByteCount));
        outRoboCat->Read(stream);
    }
    else
    {
        std::free(temporaryBuffer);
    }
}
```

After `ReceiveRoboCat` creates a temporary buffer and fills it by calling `recv` to read pending data from the socket, it passes ownership of the buffer to the input memory stream. From there, the stream's user can extract data elements in the order in which they were written. The `RoboCat::Read` method then does just this, setting the proper fields on the `RoboCat`.

tip

When using this paradigm in a complete game, you would not want to allocate the memory for the stream each time a packet arrives, as memory allocation can be slow. Instead you would have a stream of maximum size preallocated. Each time a packet comes in, you would receive directly into that preallocated stream's buffer, process the packet by reading out of the stream, and then reset the `mHead` to 0 so that the stream is ready to be received into when the next packet arrives.

In this case, it would also be useful to add functionality to the `MemoryInputStream` to allow it to manage its own memory. A constructor that takes only a max capacity could allocate the stream's `mBuffer`, and then an accessor that returns the `mBuffer` would allow the buffer to be passed directly to `recv`.

This stream functionality solves the first of the serialization issues: It provides a simple way to create a buffer, fill the buffer with values from individual fields of a source object, send that buffer to a remote host, extract the values in order, and insert them into the appropriate fields of a destination object. Additionally, the process does not interfere with any areas of the destination object that should not be changed, such as the virtual function table pointer.

Endian Compatibility

Not all CPUs store the bytes of a multibyte number in the same order. The order in which bytes are stored on a platform is referred to as the platform's **endianness**, with platforms being either **little-endian** or **big-endian**. Little-endian platforms store multibyte numbers with their low-order bytes at the lowest memory address. For instance, an integer containing the value 0x12345678 with an address of 0x01000000 would be stored in memory as shown in Figure 4.1.

Value	0x78	0x56	0x34	0x12
Address	0x01000000	0x01000001	0x01000002	0x01000003

Figure 4.1 Little-endian 0x12345678

The least significant byte, the 0x78, is first in memory. This is the "littlest" part of the number, and why the arrangement strategy is called "little" endian. Platforms that use this strategy include Intel's x86, x64, and Apple's iOS hardware.

Big-endian, alternatively, stores the most significant byte in the lowest memory address. The same number would be stored at the same address as shown in Figure 4.2.

Value	0x12	0x34	0x56	0x78
Address	0x01000000	0x01000001	0x01000002	0x01000003

Figure 4.2 Big-endian 0x12345678

Platforms that use this strategy include the Xbox 360, the PlayStation 3, and IBM's PowerPC architecture.

> **tip**
>
> Endian order is usually irrelevant when programming a single-platform, single-player game, but when transferring data between platforms with different endianness, it becomes a factor which must be considered. A good strategy to use when transferring data using a stream is to decide on an endianness for the stream itself. Then, when writing a multibyte data type, if the platform endianness does not match the chosen stream endianness, the byte order of the data should be reversed when being written into the stream. Similarly, when data is read from the stream, if the platform endianness differs from the stream endianness, the byte order should be reversed.

Most platforms provide efficient byte swapping algorithms, and some even have intrinsics or assembly instructions. However if you need to roll your own, Listing 4.3 provides effective byte swapping functions.

Listing 4.3 Byte Swapping Functions

```
inline uint16_t ByteSwap2(uint16_t inData)
{
    return (inData >> 8) | (inData << 8);
}
inline uint32_t ByteSwap4(uint32_t inData)
{
    return ((inData >> 24) & 0x000000ff)|
            ((inData >> 8) & 0x0000ff00)|
            ((inData << 8) & 0x00ff0000)|
            ((inData << 24) & 0xff000000);
}

inline uint64_t ByteSwap8(uint64_t inData)
{
    return ((inData >> 56) & 0x00000000000000ff)|
            ((inData >> 40) & 0x000000000000ff00)|
            ((inData >> 24) & 0x0000000000ff0000)|
            ((inData >> 8) & 0x00000000ff000000)|
            ((inData << 8) & 0x000000ff00000000)|
            ((inData << 24) & 0x0000ff0000000000)|
            ((inData << 40) & 0x00ff000000000000)|
            ((inData << 56) & 0xff00000000000000);
}
```

These functions handle basic unsigned integers of the given size, but not other data types that need to be byte swapped, such as floats, doubles, signed integers, large enums, and more. To do that, it takes some tricky type aliasing:

```
template <typename tFrom, typename tTo>
class TypeAliaser
{
public:
    TypeAliaser(tFrom inFromValue):
        mAsFromType(inFromValue) {}
    tTo& Get() {return mAsToType;}

    union
    {
        tFrom     mAsFromType;
        tTo        mAsToType;
    };
};
```

This class provides a method to take data of one type, such as a `float`, and treat it as a type for which there is already a byte swap function implemented. Templating some helper functions as in Listing 4.4 then enables swapping any type of primitive data using the appropriate function.

Listing 4.4 Templated Byte Swapping Functions

```
template <typename T, size_t tSize> class ByteSwapper;

//specialize for 2...
template <typename T>
class ByteSwapper<T, 2>
{
public:
    T Swap(T inData) const
    {
        uint16_t result =
          ByteSwap2(TypeAliaser<T, uint16_t>(inData).Get());
        return TypeAliaser<uint16_t, T>(result).Get();
    }
};

//specialize for 4...
template <typename T>
class ByteSwapper<T, 4>
{
public:
    T Swap(T inData) const
    {
        uint32_t result =
          ByteSwap4(TypeAliaser<T, uint32_t>(inData).Get());
        return TypeAliaser<uint32_t, T>(result).Get();
    }
};

//specialize for 8...
template <typename T>
class ByteSwapper<T, 8>
{
public:
    T Swap(T inData) const
    {
        uint64_t result =
          ByteSwap8(TypeAliaser<T, uint64_t>(inData).Get());
        return TypeAliaser<uint64_t, T>(result).Get();
    }
};
```

```
template <typename T>
T ByteSwap(T inData)
{

    return ByteSwapper<T, sizeof(T) >().Swap(inData);
}
```

Calling the templated `ByteSwap` function creates an instance of `ByteSwapper`, templated based on the size of the argument. This instance then uses the `TypeAliaser` to call the appropriate `ByteSwap` function. Ideally, the compiler optimizes the intermediate invocations away, leaving a few operations that just swap the order of some bytes in a register.

> **note**
>
> Not all data needs to be byte swapped just because the platform endianness doesn't match the stream endianness. For instance, a string of single-byte characters doesn't need to be byte swapped because even though the string is multiple bytes, the individual characters are only a single byte each. Only primitive data types should be byte swapped, and they should be swapped at a resolution that matches their size.

Using the `ByteSwapper`, the generic `Write` and `Read` functions can now properly support a stream with endianness that differs from that of the runtime platform:

```
template<typename T> void Write(T inData)
{
    static_assert(
        std::is_arithmetic<T>::value||
        std::is_enum<T>::value,
        "Generic Write only supports primitive data types");

    if(STREAM_ENDIANNESS == PLATFORM_ENDIANNESS)
    {
        Write(&inData, sizeof(inData));
    }
    else
    {
        T swappedData = ByteSwap(inData);
        Write(&swappedData, sizeof( swappedData));
    }

}
```

Bit Streams

One limitation of the memory streams described in the previous section is that they can only read and write data that is an integral number of bytes. When writing networking code, it is often desirable to represent values with as few bits as possible, and this can require reading and writing with single-bit precision. To this end, it is helpful to implement a **memory bit stream**, able to serialize data that is any number of bits. Listing 4.5 contains a declaration of an **output memory bit stream**.

Listing 4.5 Declaration of an Output Memory Bit Stream

```
class OutputMemoryBitStream
{
public:

    OutputMemoryBitStream()     {ReallocBuffer(256);}
    ~OutputMemoryBitStream()    {std::free(mBuffer);}

    void    WriteBits(uint8_t inData, size_t inBitCount);
    void    WriteBits(const void* inData, size_t inBitCount);

    const char* GetBufferPtr()    const    {return mBuffer;}
    uint32_t    GetBitLength()    const    {return mBitHead;}
    uint32_t    GetByteLength()   const    {return (mBitHead + 7) >> 3;}

    void    WriteBytes(const void* inData, size_t inByteCount
                    {WriteBits(inData, inByteCount << 3);}

private:
    void        ReallocBuffer(uint32_t inNewBitCapacity);

    char*        mBuffer;
    uint32_t    mBitHead;
    uint32_t    mBitCapacity;
};
```

The interface of the bit stream is similar to that of the byte stream, except it includes the ability to pass a number of bits to write instead of the number of bytes. The construction, destruction, and reallocation for expansion are similar as well. The new functionality lies in the two new `WriteBits` methods shown in Listing 4.6.

Listing 4.6 Implementation of an Output Memory Bit Stream

```
void OutputMemoryBitStream::WriteBits(uint8_t inData,
                                      size_t inBitCount)
{
    uint32_t nextBitHead = mBitHead + static_cast<uint32_t>(inBitCount);
    if(nextBitHead > mBitCapacity)
```

```
    {
        ReallocBuffer(std::max(mBitCapacity * 2, nextBitHead));
    }

    //calculate the byteOffset into our buffer
    //by dividing the head by 8
    //and the bitOffset by taking the last 3 bits
    uint32_t byteOffset = mBitHead >> 3;
    uint32_t bitOffset = mBitHead & 0x7;

    //calculate which bits of the current byte to preserve
    uint8_t currentMask = ~(0xff << bitOffset);
    mBuffer[byteOffset] = (mBuffer[byteOffset] & currentMask)
        |(inData << bitOffset);

    //calculate how many bits were not yet used in
    //our target byte in the buffer
    uint32_t bitsFreeThisByte = 8 - bitOffset;

    //if we needed more than that, carry to the next byte
    if(bitsFreeThisByte < inBitCount)
    {
        //we need another byte
        mBuffer[byteOffset + 1] = inData >> bitsFreeThisByte;
    }

    mBitHead = nextBitHead;
}

void OutputMemoryBitStream::WriteBits(const void* inData, size_t inBitCount)
{
    const char* srcByte = static_cast<const char*>(inData);
    //write all the bytes
    while(inBitCount > 8)
    {
        WriteBits(*srcByte, 8);
        ++srcByte;
        inBitCount -= 8;
    }
    //write anything left
    if(inBitCount > 0)
    {
        WriteBits(*srcByte, inBitCount);
    }
}
```

The innermost task of writing bits to the stream is handled by the `WriteBits(uint8_t inData, size_t inBitCount)` method, which takes a single byte and writes a given

number of bits from that byte into the bit stream. To understand how this works, consider what happens when the following code is executed:

```
OutputMemoryBitStream mbs;

mbs.WriteBits(13, 5);
mbs.WriteBits(52, 6);
```

This should write the number 13 using 5 bits and then the number 52 using 6 bits. Figure 4.3 shows these numbers in binary.

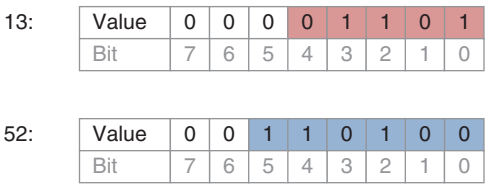

Figure 4.3 Binary representation of 13 and 52

Therefore, when the code is run, the memory pointed to by `mbs.mBuffer` should be left containing the two values, as in Figure 4.4.

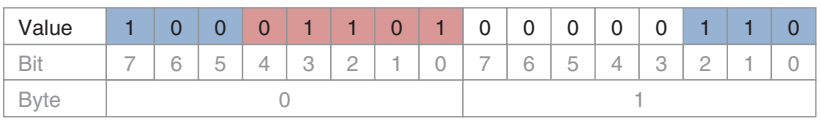

Figure 4.4 Stream buffer containing 5 bits of 13 and 6 bits of 52

Notice the 5 bits of the number 13 take up the first 5 bits of byte 0, and then the 6 bits of the number 52 take up the last 3 bits of byte 0 and the first 3 bits of byte 1.

Stepping through the code shows how the method achieves this. Assume the stream has been freshly constructed, so `mBitCapacity` is 256, `mBitHead` is 0, and there is enough room in the stream to avoid reallocation. First, the `mBitHead`, which represents the index of the next bit in the stream to be written, is decomposed into a byte index and a bit index within that byte. Because a byte is 8 bits, the byte index can always be found by dividing by 8, which is the same as shifting right by 3. Similarly, the index of the bit within that byte can be found by examining those same 3 bits that were shifted away in the previous step. Because 0x7 is 111 in binary, bitwise ANDing the `mBitHead` with 0x7 yields just the 3 bits. In the first call to write the number 13, `mBitHead` is 0, so `byteOffset` and `bitOffset` are both 0 as well.

Once the method calculates the `byteOffset` and `bitOffset`, it uses the `byteOffset` as an index into the `mBuffer` array to find the target byte. Then it shifts the data left by the bit offset and bitwise ORs it into the target byte. This is all elementary when writing the number 13 because both offsets are 0. However, consider how the stream looks at the beginning of the `WriteBits(52, 6)` call, as shown in Figure 4.5.

Value	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte	0								1							

Figure 4.5 Stream buffer immediately before the second `WriteBits` call

At this point, `mBitHead` is 5. That means `byteOffset` is 0 and `bitOffset` is 5.

Shifting 52 left by 5 bits yields the result shown in Figure 4.6.

Value	1	0	0	0	0	0	0	0
Bit	7	6	5	4	3	2	1	0

Figure 4.6 Binary representation of 52 shifted left by 5 bits

Note the high-order bits are shifted out of range, and the low-order bits become the high bits. Figure 4.7 shows the result of bitwise ORing those bits into byte 0 of the buffer.

Value	1	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte	0								1							

Figure 4.7 52, shifted left by 5 bits, bitwise ORed into the stream buffer

Byte 0 is complete, but only three of the necessary 6 bits have been written to the stream due to the overflow when shifting left. The next lines of `WriteBits` detect and handle this issue. The method calculates how many bits were initially free in the target byte by subtracting the `bitOffset` from 8. In this case, that yields 3, which is the number of bits that were able to fit. If the number of bits free is less than the number of bits to be written, the overflow branch executes.

In the overflow branch, the next byte is targeted. To calculate what to OR into the next byte, the method shifts `inData` right by the number of bits that were free. Figure 4.8 shows the result of shifting 52 to the right by 3 bits.

Value	0	0	0	0	0	1	1	0
Bit	7	6	5	4	3	2	1	0

Figure 4.8 52, Shifted to the right by 3 bits

The high-order bits that overflowed when shifted left are now shifted to the right to become the low-order bits of the higher-order byte. When the method ORs the right-shifted bits into the byte at `mBuffer[byteOffset + 1]`, it leaves the stream in the final state expected (see Figure 4.9).

Value	1	0	0	0	1	1	0	1	0	0	0	0	0	1	1	0
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte	0								1							

Figure 4.9 Proper final state of the stream's buffer

With the hard work done by `WriteBits(uint8_t inData, uint32_t inBitCount)`, all that remains is for `WriteBits(const void* inData, uint32_t inBitCount)` to break the data up into bytes and call the previous `WriteBits` method 1 byte at a time.

This output memory bit stream is functionally complete, but not ideal. It requires specifying a number of bits for every piece of data written into the stream. However, in most cases, the upper bound for the number of bits depends on the type of data being written. Only sometimes it is useful to use fewer than the upper bound. For this reason, it increases code clarity and maintainability to add some methods for basic data types:

```
void WriteBytes(const void* inData, size_t inByteCount)
    {WriteBits(inData, inByteCount << 3);}

void Write(uint32_t inData, size_t inBitCount = sizeof(uint32_t) * 8)
    {WriteBits(&inData, inBitCount);}
void Write(int inData, size_t inBitCount = sizeof(int) * 8)
    {WriteBits(&inData, inBitCount);}
void Write(float inData)
    {WriteBits(&inData, sizeof(float) * 8);}

void Write(uint16_t inData, size_t inBitCount = sizeof(uint16_t) * 8)
    {WriteBits(&inData, inBitCount);}
void Write(int16_t inData, size_t inBitCount = sizeof(int16_t) * 8)
    {WriteBits(&inData, inBitCount);}

void Write(uint8_t inData, size_t inBitCount = sizeof(uint8_t) * 8)
    {WriteBits(&inData, inBitCount);}
```

```
void Write(bool inData)
    {WriteBits(&inData, 1);}
```

With these methods, most primitive types can be written by simply passing them to the `Write` method. The default parameter takes care of supplying the corresponding number of bits. For cases where the caller desires a fewer number of bits, the methods accept an override parameter. A templated function and type traits again provide even more generality than multiple overloads:

```
template<typename T>
void Write(T inData, size_t inBitCount = sizeof(T) * 8)
{
    static_assert(std::is_arithmetic<T>::value||
                  std::is_enum<T>::value,
                  "Generic Write only supports primitive data types");
    WriteBits(&inData, inBitCount);
}
```

Even with the templated `Write` method, it is still useful to implement a specific overload for `bool` because its default bit count should be 1, not `sizeof(bool) * 8`, which would be 8.

warning

This implementation of the `Write` method works only on little-endian platforms due to the way it addresses individual bytes. If the method needs to operate on a big-endian platform, it should either byte swap the data in the templated `Write` function before the data goes into `WriteBits`, or it should address the bytes using a big-endian compatible method.

The **input memory bit stream**, which reads bits back out of the stream, works in a similar manner to the output memory bit stream. Implementation is left as an exercise and can also be found at the companion website.

Referenced Data

The serialization code can now handle all kinds of primitive and POD data, but it falls apart when it needs to handle indirect references to data, through pointers or other containers. Recall the `RoboCat` class (as shown next):

```
class RoboCat: public GameObject
{
public:
    RoboCat() mHealth(10), mMeowCount(3),
              mHomeBase(0)
    {
        mName[0] = '\0';
    }
    virtual void Update();

    void Write(OutputMemoryStream& inStream) const;
    void Read(InputMemoryStream& inStream);

private:
    int32_t                  mHealth;
    int32_t                  mMeowCount;
    GameObject*              mHomeBase;
    char                     mName[128];
    std::vector<int32_t>     mMiceIndices;

    Vector3                  mPosition;
    Quaternion               mRotation;
};
```

There are two complex member variables which the current memory stream implementation cannot yet serialize—`mHomeBase` and `mMiceIndices`. Each calls for a different serialization strategy, as discussed in the following sections.

Inlining or Embedding

Sometimes network code must serialize member variables that reference data not shared with any other object. `mMiceIndices` in `RoboCat` is a good example. It is a vector of integers tracking the indices of various mice in which the `RoboCat` is interested. Because the `std::vector<int>` is a black box, it is unsafe to use the standard `OutputMemoryStream::Write` function to copy from the address of the `std::vector<int>` into the stream. Doing so would serialize the values of any pointers that are in `std::vector`, which when deserialized on a remote host would point to garbage.

Instead of serializing the vector itself, a custom serialization function should write only the data contained within the vector. That data in RAM may actually be far away from the data of the `RoboCat` itself. However, when the custom function serializes it, it does so into the stream inline, embedded right in the middle of the `RoboCat` data. For this reason, this process is known as **inlining** or **embedding**. For instance, a function to serialize a `std::vector<int32_t>` would look like this:

```
void Write(const std::vector<int32_t>& inIntVector)
{
    size_t elementCount = inIntVector.size();
```

```
    Write(elementCount);
    Write(inIntVector.data(), elementCount * sizeof(int32_t));
}
```

First, the code serializes the length of the vector, and then all the data from the vector. Note that the `Write` method must serialize the length of the vector first so that the corresponding `Read` method can use it to allocate a vector of the appropriate length before deserializing the contents. Because the vector is just primitive integers, the method serializes it all in one straight `memcpy`. To support more complex data types, a templated version of the `std::vector` `Write` method serializes each element individually:

```
template<typename T>
void Write(const std::vector<T>& inVector)
{
    size_t elementCount = inVector.size();
    Write(elementCount);
    for(const T& element: inVector)
    {
        Write(element);
    }
}
```

Here, after serializing the length, the method individually embeds each element from the vector. This allows it to support vectors of vectors, or vectors of classes that contain vectors, and so on. Deserializing requires a similarly implemented `Read` function:

```
template<typename T>
void Read(std::vector<T>& outVector)
{
    size_t elementCount;
    Read(elementCount);
    outVector.resize(elementCount);
    for(const T& element: outVector)
    {
        Read(element);
    }
}
```

Additional specialized `Read` and `Write` functions can support other types of containers, or any data referenced by a pointer, as long as that data is wholly owned by the parent object being serialized. If the data needs to be shared with or pointed to by other objects, then a more complex solution, known as linking, is required.

Linking

Sometimes serialized data needs to be referenced by more than one pointer. For instance, consider the `GameObject* mHomeBase` in `RoboCat`. If two `RoboCat`s share the same home base, there is no way to represent that fact using the current toolbox. Embedding would just

embed a copy of the same home base in each `RoboCat` when serialized. During deserialization, this would result in the creation of two different home bases!

Other times, the data is structured in such a way that embedding is just impossible. Consider the `HomeBase` class:

```
class HomeBase: public GameObject
{
    std::vector<RoboCat*> mRoboCats;
};
```

The `HomeBase` contains a list of all its active `RoboCat`s. Consider a function to serialize a `RoboCat` using only embedding. While serializing a `RoboCat`, the function would embed its `HomeBase`, which would then embed all its active `RoboCats`, including the `RoboCat` it was currently serializing. This is a recipe for stack overflow due to infinite recursion. Clearly another tool is necessary.

The solution is to give each multiply referenced object a unique identifier and then to serialize references to those objects by serializing just the identifier. Once all the objects are deserialized on the other end of the network, a fix-up routine can use the identifiers to find the referenced objects and plug them into the appropriate member variables. It is for this reason the process is commonly referred to as **linking**.

Chapter 5 discusses how to assign unique IDs to each object sent over the network and how to maintain maps from IDs to objects and vice versa. For now, assume each stream has access to a `LinkingContext` (as shown in Listing 4.7) that contains an up-to-date map between network IDs and game objects.

Listing 4.7 Linking Context

```
class LinkingContext
{
public:

    uint32_t GetNetworkId(GameObject* inGameObject)
    {
        auto it = mGameObjectToNetworkIdMap.find(inGameObject);
        if(it != mGameObjectToNetworkIdMap.end())
        {
            return it->second;
        }
        else
        {
            return 0;
        }
    }

    GameObject* GetGameObject(uint32_t inNetworkId)
```

```
    {
        auto it = mNetworkIdToGameObjectMap.find(inNetworkId);
        if(it != mNetworkIdToGameObjectMap.end())
        {
            return it->second;
        }
        else
        {
            return nullptr;
        }
    }
private:
    std::unordered_map<uint32_t, GameObject*>
        mNetworkIdToGameObjectMap;
    std::unordered_map<GameObject*, uint32_t>
        mGameObjectToNetworkIdMap;
};
```

The `LinkingContext` enables a simple linking system in the memory stream:

```
void Write(const GameObject* inGameObject)
{
    uint32_t networkId =
        mLinkingContext->GetNetworkId(inGameObject);
    Write(networkId);
}

void Read(GameObject*& outGameObject)
{
    uint32_t networkId;
    Read(networkId);
    outGameObject = mLinkingContext->GetGameObject(networkId);
}
```

note

When fully implemented, a linking system, and the gameplay code that uses it, must be tolerant of receiving a network ID for which there is no object mapped. Because packets can be dropped, a game might receive an object with a member variable that refers to an object not yet sent. There are many different ways to handle this—the game could either ignore the entire object, or it could deserialize the object and link up whatever references are available, leaving the missing ones as null. A more complex system might keep track of the member variable with the null link so that when an object for the given network ID is received, it can link it in. The choice depends on the specifics of the game's design.

Compression

With the tools to serialize all types of data, it is possible to write code to send game objects back and forth across the network. However, it will not necessarily be efficient code that functions within the bandwidth limitations imposed by the network itself. In the early days of multiplayer gaming, games had to make do with 2400 bytes per second connections, or less. These days, game engineers are luckier to have high-speed connections many orders of magnitude faster, but they must still concern themselves with how to use that bandwidth as efficiently as possible.

A large game world can have hundreds of moving objects, and sending real-time exhaustive data about those objects to the potentially hundreds of connected players is enough to saturate even the highest bandwidth connection. This book examines many ways to make the most of the available bandwidth. Chapter 9, "Scalability," looks at high-level algorithms which determine who should see what data and which object properties need to be updated for which clients. This section, however, starts at the lowest level by examining common techniques for compressing data at the bit and byte level. That is, once a game has determined that a particular piece of data needs to be sent, how can it send it using as few bits as possible?

Sparse Array Compression

The trick to compressing data is to remove any information that does not need to be sent over the network. A good place to look for this kind of information is in any sparse or incompletely filled data structures. Consider the `mName` field in `RoboCat`. For whatever reason, the original `RoboCat` engineer decided that the best way to store the name of a `RoboCat` is with a 128-byte character array in the middle of the data type. The stream method `WriteBytes(const void* inData, uint32_t inByteCount)` can already embed the character array, but if used judiciously, it can most likely serialize the necessary data without writing a full 128 bytes.

Much of compression strategy comes down to analyzing the average case and implementing algorithms to take advantage of it, and that is the approach to take here. Given typical names in the English language, and the game design of *Robo Cat*, the odds are good that a user won't need all 128 characters to name her `RoboCat`. The same could be said about the array no matter what length it is: Just because it allows space for a worst case, serialization code doesn't have to assume that every user will exploit that worst case. As such, a custom serializer can save space by looking at the `mName` field and counting how many characters are actually used by the name. If `mName` is null terminated, the task is made trivial by the `std::strlen` function. For instance, a more efficient way to serialize the name is shown here:

```
void RoboCat::Write(OutputMemoryStream& inStream) const
{
    ...//serialize other fields up here

    uint8_t nameLength =
        static_cast<uint8_t>(strlen(mName));
```

```
    inStream.Write(nameLength);
    inStream.Write(mName, nameLength);
    ...
}
```

Notice that, just as when serializing a vector, the method first writes the length of the serialized data before writing the data itself. This is so the receiving end knows how much data to read out of the stream. The method serializes the length of the string itself as a single byte. This is only safe because the entire array holds a maximum of 128 characters.

In truth, assuming the name is infrequently accessed compared to the rest of the `RoboCat`'s data, it is more efficient from a cache perspective to represent an object's name with an `std::string`, allowing the entire `RobotCat` data type to fit in fewer cache lines. In this case, a string serializing method similar to the vector method implemented in the previous section would handle the name serialization. That makes this particular `mName` example a bit contrived for clarity's sake, but the lesson holds true, and sparse containers are a good low-hanging target for compression.

Entropy Encoding

Entropy encoding is a subject of information theory which deals with compressing data based on how unexpected it is. According to information theory, there is less information or **entropy** in a packet that contains expected values than in a packet that contains unexpected values. Therefore, code should require fewer bits to send expected values than to send unexpected ones.

In most cases, it is more important to spend CPU cycles simulating the actual game than to calculate the exact amount of entropy in a packet to achieve optimal compression. However, there is a very simple form of entropy encoding that is quite efficient. It is useful when serializing a member variable that has a particular value more frequently than any other value.

As an example, consider the `mPosition` field of `RoboCat`. It's a `Vector3` with an X, Y, and Z component. X and Z represent the cat's position on the ground, and Y represents the cat's height above ground. A naïve serialization of the position would look like so:

```
void OutputMemoryBitStream::Write(const Vector3& inVector)
{
    Write(inVector.mX);
    Write(inVector.mY);
    Write(inVector.mZ);
}
```

As written, it requires $3 \times 4 = 12$ bytes to serialize a `RoboCat`'s `mPosition` over the network. However, the naïve code does not take advantage of the fact that cats can often be found on the ground. This means that the Y coordinate for most `mPosition` vectors is going to be 0. The method can use a single bit to indicate whether the `mPosition` has the common value of 0, or some other, less common value:

```
void OutputMemoryBitStream::WritePos(const Vector3& inVector)
{
    Write(inVector.mX);
    Write(inVector.mZ);

    if(inVector.mY == 0)
    {
        Write(true);
    }
    else
    {
        Write(false);
        Write(inVector.mY);
    }
}
```

After writing the X and Y components, the method checks if the height off the ground is 0 or not. If it is 0, it writes a single true bit, indicating, "yes, this object has the usual height of 0." If the Y component is not 0, it writes a single false bit, indicating, "the height is not 0, so the next 32 bits will represent the actual height." Note that in the worst case, it now takes 33 bits to represent the height—the single flag to indicate whether this is a common or uncommon value, and then the 32 to represent the uncommon value. At first, this may seem inefficient, as serialization now may use more bits than ever before. However, calculating the true number of bits used in the average case requires factoring in exactly how common it is that a cat is on the ground.

In-game telemetry can log exactly how often a user's cat is on the floor—either from testers playing on site or from actual users playing an earlier version of the game and submitting analytics over the Internet. Assume such an experiment determines that players are on the ground 90% of the time. Basic probability then dictates the expected number of bits required to represent the height:

$$P_{OnGround} * Bits_{OnGround} + P_{InAir} * Bits_{InAir} + 0.9*1 + 0.1*33 + 4.2$$

The expected number of bits to serialize the Y component has dropped from 32 to 4.2: That saves over 3 bytes per position. With 32 players changing positions 30 times a second, this can add up to a significant savings from just this one member variable.

The compression can be even more efficient. Assume that analytics show that whenever the cat is not on the floor, it is usually hanging from the ceiling, which has a height of 100. The serialization code can then support a second common value to compress positions on the ceiling:

```
void OutputMemoryBitStream::WritePos(const Vector3& inVector)
{
    Write(inVector.mX);
    Write(inVector.mZ);

    if(inVector.mY == 0)
```

```
    {
        Write(true);
        Write(true);
    }
    else if(inVector.mY == 100)
    {
        Write(true);
        Write(false);
    }
    else
    {
        Write(false);
        Write(inVector.mY);
    }
}
```

The method still uses a single bit to indicate whether the height contains a common value or not, but then it adds a second bit to indicate which of the common values it's using. Here the common values are hardcoded into the function, but with too many more values this technique can get quite messy. In that case, a simplified implementation of Huffman coding could use a lookup table of common values, with a few bits to a store an index into that lookup table.

The question remains, though, of whether this optimization is a good one—just because the ceiling is a second common location for a cat, it's not necessarily an efficient optimization to make, so it is necessary to check the math. Assume that analytics show cats are on the ceiling 7% of the time. In that case, the new expected number of bits used to represent a height can be calculated using this equation:

$$P_{OnGround} * Bits_{OnGround} + P_{InAir} * Bits_{InAir} + P_{OnCeiling} * Bits_{OnCeiling} + 0.9 * 2 + 0.07 * 2 + 0.03 * 33 + 2.93$$

The expected number of bits is 2.93, which is 1.3 bits fewer than the first optimization. Therefore the optimization is worthwhile.

There are many forms of entropy encoding, ranging from the simple, hardcoded one described here, to the more complex and popular Huffman coding, arithmetic coding, gamma coding, run length encoding, and more. As for everything in game development, the amount of CPU power to allocate to entropy encoding versus the amount to allocate elsewhere is a design decision. Resources on other encoding methods can be found in the "Additional Readings" section.

Fixed Point

Lightning fast calculations on 32-bit floating point numbers are the boon and the benchmark of the modern computing era. However, just because the game simulation performs floating point computations doesn't mean it needs all 32 bits to represent the numbers when sent across the network. A common and useful technique is to examine the known range and precision requirements of the numbers being sent and convert them to a fixed point format so that the data can be sent with the minimum number of bits necessary. To do this, you have to

sit down with designers and gameplay engineers and figure out exactly what your game needs. Once you know, you can begin to build a system that provides that as efficiently as possible.

As an example, consider the `mLocation` field again. The serialization code already compresses the Y component quite a bit, but it does nothing for the X and Z components: They are still using a full 32 bits each. A talk with the designers reveals that the *Robo Cat* game world's size is 4000 game units by 4000 game units, and the world is centered at the origin. This means that the minimum value for an X or Z component is –2000, and the maximum value is 2000. Further discussion and gameplay testing reveal that client-side positions only need to be accurate to within 0.1 game units. That's not to say that the authoritative server's position doesn't have to be more accurate, but when sending a value to the client, it only needs to do so with 0.1 units of precision.

These limits provide all the information necessary to determine exactly how many bits should be necessary to serialize this value. The following formula provides the total number of possible values the X component might have:

(*MaxValue* + *MinValue*)/*Precision* + 1 + (2000 + +2000)/0.1 + 1 + 40001

This means there are 40001 potential values for the serialized component. If there is a mapping from an integer less than 40001 to a corresponding potential floating point value, the method can serialize the X and Z components simply by serializing the appropriate integers.

Luckily, this is a fairly simple task using something called **fixed point** numbers. A fixed point number is a number that looks like an integer, but actually represents a number equal to that integer divided by a previously decided upon constant. In this case, the constant is equal to the required level of precision needed. At that point, the method only needs to serialize the number of bits that is required to store an integer guaranteed to be less than 40001. Because $\log_2$ 40001 is 15.3, the routine should require only 16 bits each to serialize the X and Z components. Putting that all together results in the following code:

```
inline uint32_t ConvertToFixed(
    float inNumber, float inMin, float inPrecision)
{
    return static_cast<uint32_t> (
        (inNumber - inMin)/inPrecision);
}

inline float ConvertFromFixed(
    uint32_t inNumber, float inMin, float inPrecision )
{
    return static_cast<float>(inNumber) *
        inPrecision + inMin;
}
```

```
void OutputMemoryBitStream::WritePosF(const Vector3& inVector)
{
    Write(ConvertToFixed(inVector.mX, -2000.f, 0.1f), 16);
    Write(ConvertToFixed(inVector.mZ, -2000.f, 0.1f), 16);
    ... //write Y component here ...
}
```

The game stores the vector's components as full floating point numbers, but when it needs to send them over the network, the serialization code converts them to fixed point numbers between 0 and 40000 and sends them using only 16 bits a piece. This saves another full 32 bits on the vector, cutting its expected size down to 35 bits from the original 96.

> **note**
>
> On some CPUs, such as the PowerPC in the Xbox 360 and PS3, it can be computationally expensive to convert from a floating point to an integer and back. However, it is often worth the cost, given the amount of bandwidth it conserves. As with most optimizations, it is a tradeoff which must be decided upon based on the specifics of the individual game being developed.

Geometry Compression

Fixed point compression takes advantage of game-specific information to serialize data with as few bits as possible. Interestingly, this is just information theory at work again: Because there are constraints on the possible values for a variable, it requires a smaller number of bits to represent that information. This technique applies when serializing any data structure with known constraints on its contents.

Many geometric data types fall under just this case. This section discusses the quaternion and the transformation matrix. A **quaternion** is a data structure containing four floating point numbers, useful for representing a rotation in three dimensions. The exact uses of the quaternion are beyond the scope of this text, but more information can be found in the references in the "Additional Readings" section. What is important for this discussion is that when representing a rotation, a quaternion is normalized, such that each component is between –1 and 1, and the sum of the squares of each component is 1. Because the sum of the squares of the components is fixed, serializing a quaternion requires serializing only three of the four components, as well as a single bit to represent the sign of the fourth component. Then the deserializing code can reconstruct the final component by subtracting the squares of the other components from 1.

In addition, because all components are between –1 and 1, fixed point representation can further improve compression of the components, if there is an acceptable precision loss that does not affect gameplay. Often, 16 bits of precision are enough, giving 65535 possible values to represent the range from –1 to 1. This means that a four-component quaternion, which takes 128 bits in memory, can be serialized fairly accurately with as few as 49 bits:

```
void OutputMemoryBitStream::Write(const Quaternion& inQuat)
{
    float precision = (2.f / 65535.f);
    Write(ConvertToFixed(inQuat.mX, -1.f, precision), 16);
    Write(ConvertToFixed(inQuat.mY, -1.f, precision), 16);
    Write(ConvertToFixed(inQuat.mZ, -1.f, precision), 16);
    Write(inQuat.mW < 0);
}

void InputMemoryBitStream::Read(Quaternion& outQuat)
{
    float precision = (2.f / 65535.f);

    uint32_t f = 0;

    Read(f, 16);
    outQuat.mX = ConvertFromFixed(f, -1.f, precision);
    Read( f, 16 );
    outQuat.mY = ConvertFromFixed(f, -1.f, precision);
    Read(f, 16);
    outQuat.mZ = ConvertFromFixed(f, -1.f, precision);

    outQuat.mW = sqrtf(1.f -
                       outQuat.mX * outQuat.mX +
                       outQuat.mY * outQuat.mY +
                       outQuat.mZ * outQuat.mZ );
    bool isNegative;
    Read(isNegative);

    if(isNegative)
    {
        outQuat.mW *= -1;
    }
}
```

Geometric compression can also help when serializing an affine transformation matrix. A transformation matrix is 16 floats, but to be affine, it must decompose into a 3-float translation, a quaternion rotation, and a 3-float scale, for a total of 10 floats. Entropy encoding can then help save even more bandwidth if there are more constraints on the typical matrix to serialize. For instance, if the matrix is usually unscaled, the routine can indicate this with a single bit. If the scale is uniform, the routine can indicate this with a different bit pattern and then serialize only one component of the scale instead of all three.

Maintainability

Focusing solely on bandwidth efficiency can yield some slightly ugly code in some places. There are a few tradeoffs worth considering, sacrificing a little efficiency for ease of maintainability.

Abstracting Serialization Direction

Every new data structure or compression technique discussed in the previous sections has required both a read method and a write method. Not only does that mean implementing two methods for each new piece of functionality, but the methods must remain in sync with each other: If you change how a member variable is written, you must change how it's read. Having two such tightly coupled methods for each data structure is a bit of a recipe for frustration. It would be much cleaner if it were possible somehow to have only one method per data structure that could handle both reading and writing.

Luckily, through the use of inheritance and virtual functions, it is indeed possible. One way to implement this is to make `OutputMemoryStream` and `InputMemoryStream` both derive from a base `MemoryStream` class with a `Serialize` method:

```
class MemoryStream
{
     virtual void Serialize(void* ioData,
                            uint32_t inByteCount) = 0;
    virtual bool IsInput() const = 0;
};

class InputMemoryStream: public MemoryStream
{
    ...//other methods above here
    virtual void Serialize(void* ioData, uint32_t inByteCount)
    {
        Read(ioData, inByteCount);
    }
    virtual bool IsInput() const {return true;}

};

class OutputMemoryStream: public MemoryStream
{
    ...//other methods above here
    virtual void Serialize(void* ioData, uint32_t inByteCount)
    {
        Write(ioData, inByteCount);
    }

    virtual bool IsInput() const {return false;}
}
```

By implementing `Serialize`, the two child classes can take a pointer to data and a size and then perform the appropriate action, either reading or writing. Using the `IsInput` method, a function can check whether it has been passed an input stream or output stream. Then, the base `MemoryStream` class can implement a templated `Serialize` method assuming that the non-templated version is properly implemented by a subclass:

```
template<typename T> void Serialize(T& ioData)
{
    static_assert(std::is_arithmetic<T>::value||
                  std::is_enum<T>::value,
    "Generic Serialize only supports primitive data types");

    if(STREAM_ENDIANNESS == PLATFORM_ENDIANNESS)
    {
        Serialize(&ioData, sizeof(ioData) );
    }
    else
    {
        if(IsInput())
        {
            T data;
            Serialize(&data, sizeof(T));
            ioData = ByteSwap(data);
        }
        else
        {
            T swappedData = ByteSwap(ioData);
            Serialize(&swappedData, sizeof(swappedData));
        }
    }
}
```

The templated `Serialize` method takes generic data as a parameter and will either read it or write it, depending on the child class's non-templated `Serialize` method. This facilitates the replacement of each pair of custom `Read` and `Write` methods with a corresponding `Serialize` method. The custom `Serialize` method needs to take only a `MemoryStream` as a parameter and it can read or write appropriately using the stream's virtual `Serialize` method. This way, a single method handles both reading and writing for a custom class, ensuring that input and output code never get out of sync.

> **warning**
>
> This implementation is slightly more inefficient than the previous one because of all the virtual function calls required. This system can be implemented using templates instead of virtual functions to regain some of the performance hit, but that is left as an exercise for you to try.

Data-Driven Serialization

Most object serialization code follows the same pattern: For each member variable in an object's class, serialize that member variable's value. There may be some optimizations, but the general structure of the code is usually the same. In fact, it is so similar that if a game somehow

had data at runtime about what member variables were in an object, it could use a single serialization method to handle most of serialization needs.

Some languages, like C# and Java, have built-in reflection systems that allow runtime access to class structure. In C++, however, reflecting class members at runtime requires a custom built system. Luckily, building a basic reflection system is not too complicated (see Listing 4.8).

Listing 4.8 Basic Reflection System

```
enum EPrimitiveType
{
    EPT_Int,
    EPT_String,
    EPT_Float
};

class MemberVariable
{
public:
    MemberVariable(const char* inName,
        EPrimitiveType inPrimitiveType, uint32_t inOffset):
    mName(inName),
    mPrimitiveType(inPrimitiveType),
    mOffset(inOffset) {}

    EPrimitiveType     GetPrimitiveType() const {return mPrimitiveType;}
    uint32_t           GetOffset()        const {return mOffset;}

private:
    std::string       mName;
    EPrimitiveType    mPrimitiveType;
    uint32_t          mOffset;
};

class DataType
{
public:
    DataType(std::initializer_list<const MemberVariable& > inMVs):
    mMemberVariables(inMVs)
    {}

    const std::vector<MemberVariable>& GetMemberVariables() const
    {
        return mMemberVariables;
    }
private:
    std::vector< MemberVariable >    mMemberVariables;
};
```

`EPrimitiveType` represents the primitive type of a member variable. This system supports only `int`, `float`, and `string`, but it is easy to extend with any primitive type desired.

The `MemberVariable` class represents a single member variable in a compound data type. It holds the member variable's name (for debugging purposes), its primitive type, and its memory offset in its parent data type. Storing the offset is a critical: Serialization code can add the offset to the base address of a given object to find the location in memory of the member variable's value for that particular object. This is how it will read and write the member variable's data.

Finally, the `DataType` class holds all the member variables for a particular class. For each class that supports data-driven serialization, there is one corresponding instance of `DataType`. With the reflection infrastructure in place, the following code loads up the reflection data for a sample class:

```
#define OffsetOf(c, mv) ((size_t) & (static_cast<c*>(nullptr)->mv)))

class MouseStatus
{
public:
    std::string    mName;
    int            mLegCount, mHeadCount;
    float          mHealth;

    static DataType* sDataType;
    static void InitDataType()
    {
        sDataType = new DataType(
        {
            MemberVariable("mName",
                EPT_String, OffsetOf(MouseStatus,mName)),
            MemberVariable("mLegCount",
                EPT_Int, OffsetOf(MouseStatus, mLegCount)),
            MemberVariable("mHeadCount",
                EPT_Int, OffsetOf(MouseStatus, mHeadCount)),
            MemberVariable("mHealth",
                EPT_Float, OffsetOf(MouseStatus, mHealth))
        });
    }
};
```

Here, a sample class tracks a `RoboMouse`'s status. The static `InitDataType` function must be called at some point to initialize the `sDataType` member variable. That function creates the `DataType` that represents the `MouseStatus` and fills in the `mMemberVariables` entries. Notice the use of a custom `OffsetOf` macro to calculate the proper offset of each member variable. The built-in C++ `offsetof` macro has undefined behavior for non-POD classes. As

such, some compilers actually return compile errors when `offsetof` is used on classes with virtual functions or other non-POD types. As long as the class doesn't define a custom unary & operator, and the class hierarchy doesn't use virtual inheritance or have any member variables that are references, the custom macro will work. Ideally, instead of having to fill in the reflection data with handwritten code, a tool would analyze the C++ header files and automatically generate the reflection data for the classes.

From here, implementing a simple serialize function is just a matter of looping through the member variables in a data type:

```
void Serialize(MemoryStream* inMemoryStream,
            const DataType* inDataType, uint8_t* inData)
{
    for(auto& mv: inDataType->GetMemberVariables())
    {
        void* mvData = inData + mv.GetOffset();
        switch(mv.GetPrimitiveType())
        {
            EPT_Int:
               inMemoryStream->Serialize(*(int*) mvData);
               break;
            EPT_String:
               inMemoryStream->Serialize(*(std::string*) mvData);
               break;
            EPT_Float:
               inMemoryStream->Serialize(*(float*) mvData);
               break;
        }
    }
}
```

The `GetOffset` method of each member variable calculates a pointer to the instance's data for that member. Then the switch on `GetPrimitiveType` casts the data to the appropriate type and lets the typed `Serialize` function take care of the actual serialization.

This technique can be made more powerful by expanding the metadata tracked in the `MemberVariable` class. For instance, it could store the number of bits to use for each variable for automatic compression. Additionally, it could store potential common values for the member variable to support a procedural implementation of some entropy encoding.

As a whole, this method trades performance for maintainability: There are more branches that might cause pipeline flushes, but there is less code to write overall and, therefore, fewer chances for errors. As an extra benefit, a reflection system is useful for many things besides network serialization. It can be helpful when implementing serialization to disk, garbage collection, a GUI object editor, and more.

Summary

Serialization is the process of taking a complex data structure and breaking it down into a linear array of bytes, which can be sent to another host across a network. The naïve approach of simply using `memcpy` to copy the structure into a byte buffer does not usually work. The stream, the basic workhorse for serialization makes it possible to serialize complex data structures, including those that reference other data structures and relink those references after deserialization.

There are several techniques for serializing data efficiently. Sparse data structures can be serialized into more compact forms. Expected values of member variables can be compressed losslessly using entropy encoding. Geometric or other similarly constrained data structures can also be compressed losslessly by making use of the constraints to send only the data that is necessary to reconstruct the data structure. When slightly lossy compression is acceptable, floating point numbers can be turned into fixed point numbers based on the known range and necessary precision of the value.

Efficiency often comes at the cost of maintainability, and sometimes it is worthwhile to reinject some maintainability into a serialization system. `Read` and `Write` methods for a data structure can be collapsed into a single `Serialize` method which reads or writes depending on the stream on which it is operating, and serialization can be data-driven, using auto- or hand-generated metadata to serialize objects without requiring custom, per-data structure read and write functions.

With these tools, you have everything you need to package up an object and send it to a remote host. The next chapter discusses both how to frame this data so that the remote host can create or find the appropriate object to receive the data, and how to efficiently handle partial serialization when a game requires that only a subset of an object's data be serialized.

Review Questions

1. Why is it not necessarily safe to simply `memcpy` an object into a byte buffer and send that buffer to a remote host?
2. What is endianness? Why is it a concern when serializing data? Explain how to handle endian issues when serializing data.
3. Describe how to efficiently compress a sparse data structure.
4. Give two ways to serialize an object with pointers in it. Give an example of when each way is appropriate.
5. What is entropy encoding? Give a basic example of how to use it.
6. Explain how to use fixed point numbers to save bandwidth when serializing floating point numbers.

7. Explain thoroughly why the `WriteBits` function as implemented in this chapter only works properly on little-endian platforms. Implement a solution that will work on big-endian platforms as well.
8. Implement `MemoryOutputStream::Write(const unordered_map<int, int >&)` that allows the writing of a map from integer to integer into the stream.
9. Write the corresponding `MemoryOutputStream::Read(unordered_map<int, int >&)` method.
10. Template your implementation of `MemoryOutputStream::Read` from Question 9 so it works properly for `template <tKey, tValue> unordered_map<tKey, tValue>`.
11. Implement an efficient `Read` and `Write` for an affine transformation matrix, taking advantage of the fact that the scale is usually 1, and when not 1, is usually at least uniform.
12. Implement a serialization module with a generic serialize method that relies on templates instead of virtual functions.

Additional Readings

Bloom, Charles. (1996, August 1). *Compression: Algorithms: Statistical Coders*. Retrieved from http://www.cbloom.com/algs/statisti.html. Accessed September 12, 2015.

Blow, Jonathan. (2004, January 17). *Hacking Quaternions*. Retrieved from http://number-none.com/product/Hacking%20Quaternions/. Accessed September 12, 2015.

Ivancescu, Gabriel. (2007, December 21). *Fixed Point Arithmetic Tricks*. Retrieved from http://x86asm.net/articles/fixed-point-arithmetic-and-tricks/. Accessed September 12, 2015.

This page intentionally left blank

CHAPTER 5

OBJECT REPLICATION

Serializing object data is only the first step in transmitting state between hosts. This chapter investigates a generalized replication framework which supports synchronization of world and object state between remote processes.

The State of the World

To be successful, a multiplayer game must make concurrent players feel like they are playing in the same world. When one player opens a door or kills a zombie, all players in range need to see that door open, or that zombie explode. Multiplayer games provide this shared experience by constructing a **world state** at each host and exchanging any information necessary to maintain consistency between each host's state.

Depending on the game's network topology, discussed more in Chapter 6, "Network Topologies and Sample Games," there are various ways to create and enforce consistency between remote hosts' world states. One common method is to have a server transmit the state of the world to all connected clients. The clients receive this transmitted state and update their own world state accordingly. In this way, all players on client hosts eventually experience the same world state.

Assuming some kind of object-oriented game object model, the state of the world can be defined as the state of all game objects in that world. Thus, the task of transmitting the world state can be decomposed into the task of transmitting the state of each of those objects.

This chapter addresses the task of transmitting object state between hosts in an effort to maintain a consistent world state for multiple, remote players.

Replicating an Object

The act of transmitting an object's state from one host to another is known as **replication**. Replication requires more than just the serialization discussed in Chapter 4, "Object Serialization." To successfully replicate an object, a host must take three preparatory steps before serializing the object's internal state:

1. Mark the packet as a packet containing object state.
2. Uniquely identify the replicated object.
3. Indicate the class of the object being replicated.

First the sending host marks the packet as one containing object state. Hosts may need to communicate in ways other than object replication, so it is not safe to assume that each incoming datagram contains object replication data. As such, it is useful to create an enum `PacketType` to identify the type of each packet. Listing 5.1 gives an example.

Listing 5.1 `PacketType` Enum

```
enum PacketType
{
    PT_Hello,
    PT_ReplicationData,
```

```
    PT_Disconnect,
    PT_MAX
};
```

For every packet it sends, the host first serializes the corresponding `PacketType` into the packet's `MemoryStream`. This way, the receiving host can read the packet type immediately off each incoming datagram and then determine how to process it. Traditionally, the first packet exchanged between hosts is flagged as some kind of "hello" packet, used to initiate communication, allocate state, and potentially begin an authentication process. The presence of `PT_Hello` as the first byte in an incoming datagram signifies this type of packet. Similarly, `PT_Disconnect` as the first byte indicates a request to begin the disconnect process. `PT_MAX` is used later by code that needs to know the maximum number of elements in the packet type enum. To replicate an object, a sending host serializes `PT_ReplicationData` as the first byte of a packet.

Next, the sending host needs to identify the serialized object to the receiving host. This is so the receiving host can determine if it already has a copy of the incoming object. If so, it can update the object with the serialized state instead of instantiating a new object. Remember that the `LinkingContext` described in Chapter 4 already relies on objects having unique identifier tags. These tags can also identify objects for the purpose of state replication. In fact, the `LinkingContext` can be expanded, as shown in Listing 5.2, to assign unique network identifiers to objects that don't currently have them.

Listing 5.2 Enhanced `LinkingContext`

```
class LinkingContext
{
public:
    LinkingContext():
    mNextNetworkId(1)
    {}

    uint32_t GetNetworkId(const GameObject* inGameObject,
                          bool inShouldCreateIfNotFound)
    {
        auto it = mGameObjectToNetworkIdMap.find(inGameObject);
        if(it != mGameObjectToNetworkIdMap.end())
        {
            return it->second;
        }
        else if(inShouldCreateIfNotFound)
        {
            uint32_t newNetworkId = mNextNetworkId++;
            AddGameObject(inGameObject, newNetworkId);
            return newNetworkId;
        }
```

```
        else
        {
            return 0;
        }
    }

    void AddGameObject(GameObject* inGameObject, uint32_t inNetworkId)
    {
        mNetworkIdToGameObjectMap[inNetworkId] = inGameObject;
        mGameObjectToNetworkIdMap[inGameObject] = inNetworkId;
    }

    void RemoveGameObject(GameObject *inGameObject)
    {
        uint32_t networkId = mGameObjectToNetworkIdMap[inGameObject];
        mGameObjectToNetworkIdMap.erase(inGameObject);
        mNetworkIdToGameObjectMap.erase(networkId);
    }

    //unchanged...
    GameObject* GetGameObject(uint32_t inNetworkId);

private:
    std::unordered_map<uint32_t, GameObject*> mNetworkIdToGameObjectMap;
    std::unordered_map<const GameObject*, uint32_t>
         mGameObjectToNetworkIdMap;

    uint32_t mNextNetworkId;
}
```

The new member variable `mNextNetworkId` keeps track of the next unused network identifier, and increments each time one is used. Because it is a 4-byte unsigned integer, it is usually safe to assume it will not overflow: In cases where more than 4 billion unique replicated objects might be necessary over the duration of a game, you will need to implement a more complex system. For now, assume that incrementing the counter safely provides unique network identifiers.

When a host is ready to write `inGameObject`'s identifier into an object state packet, it calls `mLinkingContext->GetNetworkId(inGameObject, true)`, telling the linking context to generate a network identifier if necessary. It then writes this identifier into the packet after the `PacketType`. When the remote host receives this packet, it reads the identifier and uses its own linking context to look up the referenced object. If the receiving host finds an object, it can deserialize the data into it directly. If it does not find the object, it needs to create it.

For a remote host to create an object, it needs information regarding what class of object to create. The sending host provides this by serializing some kind of class identifier after the object

identifier. One brute force way to achieve this is to select a hardcoded class identifier from a set using dynamic casts, as show in Listing 5.3. The receiver would then use a switch statement like the one shown in Listing 5.4 to instantiate the correct class based on the class identifier.

Listing 5.3 Hardcoded, Tightly Coupled Class Identification

```
void WriteClassType(OutputMemoryBitStream& inStream,
                    const GameObject* inGameObject)
{
    if(dynamic_cast<const RoboCat*>(inGameObject))
    {
        inStream.Write(static_cast<uint32_t>('RBCT'));
    }
    else if(dynamic_cast<const RoboMouse*>(inGameObject))
    {
        inStream.Write(static_cast<uint32_t>('RBMS'));
    }
    else if(dynamic_cast<const RoboCheese*>(inGameObject))
    {
        inStream.Write(static_cast<uint32_t>('RBCH'));
    }
}
```

Listing 5.4 Hardcoded, Tightly Coupled Object Instantiation

```
GameObject* CreateGameObjectFromStream(InputMemoryBitStream& inStream)
{
    uint32_t classIdentifier;
    inStream.Read(classIdentifier);
    switch(classIdentifier)
    {
        case 'RBCT':
            return new RoboCat();
            break;
        case 'RBMS':
            return new RoboMouse();
            break;
        case 'RBCH':
            return new RoboCheese();
            break;
    }

    return nullptr;
}
```

Although this works, it is inadequate for several reasons. First, it uses a `dynamic_cast`, which usually requires C++'s built-in RTTI to be enabled. RTTI is often disabled in games because it

requires extra memory for every polymorphic class type. More importantly, this approach is inferior because it couples the game object system with the replication system. Every time you add a new gameplay class that may be replicated, you have to edit both the `WriteClassType` and `CreateGameObjectFromStream` functions in the networking code. This is easy to forget, and can cause the code to grow out of sync. Also, if you want to reuse your replication system in a new game, it requires completely rewriting these functions, which reference the gameplay code of your old game. Finally, the coupling makes unit testing more difficult, as tests cannot load the network unit without also loading the gameplay unit. In general, it is fine for gameplay code to depend on network code, but network code should almost never depend on gameplay.

One clean way to reduce the coupling between gameplay and network code is to abstract the object identification and creation routines from the replication system using an object creation registry.

Object Creation Registry

An **object creation registry** maps a class identifier to a function that creates an object of the given class. Using the registry, the network module can look up the creation function by id and then execute it to create the desired object. If your game has a reflection system, you probably already have such a system implemented, but if not, it is not difficult to create.

Each replicable class must be prepared for the object creation registry. First, assign each class a unique identifier and store it in a static constant named `kClassId`. Each class could use a GUID to ensure no overlap between identifiers, though 128-bit identifiers can be unnecessarily heavy considering the small subset of classes that need to be replicated. A good alternative is to use a four-character literal based on the name of the class and then check for conflicting names when the classes are submitted to the registry. A final alternative is to create class ids at compile time using a build tool which autogenerates the code to ensure uniqueness.

warning

Four-character literals are implementation dependent. Specifying 32-bit values with a literal using four characters like 'DXT5' or 'GOBJ' can be a simple way to come up with well-differentiated identifiers. They are also nice because they stick out clearly when present in a memory dump of your packets. For this reason, many third-party engines, from *Unreal* to C4, use them as markers and identifiers. Unfortunately, they are classified as implementation dependent in the C++ standard, which means not all compilers handle the conversion of a string literal into an integer in the same way. Most compilers, including GCC and Visual Studio, use the same convention, but if you are using multicharacter literals to communicate between processes compiled with different compilers, run some tests first to make sure both compilers translate the literals the same way.

Once each class has a unique identifier, add a `GetClassId` virtual function to `GameObject`. Override this function for each child class of `GameObject` so that it returns the identifier of the class. Finally, add a static function to each child class which creates and returns an instance of the class. Listing 5.5 shows how `GameObject` and two child classes should be prepared for the registry.

Listing 5.5 Classes Prepared for the Object Creation Registry

```
class GameObject
{
public:
    //...
    enum{kClassId = 'GOBJ'};
    virtual uint32_t GetClassId() const {return kClassId;}
    static GameObject* CreateInstance() {return new GameObject();}
    //...
};

class RoboCat: public GameObject
{
public:
    //...
    enum{kClassId = 'RBCT'};
    virtual uint32_t GetClassId() const {return kClassId;}
    static GameObject* CreateInstance() {return new RoboCat();}
    //...
};

class RoboMouse: public GameObject
{
    //...
    enum{kClassId = 'RBMS'};
    virtual uint32_t GetClassId() const {return kClassId;}
    static GameObject* CreateInstance() {return new RoboMouse();}
    //...
};
```

Note that each child class needs the `GetClassId` virtual function implemented. Even though the code looks identical, the value returned changes because the `kClassId` constant is different. Because the code is similar for each class, some developers prefer to use a preprocessor macro to generate it. Complex preprocessor macros are generally frowned on because modern debuggers do not handle them well, but they can lessen the chance of errors that come from copying and pasting code over and over. In addition, if the copied code needs to change, just changing the macro will propagate the changes through to all classes. Listing 5.6 demonstrates how to use a macro in this case.

Listing 5.6 Classes Prepared for the Object Creation Registry Using a Macro

```
#define CLASS_IDENTIFICATION(inCode, inClass)\
enum{kClassId = inCode}; \
virtual uint32_t GetClassId() const {return kClassId;} \
static GameObject* CreateInstance() {return new inClass();}

class GameObject
{
public:
    //...
    CLASS_IDENTIFICATION('GOBJ', GameObject)
    //...
};

class RoboCat: public GameObject
{
    //...
    CLASS_IDENTIFICATION('RBCT', RoboCat)
    //...
};
class RoboMouse: public GameObject
{
    //...
    CLASS_IDENTIFICATION('RBMS', RoboMouse)
    //...
};
```

The backslashes at the end of each line of the macro definition instruct the compiler that the definition continues to the following line.

With the class identification system in place, create an `ObjectCreationRegistry` to hold the map from class identifier to creation function. Gameplay code, completely independent from the replication system, can fill this in with replicable classes, as show in Listing 5.7. `ObjectCreationRegistry` doesn't technically have to be a singleton as shown, it just needs to be accessible from both gameplay and network code.

Listing 5.7 `ObjectCreationRegistry` Singleton and Mapping

```
typedef GameObject* (*GameObjectCreationFunc)();

class ObjectCreationRegistry
{
public:
    static ObjectCreationRegistry& Get()
    {
        static ObjectCreationRegistry sInstance;
        return sInstance;
    }
```

```
    template<class T>
    void RegisterCreationFunction()
    {
        //ensure no duplicate class id
        assert(mNameToGameObjectCreationFunctionMap.find(T::kClassId) ==
                mNameToGameObjectCreationFunctionMap.end());
        mNameToGameObjectCreationFunctionMap[T::kClassId] =
                T::CreateInstance;
    }

    GameObject* CreateGameObject(uint32_t inClassId)
    {
        //add error checking if desired- for now crash if not found
        GameObjectCreationFunc creationFunc =
            mNameToGameObjectCreationFunctionMap[inClassId];
        GameObject* gameObject = creationFunc();
        return gameObject;
    }

private:
    ObjectCreationRegistry() {}
    unordered_map<uint32_t, GameObjectCreationFunc>
        mNameToGameObjectCreationFunctionMap;
};

void RegisterObjectCreation()
{
    ObjectCreationRegistry::Get().RegisterCreationFunction<GameObject>();
    ObjectCreationRegistry::Get().RegisterCreationFunction<RoboCat>();
    ObjectCreationRegistry::Get().RegisterCreationFunction<RoboMouse>();
}
```

The `GameObjectCreationFunc` type is a function pointer which matches the signature of the `CreateInstance` static member functions in each class. The `RegisterCreationFunction` is a template used to prevent a mismatch between class identifier and creation function. Somewhere in the gameplay startup code, call `RegisterObjectCreation` to populate the object creation registry with class identifiers and instantiation functions.

With this system in place, when a sending host needs to write a class identifier for a `GameObject`, it just calls its `GetClassId` method. When the receiving host needs to create an instance of a given class, it simply calls `Create` on the object creation registry and passes the class identifier.

In effect, this system represents a custom-built version of C++'s RTTI system. Because it is hand built for this purpose, you have more control over its memory use, its type identifier size, and its cross-compiler compatibility than you would just using C++'s `typeid` operator.

> **tip**
>
> If your game uses a reflection system like the one described in the generalized serialization section of Chapter 4, you can augment that system instead of using the one described here. Just add a `GetDataType` virtual function to each `GameObject` which returns the object's `DataType` instead of a class identifier. Then add a unique identifier to each `DataType`, and an instantiation function. Instead of mapping from class identifier to creation function, the object creation registry becomes more of a **data type registry**, mapping from data type identifier to `DataType`. To replicate an object, get its `DataType` through the `GetDataType` method and serialize the `DataType`'s identifier. To Instantiate it, look up the `DataType` by identifier in the registry and then use the `DataType`'s instantiation function. This has the advantage of making the `DataType` available for generalized serialization on the receiving end of the replication.

Multiple Objects per Packet

Remember it is efficient to send packets as close in size to the MTU as possible. Not all objects are big, so there is an efficiency gain in sending multiple objects per packet. To do so, once a host has tagged a packet as a `PT_ReplicationData` packet, it merely repeats the following steps for each object:

1. Writes the object's network identifier
2. Writes the object's class identifier
3. Writes the object's serialized data

When the receiving host finishes deserializing an object, any unused data left in the packet must be for another object. So, the host repeats the receiving process until there is no remaining unused data.

Naïve World State Replication

With multi-object replication code in place, it is straightforward to replicate the entire world state by replicating each object in the world. If you have a small enough game world, like that of the original *Quake*, then the entire world state can fit entirely within a single packet. Listing 5.8 introduces a replication manager that replicates the entire world in this manner.

Listing 5.8 Replicating World State

```
class ReplicationManager
{
public:
    void ReplicateWorldState(OutputMemoryBitStream& inStream,
                             const vector<GameObject*>& inAllObjects);
```

```
private:
    void ReplicateIntoStream(OutputMemoryBitStream& inStream,
                             GameObject* inGameObject);

    LinkingContext* mLinkingContext;
};

void ReplicationManager::ReplicateIntoStream(
    OutputMemoryBitStream& inStream,
    GameObject* inGameObject)
{
    //write game object id
    inStream.Write(mLinkingContext->GetNetworkId(inGameObject, true));

    //write game object class
    inStream.Write(inGameObject->GetClassId());

    //write game object data
    inGameObject->Write(inStream);
}

void ReplicationManager::ReplicateWorldState(
    OutputMemoryBitStream& inStream,
    const vector<GameObject*>& inAllObjects)
{
    //tag as replication data
    inStream.WriteBits(PT_ReplicationData, GetRequiredBits<PT_MAX>::Value );

    //write each object
    for(GameObject* go: inAllObjects)
    {
        ReplicateIntoStream(inStream, go);
    }
}
```

`ReplicateWorldState` is a public function which a caller can use to write replication data for a collection of objects into an outgoing stream. It first tags the data as replication data and then uses the private `ReplicateIntoStream` to write each object individually. `ReplicateIntoStream` uses the linking context to write the network ID of each object and the virtual `GetClassId` to write the object's class identifier. It then depends on a virtual `Write` function on the game object to serialize the actual data.

GETTING THE REQUIRED BITS TO SERIALIZE A VALUE

Remember that the bit stream allows serialization of a field's value using an arbitrary number of bits. The number of bits must be large enough to represent the maximum value possible for the field. When serializing an enum, the compiler can actually calculate

the best number of bits at compile time, removing the chance for error when elements are added or removed from the enum. The trick is to make sure that the final element of the enum is always suffixed as a `_MAX` element. For instance, for the `PacketType` enum, it is named `PT_MAX`. This way, the value of the `_MAX` element will always increment or decrement automatically when elements are added or removed, and you have an easy way to track the maximum value for the enum.

The `ReplicateWorldState` method passes this final enum value as a template argument to `GetRequiredBits` to calculate the number of bits required to represent the maximum packet type. To do so most efficiently, at compile time, it uses something known as **template metaprogramming**, a somewhat dark art of C++ engineering. It turns out the language of C++ templates is so complex it is actually Turing universal, and a compiler can calculate any arbitrary function as long as the inputs are known at compile time. In this case, the code for calculating the number of bits required to represent a maximum value is as follows:

```
template<int tValue, int tBits>
struct GetRequiredBitsHelper
{

    enum {Value = GetRequiredBitsHelper<(tValue >> 1),
                  tBits + 1>::Value};
};

template<int tBits>
struct GetRequiredBitsHelper<0, tBits>
{

    enum {Value = tBits};
};

template<int tValue>
struct GetRequiredBits
{
    enum {Value = GetRequiredBitsHelper<tValue, 0>::Value};
};
```

Template metaprogramming has no explicit loop functionality, so it must use recursion in lieu of iteration. Thus, `GetRequiredBits` relies on the recursive `GetRequiredBitsHelper` to find the highest bit set in the argument value and thus calculate the number of bits necessary for representation. It does so by incrementing the `tBits` argument each time it shifts the `tValue` argument one bit to the right. When `tValue` is finally 0, the base case specialization is invoked, which simply returns the accumulated value in `tBits`.

With the advent of C++11, the `constexpr` keyword allows some of the functionality of template metaprogramming with less complexity, but at the time of writing it is not currently supported by all modern compilers (i.e., Visual Studio 2013) so it is safer to go with templates for compatibility.

When the receiving host detects a replication state packet, it passes it to the replication manager, which loops through each serialized game object in the packet. If a game object does not exist, the client creates it and deserializes the state. If a game object does exist, the client finds it and deserializes state into it. When the client is done processing the packet, it destroys any local game objects that did not have data in the packet, as the lack of data means the game object no longer exists in the world of the sending host. Listing 5.9 shows additions to the replication manager that allow it to process an incoming packet identified as containing replication state.

Listing 5.9 Replicating World State

```
class ReplicationManager
{
public:
    void ReceiveReplicatedObjects(InputMemoryBitStream& inStream);

private:
    GameObject* ReceiveReplicatedObject(InputMemoryBitStream& inStream);

    unordered_set<GameObject*> mObjectsReplicatedToMe;
};

void ReplicationManager::ReceiveReplicatedObjects(
    InputMemoryBitStream& inStream)
{
    unordered_set<GameObject*> receivedObjects;

    while(inStream.GetRemainingBitCount() > 0)
    {
        GameObject* receivedGameObject = ReceiveReplicatedObject(inStream);
        receivedObjects.insert(receivedGameObject);
    }

    //now run through mObjectsReplicatedToMe.
    //if an object isn't in the recently replicated set,
    //destroy it
    for(GameObject* go: mObjectsReplicatedToMe)
    {
        if(receivedObjects.find(go)!= receivedObjects.end())
        {
            mLinkingContext->Remove(go);
            go->Destroy();
        }
    }

    mObjectsReplicatedToMe = receivedObjects;
}

GameObject* ReplicationManager::ReceiveReplicatedObject(
    InputMemoryBitStream& inStream)
```

```
{
    uint32_t networkId;
    uint32_t classId;
    inStream.Read(networkId);
    inStream.Read(classId);

    GameObject* go = mLinkingContext->GetGameObject(networkId);
    if(!go)
    {
        go = ObjectCreationRegistry::Get().CreateGameObject(classId);
        mLinkingContext->AddGameObject(go, networkId);
    }

    //now read update
    go->Read(inStream);

    //return gameobject so we can track it was received in packet
    return go;
}
```

Once the packet receiving code reads the packet type and determines the packet is replication data, it can pass the stream to `ReceiveWorld`. `ReceiveWorld` uses `ReceiveReplicatedObject` to receive each object and tracks each received object in a set. Once all objects are received, it checks for any objects that were received in the previous packet that were not received in this packet and destroys them to keep the world in sync.

Sending and receiving world state in this manner is simple, but is limited by the requirement that the entire world state must fit within each packet. To support larger worlds, you need an alternate method of replicating state.

Changes in World State

Because each host maintains its own copy of the world state, it is not necessary to replicate the entire world state in a single packet. Instead, the sender can create packets that represent changes in world state, and the receiver can then apply these changes to its own world state. This way, a sender can use multiple packets to synchronize a very large world with a remote host.

When replicating world state in this manner, each packet can be said to contain a **world state delta**. Because the world state is composed of object states, a world state delta contains one **object state delta** for each object that needs to change. Each object state delta represents one of three replication actions:

1. Create game object
2. Update game object
3. Destroy game object

Replicating an object state delta is similar to replicating an entire object state, except the sender needs to write the object action into the packet. At this point, the prefix to serialized data is getting

so complex that it can be useful to create a **replication header** that incorporates the object's network identifier, replication action, and class if necessary. Listing 5.10 shows an implementation.

Listing 5.10 Replication Header

```
enum ReplicationAction
{
    RA_Create,
    RA_Update,
    RA_Destroy,
    RA_MAX
};

class ReplicationHeader
{
public:
    ReplicationHeader() {}

    ReplicationHeader(ReplicationAction inRA, uint32_t inNetworkId,
                      uint32_t inClassId = 0):
    mReplicationAction(inRA),
    mNetworkId(inNetworkId),
    mClassId(inClassId)
    {}

    ReplicationAction   mReplicationAction;
    uint32_t            mNetworkId;
    uint32_t            mClassId;

    void Write(OutputMemoryBitStream& inStream);
    void Read(InputMemoryBitStream& inStream);
};

void ReplicationHeader::Write(OutputMemoryBitStream& inStream)
{
    inStream.WriteBits(mReplicationAction, GetRequiredBits<RA_MAX>::Value );
    inStream.Write(mNetworkId);
    if( mReplicationAction!= RA_Destroy)
    {
        inStream.Write(mClassId);
    }
}

void ReplicationHeader::Read(InputMemoryBitStream& inStream)
{
    inStream.Read(mReplicationAction, GetRequiredBits<RA_MAX>::Value);
    inStream.Read(mNetworkId);
    if(mReplicationAction!= RA_Destroy)
    {
        inStream.Read(mClassId);
    }
};
```

The `Read` and `Write` methods aid in serializing the header into a packet's memory stream ahead of the object's data. Note that it is not necessary to serialize the object's class identifier in the case of object destruction.

When a sender needs to replicate a collection of object state deltas, it creates a memory stream, marks it as a `PT_ReplicationData` packet, and then serializes a `ReplicationHeader` and appropriate object data for each change. The `ReplicationManager` should have three distinct methods to replicate creation, update, and destruction, as shown in Listing 5.11. These encapsulate the `ReplicationHeader` creation and serialization so that they aren't exposed outside the `ReplicationManager`.

Listing 5.11 Replicating Sample Object State Deltas

```
ReplicationManager::ReplicateCreate(OutputMemoryBitStream& inStream,
                                    GameObject* inGameObject)
{
    ReplicationHeader rh(RA_Create,
                         mLinkingContext->GetNetworkId(inGameObject,
                                                       true),
                         inGameObject->GetClassId());
    rh.Write(inStream);
    inGameObject->Write(inStream);
}

void ReplicationManager::ReplicateUpdate(OutputMemoryBitStream& inStream,
                                         GameObject* inGameObject)
{
    ReplicationHeader rh(RA_Update,
                         mLinkingContext->GetNetworkId(inGameObject,
                                                       false),
                         inGameObject->GetClassId());
    rh.Write(inStream);
    inGameObject->Write(inStream);
}

void ReplicationManager::ReplicateDestroy(OutputMemoryBitStream&inStream,
                                          GameObject* inGameObject)
{
    ReplicationHeader rh(RA_Destroy,
                         mLinkingContext->GetNetworkId(inGameObject,
                                                       false));
    rh.Write(inStream);
}
```

When a receiving host processes a packet, it now must appropriately apply each action. Listing 5.12 shows how.

Listing 5.12 Processing Replication Actions

```
void ReplicationManager::ProcessReplicationAction(
    InputMemoryBitStream& inStream)
{
    ReplicationHeader rh;
    rh.Read(inStream);

    switch(rh.mReplicationAction)
    {
        case RA_Create:
        {
            GameObject* go =
            ObjectCreationRegistry::Get().CreateGameObject(rh.mClassId);
            mLinkingContext->AddGameObject(go, rh.mNetworkId);
            go->Read(inStream);
            break;
        }
        case RA_Update:
        {
            GameObject* go =
            mLinkingContext->GetGameObject(rh.mNetworkId);
            //we might have not received the create yet,
            //so serialize into a dummy to advance read head
            if(go)
            {
                go->Read(inStream);
            }
            else
            {
                uint32_t classId = rh.mClassId;
                go =
                ObjectCreationRegistry::Get().CreateGameObject(classId);
                go->Read(inStream);
                delete go;
            }
            break;
        }
        case RA_Destroy:
        {
            GameObject* go = mLinkingContext->GetGameObject(rh.mNetworkId);
            mLinkingContext->RemoveGameObject(go);
            go->Destroy();
            break;
        }
        default:
            //not handled by us
            break;
    }
}
```

After identifying a packet as one containing object state, the receiver loops through each header and chunk of serialized object data. If the header indicates creation, the receiver ensures that the object does not already exist. If it does not, it creates the object with the serialized data.

If the replication header indicates an object update, the receiver finds the object and deserializes the data into it. Due to any number of factors, including unreliability of the network, it is possible that the receiver might not find the target game object. In this case, the receiver still needs to process the rest of the packet, so it must advance the memory stream's read head by an appropriate amount. It can do this by creating a temporary dummy object, serializing the object state into the dummy, and then deleting the dummy object. If this is too inefficient, or not possible due to the way in which objects are constructed, you can add a field to the object replication header indicating the size of the serialized data. Then, the receiver can look up the size of the serialized data for the unlocatable object and advance the memory stream's current read head by that amount.

warning

Partial world and object state replication only work if the sender has an accurate representation of the receiver's current world state. This accuracy helps the sender determine which changes it needs to replicate. Because the Internet is inherently unreliable, this is not as simple as assuming that the receiver's world state is based on the latest packets transmitted by the sender. Either hosts need to send packets via TCP, so reliability is guaranteed, or they need to use an application level protocol designed on top of UDP to provide reliability. Chapter 7, "Latency, Jitter, and Reliability," addresses this topic.

Partial Object State Replication

When sending an object update, the sender might not need to send every property in the object. The sender may want to serialize only the subset of properties that have changed since the last update. To enable this, you can use a bit-field to represent the serialized properties. Each bit can represent a property or group of properties to be serialized. For instance, the `MouseStatus` class from Chapter 4 might use the enum in listing 5.13 to assign properties to bits.

Listing 5.13 `MouseStatus` Properties Enum

```
enum MouseStatusProperties
{
    MSP_Name      = 1 << 0,
    MSP_LegCount  = 1 << 1,
```

```
    MSP_HeadCount = 1 << 2,
    MSP_Health    = 1 << 3,
    MSP_MAX
};
```

These enum values can be bitwise ORed together to represent multiple properties. For instance, an object state delta containing values for `mHealth` and `mLegCount` would use `MSP_Health | MSP_LegCount`. Note that a bit-field containing a 1 for each bit indicates that all properties should be serialized.

The `Write` method of a class should be amended to take a property bit-field indicating which properties to serialize into the stream. Listing 5.14 provides an example for the `MouseStatus` class.

Listing 5.14 Using Property Bit-Fields to Write Properties

```
void MouseStatus::Write(OutputMemoryBitStream& inStream,
                        uint32_t inProperties)
{
    inStream.Write(inProperties, GetRequiredBits<MSP_MAX >::Value);
    if((inProperties & MSP_Name) != 0)
    {
        inStream.Write(mName);
    }
    if((inProperties & MSP_LegCount)!= 0)
    {
        inStream.Write(mLegCount);
    }
    if((inProperties & MSP_HeadCount) != 0)
    {
        inStream.Write(mHeadCount);
    }
    if((inProperties & MSP_Health)!= 0)
    {
        inStream.Write(mHealth);
    }
}
```

Before writing any properties, the method writes `inProperties` into the stream so that the deserialization procedure can read only the written properties. It then checks the individual bits of the bit-field to write the desired properties. Listing 5.15 demonstrates the deserialization process.

Listing 5.15 Deserialization of Partial Object Update

```
void MouseStatus::Read(InputMemoryBitStream& instream)
{
    uint32_t writtenProperties;
    inStream.Read(writtenProperties, GetRequiredBits<MSP_MAX>::Value);
    if((writtenProperties & MSP_Name) != 0)
    {
        inStream.Read(mName );
    }
    if((writtenProperties & MSP_LegCount) != 0)
    {
        inStream.Read(mLegCount);
    }
    if((writtenProperties & MSP_HeadCount) != 0)
    {
        inStream.Read(mHeadCount);
    }
    if((writtenProperties & MSP_Health) != 0)
    {
        inStream.Read(mHealth);
    }
}
```

The `Read` method first reads the `writtenProperties` field so it can use the value to deserialize only the correct properties.

This bit-field approach to partial object state replication also works with the more abstract, bidirectional, data-driven serialization routines given at the end of Chapter 4. Listing 5.16 extends that chapter's implementation of `Serialize` to support a bit-field for partial object state replication.

Listing 5.16 Bidirectional, Data-Driven Partial Object Update

```
void Serialize(MemoryStream* inStream, const DataType* inDataType,
               uint8_t* inData, uint32_t inProperties)
{
    inStream->Serialize(inProperties);

    const auto& mvs = inDataType->GetMemberVariables();
    for(int mvIndex = 0, c = mvs.size(); mvIndex < c; ++mvIndex)
    {
        if(((1 << mvIndex) & inProperties) != 0)
        {
            const auto& mv = mvs[mvIndex];
            void* mvData = inData + mv.GetOffset();
            switch(mv.GetPrimitiveType())
            {
```

```
                case EPT_Int:
                    inStream->Serialize(*reinterpret_cast<int*>(mvData));
                    break;
                case EPT_String:
                    inStream->Serialize(
                        *reinterpret_cast<string*>(mvData));
                    break;
                case EPT_Float:
                    inStream->Serialize(
                        *reinterpret_cast<float*>(mvData));
                    break;
            }
        }
    }
}
```

Instead of manually defining the meaning of each bit using an enum, the data-driven approach uses the index of the bit to represent the index of the member variable being serialized. Note that `Serialize` is called on the `inProperties` value right away. For an output stream, this will write the bit-field into the stream. However, for an input stream, this will read the written properties into the variable, overwriting anything that was passed in. This is correct behavior, as an input operation needs to use the serialized bit-field that corresponds to each of the serialized properties. If there are more than 32 potential properties to serialize, use a `uint64_t` for the properties. If there are more than 64 properties, consider grouping several properties under the same bit or splitting up the class.

RPCs as Serialized Objects

In a complex multiplayer game, a host might need to transmit something other than object state to another host. Consider the case of a host wanting to transmit the sound of an explosion to another host, or to flash another host's screens. Actions like this are best transmitted using **remote procedure calls**, or **RPCs**. A remote procedure call is the act of one host causing a procedure to execute on one or more remote hosts. There are many application-level protocols available for this, ranging from text-based ones like XML-RPC to binary ones like ONC-RPC. However, if a game already supports the object replication system described in this chapter, it is a simple matter to add an RPC layer on top of it.

Each procedure invocation can be thought of as a unique object, with a member variable for each parameter. To invoke an RPC on a remote host, the invoking host replicates an object of the appropriate type, with the member variables filled in correctly, to the target host. For instance, for the function `PlaySound`,

```
void PlaySound(const string& inSoundName, const Vector3& inLocation,
               float inVolume);
```

The `PlaySoundRPCParams` struct would have three member variables:

```
struct PlaySoundRPCParams
{
    string mSoundName;
    Vector3 mLocation;
    float mVolume;
};
```

To invoke `PlaySound` on a remote host, the invoker creates a `PlayerSoundRPCParams` object, sets the member variables, and then serializes the object into an object state packet. This can result in spaghetti code if many RPCs are used, as well as run through a lot of network object identifiers that aren't really necessary, as RPC invocation objects don't need to be uniquely identified.

A cleaner solution is to create a modular wrapper around the RPC system and then integrate it with the replication system. To do this, first add an additional replication action type, `RA_RPC`. This replication action identifies the serialized data that follows it as an RPC invocation, and allows the receiving host to direct it to a dedicated RPC processing module. It also tells the `ReplicationHeader` serialization code that a network identifier is not necessary for this action and thus should not be serialized. When the `ReplicationManager's ProcessReplicationAction` detects an `RA_RPC` action, it should pass the packet to the RPC module for further processing.

The RPC module should contain a data structure that maps from each RPC identifier to an unwrapping glue function that can deserialize parameters for and then invoke the appropriate function. Listing 5.17 shows a sample `RPCManager`.

Listing 5.17 An Example `RPCManager`

```
typedef void (*RPCUnwrapFunc)(InputMemoryBitStream&)

class RPCManager
{
public:
    void RegisterUnwrapFunction(uint32_t inName, RPCUnwrapFunc inFunc)
    {
        assert(mNameToRPCTable.find(inName) == mNameToRPCTable.end());
        mNameToRPCTable[inName] = inFunc;
    }

    void ProcessRPC(InputMemoryBitStream& inStream)
    {
        uint32_t name;
        inStream.Read(name);
        mNameToRPCTable[name](inStream);
    }
    unordered_map<uint32_t, RPCUnwrapFunc> mNameToRPCTable;
};
```

In this example, each RPC is identified with a four-character code unsigned integer. If desired, the `RPCManager` can use full strings instead: While strings allow for more variety, they use more bandwidth. Note the similarity to the object creation registry. Registering functions through a hash map is a common way to decouple seemingly dependent systems.

When the `ReplicationManager` detects the `RA_RPC` action, it passes the received memory stream to the RPC module for processing, which then unwraps and calls the correct function locally. To support this, game code must register an unwrap function for each RPC. Listing 5.18 shows how to register the `PlaySound` function.

Listing 5.18 Registering an RPC

```
void UnwrapPlaySound(InputMemoryBitStream& inStream)
{
    string soundName;
    Vector3 location;
    float volume;

    inStream.Read(soundName);
    inStream.Read(location);
    inStream.Read(volume);
    PlaySound(soundName, location, volume);
}

void RegisterRPCs(RPCManager* inRPCManager)
{
    inRPCManager->RegisterUnwrapFunction('PSND', UnwrapPlaySound);
}
```

`UnwrapPlaySound` is a glue function which handles the task of deserializing the parameters and invoking `PlaySound` with them. Gameplay code should invoke the `RegisterRPCs` function and pass it an appropriate `RPCManager`. Other RPCs can be added to the `RegisterRPCs` function as desired. Presumably the `PlaySound` function is implemented elsewhere.

Finally, to invoke an RPC, the caller needs a function to write the appropriate `ObjectReplicationHeader` and parameters into an outgoing packet. Depending on the implementation, it can either create the packet and send it, or check with the gameplay code or the networking module to see if any packet is already pending to go out to the remote host. Listing 5.19 gives an example of a wrapper function that writes an RPC call into an outgoing packet.

Listing 5.19 Writing `PlaySoundRPC` into a Pending Packet

```
void PlaySoundRPC(OutputMemoryBitStream& inStream,
                  const string&inSoundName,
                   const Vector3& inLocation, float inVolume)
{
    ReplicationHeader rh(RA_RPC);
    rh.Write(inStream);
    inStream.Write( inSoundName);
    inStream.Write(inLocation);
    inStream.Write(inVolume);
}
```

It can be a lot of work to manually generate the wrapping and unwrapping glue functions, register them with the `RPCManager`, and keep their parameters in sync with the underlying functions. For this reason, most engines that support RPCs use build tools to autogenerate the glue functions and register them with an RPC module.

> **note**
>
> Sometimes, a host may wish to remotely invoke a method on a specific object instead of just calling a free function. While similar, this is technically known as a **Remote Method Invocation**, or **RMI**, as opposed to a remote procedural call. A game that supports these could use the network identifier in the `ObjectReplicationHeader` to identify the target object of the RMI. An identifier of zero would indicate a free function RPC and a nonzero value would indicate an RMI on the specified game object. Alternatively, to conserve bandwidth at the expense of code size, a new replication action, `RA_RMI`, could indicate the relevance of the network identifier field, whereas the `RA_RPC` action would continue to ignore it.

Custom Solutions

No matter how many general-purpose object replication or RPC invocation tools an engine includes, some games still call for custom replication and messaging code. Either some desired functionality is not available, or, for certain rapidly changing values, the framework of generalized object replication is just too bulky and bandwidth inefficient. In these cases, you can always add custom replication actions by extending the `ReplicationAction` enum and adding cases to the switch statement in the `ProcessReplicationFunction`. By special casing the `ReplicationHeader` serialization for your object, you can include or omit the corresponding network identifier and class identifier as desired.

If your customization falls outside the purview of the `ReplicationManager` entirely, you can also extend the `PacketType` enum to create entirely new packet types and

managers to handle them. Following the design pattern of the registration maps used in the `ObjectCreationRegistry` and `RPCManager`, it is easy to inject higher-level code to handle these custom packets without polluting the lower-level networking system.

Summary

Replicating an object involves more than just sending its serialized data from one host to another. First, an application-level protocol must define all possible packet types, and the network module should tag packets containing object data as such. Each object needs a unique identifier, so that the receiving host can direct incoming state to the appropriate object. Finally, each class of object needs a unique identifier so that the receiving host can create an object of the correct class if one does not exist already. Networking code should not depend on gameplay classes, so use an indirect map of some sort to register replicable classes and creation functions with the network module.

Small-scale games can create a shared world between hosts by replicating each object in the world in each outgoing packet. Larger games cannot fit replication data for all objects in every packet, so they must employ a protocol that supports transmission of world state deltas. Each delta can contain replication actions to create an object, update an object, or destroy an object. For efficiency, update-object actions may send serialization data for only a subset of object properties. The appropriate subset depends on the overall network topology and reliability of the application-level protocol.

Sometimes, games need to replicate more than just object state data between hosts. Often, they need to invoke remote procedure calls on each other. One simple way to support RPC invocation is to introduce the RPC replication action and fold RPC data into replication packets. An RPC module can handle registration of RPC wrapping, unwrapping, and invocation functions, and the replication manager can channel any incoming RPC requests to this module.

Object replication is a key piece of the low-level multiplayer game tool chest, and will be a critical ingredient when implementing support for some of the higher-level network topologies described in Chapter 6.

Review Questions

1. What three key values should be in a packet replicating object state, other than the object's serialized data?
2. Why is it undesirable for networking code to depend on gameplay code?
3. Explain how to support creation of replicated objects on the receiving host without giving the networking code a dependency on gameplay classes.
4. Implement a simple game with five moving game objects in it. Replicate those objects to a remote host by sending the remote host a world state packet 15 times a second.

5. Regarding the game in Question 4, what problem develops as the number of game objects increase? What is a solution to this problem?
6. Implement a system which supports sending updates of only some of an object's properties to a remote host.
7. What is an RPC? What is an RMI? How are they different?
8. Using the chapter's framework, implement an RPC `SetPlayerName(const string& inName)` which tells other hosts the local player's name.
9. Implement a custom packet type that replicates which keys a player is currently holding down on the keyboard, using a reasonably efficient amount of bandwidth. Explain how to integrate this into this chapter's replication framework.

Additional Readings

Carmack, J. (1996, August). *Here Is the New Plan*. Retrieved from http://fabiensanglard.net/quakeSource/johnc-log.aug.htm. Accessed September 12, 2015.

Srinivasan, R. (1995, August). *RPC: Remote Procedure Call Protocol Specification Version 2*. Retrieved from http://tools.ietf.org/html/rfc1831. Accessed September 12, 2015.

Van Waveren, J. M. P. (2006, March 6). *The DOOM III Network Architecture*. Retrieved from http://mrelusive.com/publications/papers/The-DOOM-III-Network-Architecture.pdf. Accessed September 12, 2015.

Winer, Dave (1999, June 15). *XML-RPC Specification*. Retrieved from http://xmlrpc.scripting.com/spec.html. Accessed September 12, 2015.

CHAPTER 6

NETWORK TOPOLOGIES AND SAMPLE GAMES

The first part of this chapter takes a look at the two main configurations that can be used when multiple computers must communicate in a networked game: client-server and peer-to-peer. The remainder of the chapter combines all the topics covered up to this point in the book, and creates initial versions of two sample games.

Network Topologies

By and large, Chapters 1 to 5 have focused specifically on the issue of two computers communicating over the Internet and sharing information in a manner that is conducive to networked games. Although there absolutely are networked two-player games, many of the more popular games feature higher player counts. But even with only two players, some important questions arise. How will the players send game updates to each other? Will there be object replication as in Chapter 5, or will only the input state be replicated? What happens if the computers disagree on the game state? These are all important questions that must be answered for any networked multiplayer game.

A **network topology** determines how the computers in a network are connected to each other. In the context of a game, the topology determines how the computers participating in the game will be organized in order to ensure all players can see an up-to-date version of the game state. As with the decision of network protocol, there are tradeoffs regardless of the selected topology. This section explores the two main types of topologies used by games, client-server and peer-to-peer, and the small variations that can also exist within these types.

Client-Server

In a **client-server** topology, one game instance is designated the server, and all of the other game instances are designated as clients. Each client only ever communicates with the server, while the server is responsible for communicating with all of the clients. Figure 6.1 illustrates this topology.

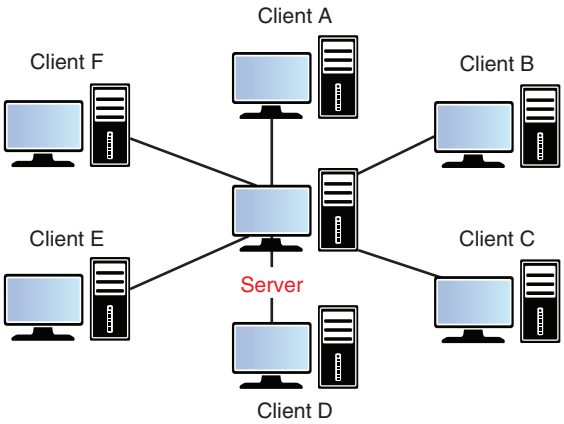

Figure 6.1 Client-server topology

In a client-server topology, given n clients there are a total of $O(2n)$ connections. However, it is asymmetric in that the server will have $O(n)$ connections (one to each client), while each client will only have one connection to the server. In terms of bandwidth, if there are n clients and each client sends b bytes per second of data, the server must have enough bandwidth to

handle $b \cdot n$ incoming bytes per second. Similarly, if the server needs to send c bytes per second of data to each client, the server must support $c \cdot n$ outgoing bytes per second. However, each client need only support c bytes per second downstream and b bytes per second upstream. This means that as the number of clients increase, the bandwidth required for the server will increase linearly. In theory, the bandwidth requirements for the client will not change based on the number of clients. However, in practice, supporting more clients leads to more objects in the world to replicate, which may lead to a slight increase in bandwidth for each client.

Although by no means the only approach to client-server, most games that implement client-server utilize an **authoritative** server. This means that the game server's simulation of the game is considered to be correct. If the client ever finds itself in disagreement with the server, it should update its game state based on what the server says is the game state. For instance, in the sample *Robo Cat Action* game discussed later in this chapter, each player cat can throw a ball of yarn. But with an authoritative server model, the client is forbidden from making a determination of whether or not the yarn hits another player. Instead, the client must inform the server that it wants to throw a ball of yarn. The server then decides both if the client is even allowed to throw a ball of yarn and, if so, whether or not the other player is hit by the ball of yarn.

By placing the server as an authority, this means there is some amount of "lag" or delay in actions performed by the client. The topic of latency is discussed in great detail in Chapter 7, "Latency, Jitter, and Reliability," but a brief discussion is in order. In the case of the ball throw, the server is the only game instance allowed to make a decision on what happens. But it will take some time to send a ball throw request to the server, which in turn will process it before sending the result to all of the clients. One contributing factor of this delay will be the **round trip time**, or **RTT**, which is the amount of time (typically expressed in milliseconds) that it takes for packets to travel to and back from a particular computer on the network. In an ideal scenario, this RTT is 100 ms or less, though even on modern Internet connections there are many factors that may not allow for such a low RTT.

Suppose there is a game with a server and two clients, Clients A and B. Because the server sends all game data to each client, this means that if Client A throws a ball of yarn, the packet containing the yarn throw request must first travel to the server. Then the server will process the throw before sending the result back to Clients A and B. In this scenario, the worst case network latency experienced by Client B would be equal to ½ Client A's RTT, plus the server processing time, plus ½ Client B's RTT. In fast network conditions, this may not be an issue, but realistically, most games must use a variety of techniques to hide this latency. This is covered in detail in Chapter 8, "Improved Latency Handling."

There is also a subclassification of types of servers. Some servers are **dedicated**, meaning they only run the game state and communicate with all of the clients. The dedicated server process is completely separate from any client processes running the game. This means that the dedicated server typically is headless and does not actually display any graphics. This type of server is often used by big-budget games such as *Battlefield*, which allows the developer to run multiple dedicated server processes on a single powerful machine.

The alternative to a dedicated server is a **listen server**. In this configuration, the server is also an active participant in the game itself. One advantage of a listen server configuration is that it may reduce deployment costs, because it is not necessary to rent servers in a data center—instead, one of the players can use their machine as both a server and a client. However, the disadvantage of a listen server is that a machine running as a listen server must be powerful enough and have a fast enough connection to handle this increased load. The listen server approach is sometimes erroneously referred to as a peer-to-peer connection, but a more precise term is peer *hosted*. There is still a server, it just so happens that the server is hosted by a player in the game.

One caution about a listen server is that assuming it is authoritative it will have a complete picture of the game state. This means that the player running the listen server could potentially use this information to cheat. Furthermore, in a client-server model typically only the server knows the network address of all of the active clients. This can be a huge issue in the event that the server disconnects—whether due to a network issue or perhaps an angry player deciding to exit their game. Some games that utilize a listen server implement the concept of **host migration**, which means that if a listen server disconnects, one of the clients is promoted to be the new server. In order for this to be possible, however, there must be some amount of communication between the clients. This means that host migration requires a hybrid model where there are both a client-server *and* a peer-to-peer topology.

Peer-to-Peer

In a **peer-to-peer** topology, each individual participant is connected to every other participant. As is apparent from Figure 6.2, this means that there is a great deal of data transmitted back and forth between clients. The number of connections is a quadratic function, or in other words,

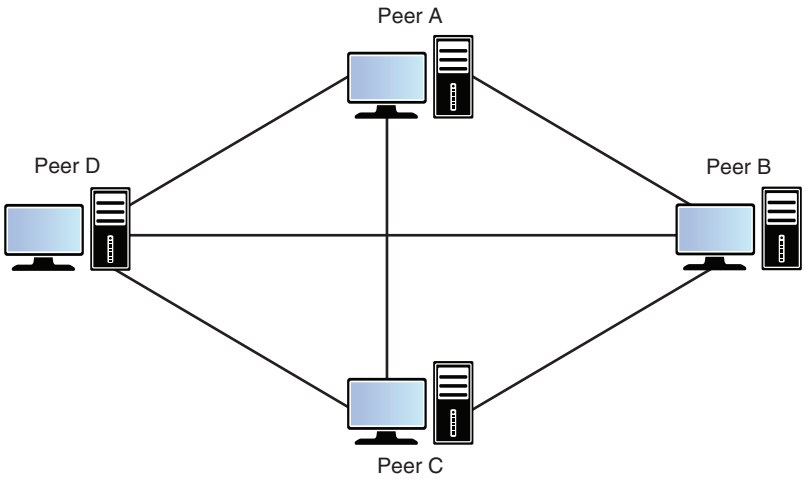

Figure 6.2 Peer-to-peer topology

given n peers, each peer must have $O(n-1)$ connections, which leads to $O(n^2)$ connections across the network. This also means that the bandwidth requirements for each peer increases as more and more peers connect to the game. However, unlike in client-server, the bandwidth requirements are symmetric, so every peer will require the same amount of available bandwidth upstream and downstream.

The concept of authority is much more nebulous in a peer-to-peer game. One possible approach is that certain peers have authority over certain parts of the game, but in practice such a system can be difficult to implement. A more common approach in peer-to-peer games is to share all actions across every peer, and have every peer simulate these actions. This model is sometimes called an **input sharing** model.

One aspect of the peer-to-peer topology that makes input sharing more viable is the fact that there is less latency to be concerned about. As opposed to the client-server model, which has an intermediary between clients, in a peer-to-peer game all peers are communicating with each other directly. This means that at worst, the latency between peers is ½ RTT. However, there still is some latency, which can lead to what is the largest technical challenge in a peer-to-peer game: ensuring that all peers remain synchronized with each other.

Recall that the discussion of the deterministic lockstep model in Chapter 1 presented one such approach. To recap, in the *Age of Empires* implementation, the game was broken down into "turns" of 200 ms. All input commands during these 200 ms are queued up, and when the 200 ms ends, the commands are sent to all of the peers. Furthermore, there is a one turn delay such that when each peer is displaying the results of turn 1, the commands are being queued to be executed on turn 3. Although this type of turn synchronization is conceptually simple, the actual implementation details can be far more complex. The *Robo Cat RTS* sample game, discussed later in this chapter, implements a very similar model.

Furthermore, it is important to ensure that the game state is consistent between all peers. This means that the game implementation needs to be fully deterministic. In other words, a given set of inputs must always result in the same outputs. A few important aspects of this include using checksums to verify consistency of the game state across peer and synchronizing random number generation across all peers, both topics that are covered in detail later in this chapter.

Another issue that arises in peer-to-peer is connecting new players. Since every peer must know the address of every other peer, in theory a new player could connect to any peer. However, matchmaking services that list available games typically only accept a single address—in this case, one peer may be selected as a so-called master peer, who is the only peer that greets new players.

Finally, the server disconnection problem that is a concern in server-client doesn't really exist in peer-to-peer. Typically, if communication is lost with a peer, the game may pause for a few seconds before removing the peer from the game. Once the peer is disconnected, the remaining peers can continue simulating the game.

Implementing Client-Server

Combining all of the concepts that have been covered to this point in the book, it is now possible to create an initial version of a networked game. This section discusses one such game, *Robo Cat Action*, a top-down game featuring cats competing to collect as many mice as possible, all while throwing balls of yarn at each other. The game is shown in action in Figure 6.3. This first version of this game code is in the `Chapter6/RoboCatAction` directory of the online code repository.

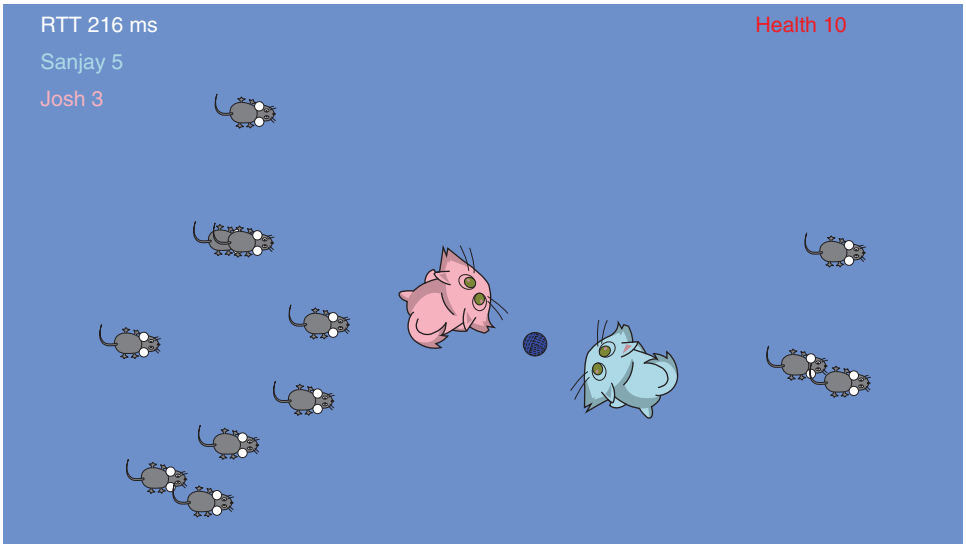

Figure 6.3 The initial version of *Robo Cat Action*

The controls for *Robo Cat Action* are not very complex. The D and A keys can be used to rotate the cat clockwise and counterclockwise, respectively. The W and S keys can be used to move the cat forward and back. The K key can be used to throw a ball of yarn that damages other cats. Mice can also be collected by moving over them.

This first version of the game code makes a large assumption: That there is little-to-no network latency, and that all packets will arrive at their destinations. This is clearly an unrealistic assumption for any networked game, and subsequent chapters, especially Chapter 7, "Latency, Jitter, and Reliability," discuss how to remove these assumptions. But for now, it is useful to discuss the basics of a client-server game without worrying about the added complexity of handling latency or packet loss.

Separating Server and Client Code

One of the cornerstones of the client-server model with an authoritative server is that the code that executes on the server is different from the code that executes on each client. Take the

example of the main character, the robo-cat. One of the properties of the cat is the `mHealth` variable that tracks its remaining health. The server needs to know about the health of the cat, because if the health hits zero, then the cat should go into its respawn state (cats have at least nine lives, after all). Similarly, the client needs to know how much health the cat has because it will display the remaining health in the top right corner. Even though the server's instance of `mHealth` is the authoritative version of the variable, the client will need to cache the variable locally in order to display it in the user interface.

The same can be said about functions. There may be some member functions of the `RoboCat` class that are needed only for the server, some that are needed only for the client, and some that are needed for both. To account for this, *Robo Cat Action* takes advantage of inheritance and virtual functions. Thus, there is a `RoboCat` base class and two derived classes: `RoboCatServer` and `RoboCatClient`, both of which override and implement new member functions as necessary. From a performance standpoint, using virtual functions in this manner may not give the highest possible performance, but from the perspective of ease-of-use, an inheritance hierarchy is perhaps the simplest.

The concept of splitting up the code into separate classes is taken a step further—inspecting the code will reveal that the code is separated into three separate targets. The first target is the RoboCat library that contains the shared code that is used by both the server and the client. This includes classes such as the `UDPSocket` class as implemented in Chapter 3 and the `OutputMemoryBitSteam` class as implemented in Chapter 4. Next, there are two executable targets—RoboCatServer for the server, and RoboCatClient for the client.

note

Because there are two separate executables for the server and client, in order to test *Robo Cat Action*, you must run both executables separately. The server takes a single command line parameter to specify the port to accept connections on. For example:

```
RoboCatServer 45000
```

This specifies that the server should listen for connecting clients on port 45000.

The client executable takes in two command line parameters: the full address of the server (including the port) and the name of the connecting client. So for instance:

```
RoboCatClient 127.0.0.1:45000 John
```

This specifies that the client wants to connect to the server at localhost port 45000, with a player name of "John." Naturally, multiple clients can connect to one server, and because the game does not use very many resources, multiple instances of the game can be run on one machine for the purposes of testing.

For the example of the `RoboCat` class hierarchy, the three individual classes reside in different targets—the base `RoboCat` class is in the shared library, and the `RoboCatServer` and `RoboCatClient` classes are unsurprisingly in their corresponding executable. This approach leads to a very clean separation of the code, and it makes it clear which code is specific to only the server or the client. To help visualize the approach, Figure 6.4 presents the class hierarchy for the `GameObject` class in *Robo Cat Action*.

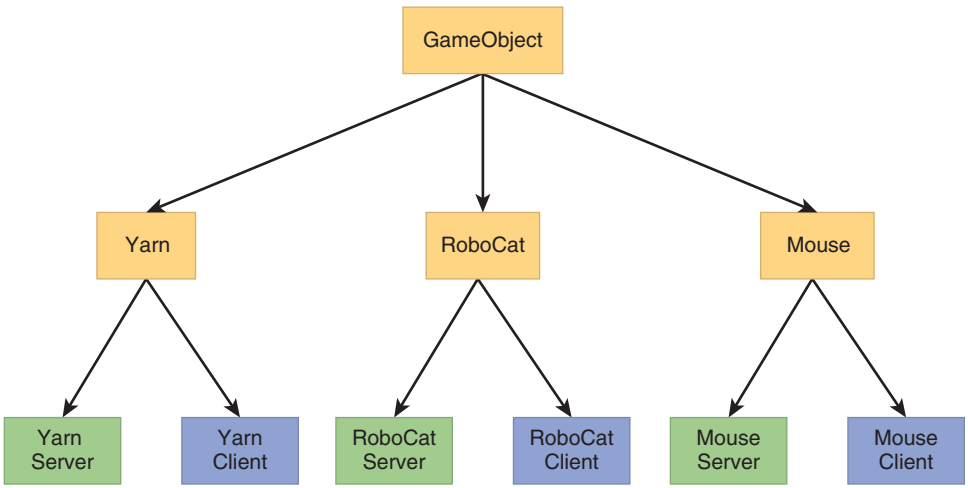

Figure 6.4 Hierarchy of the `GameObject` class in *Robo Cat Action* (items in gold are in the shared library, items in blue are in the client executable, and items in green are in the server executable)

Network Manager and Welcoming New Clients

The `NetworkManager` and the derived classes `NetworkManagerClient` and `NetworkManagerServer` do much of the heavy lifting in terms of interacting with the network. For example, all of the code that reads in available packets into a queue of packets to be processed is placed in the base `NetworkManager` class. The code to handle packets is very similar to what was covered in Chapter 3, "Berkeley Sockets," so it won't be covered again here.

One of the other responsibilities of the `NetworkManager` is to handle new clients joining the game. *Robo Cat Action* is designed for drop in/drop out multiplayer, so at any time a new client can try to join the match. As you might imagine, the responsibilities when welcoming new clients are different between the server and the client, and thus the functionality is split up between `NetworkManagerClient` and `NetworkManagerServer`.

Before we dive into the code, it's worthwhile to look at the connecting process at a high level. In essence, there are four steps to the procedure:

1. When a client wants to join a game, it sends the server a "hello" packet. This packet only contains the literal `"HELO"` (to identify the type of packet) and the serialized string representing the player's name. The client will keep sending these hello packets until it is acknowledged by the server.
2. Once the server receives the hello packet, it assigns a player ID to the new player, and also does some bookkeeping such as associating the incoming `SocketAddress` with the player ID. Then the server sends a "welcome" packet to the client. This packet contains the literal `"WLCM"` and the ID assigned to the player.
3. When the client receives the welcome packet, it saves its player ID, and starts sending and receiving replication information to the server.
4. At some point in the future, the server sends the information about any objects spawned for the new client to both the new client and all of the existing clients.

In this particular case, it is fairly straightforward to build redundancy into the system in the event of packet loss. If the client doesn't receive the welcome packet, it will continue sending hello packets to the server. If the server receives a hello packet from a client whose `SocketAddress` is already on file, it will simply resend the welcome packet.

Looking at the code more closely, there are two literals used to identify the packets, and so these are initialized as constants in the base `NetworkManager` class:

```
static const uint32_t kHelloCC = 'HELO';
static const uint32_t kWelcomeCC = 'WLCM';
```

Specifically on the client side of things, the `NetworkManagerClient` defines an enum to specify the current state of the client:

```
enum NetworkClientState
{
   NCS_Uninitialized,
   NCS_SayingHello,
   NCS_Welcomed
};
```

When the `NetworkManagerClient` is initialized, it sets its `mState` member variable to `NCS_SayingHello`. While in the `NCS_SayingHello` state, the client will keep sending hello packets to the server. On the other hand, if the client has been welcomed, then it needs to start sending updates to the server. In this case, the updates are input packets, which are covered shortly.

Furthermore, the client also knows the type of packets it is receiving based on the four-character literals that identify the packet. In the case of *Robo Cat Action*, there are only two types of packets it might receive: a welcome packet, and a state packet, which contains replication data. The code to handle sending and receiving packets is implemented in a manner similar to a state machine, as shown in Listing 6.1.

Listing 6.1 Client Sending and Receiving Packets

```
void NetworkManagerClient::SendOutgoingPackets()
{
    switch(mState)
    {
    case NCS_SayingHello:
        UpdateSayingHello();
        break;
    case NCS_Welcomed:
        UpdateSendingInputPacket();
        break;
    }
}

void NetworkManagerClient::ProcessPacket
(
    InputMemoryBitStream& inInputStream,
    const SocketAddress& inFromAddress
)
{
    uint32_t packetType;
    inInputStream.Read(packetType);
    switch(packetType)
    {
    case kWelcomeCC:
        HandleWelcomePacket(inInputStream);
        break;
    case kStateCC:
        HandleStatePacket(inInputStream);
        break;
    }
}
```

In terms of sending the hello packets, the only wrinkle is that the client ensures that it does not send hello packets too frequently. It does this by checking the elapsed time since the last hello packet. The actual packet itself is very straightforward, as the client need only to write the `'HELO'` literal and its name. Similarly, the welcome packet only contains the player ID as a payload, so the client only needs to save this ID. This code is shown in Listing 6.2. Notice how `HandleWelcomePacket` tests to ensure that the client is in the expected state for a welcome packet. This is to ensure no bugs can result in the event that a welcome packet is received after the client has already been welcomed. A similar test is used in `HandleStatePacket`.

Listing 6.2 Client Sending Hello Packets and Reading Welcome Packets

```
void NetworkManagerClient::UpdateSayingHello()
{
   float time = Timing::sInstance.GetTimef();

   if(time > mTimeOfLastHello + kTimeBetweenHellos)
   {
      SendHelloPacket();
      mTimeOfLastHello = time;
   }
}

void NetworkManagerClient::SendHelloPacket()
{
   OutputMemoryBitStream helloPacket;

   helloPacket.Write(kHelloCC);
   helloPacket.Write(mName);

   SendPacket(helloPacket, mServerAddress);
}

void NetworkManagerClient::HandleWelcomePacket(InputMemoryBitStream&
                                                inInputStream)
{
   if(mState == NCS_SayingHello)
   {
      //if we received a player id, we've been welcomed!
      int playerId;
      inInputStream.Read(playerId);
      mPlayerId = playerId;
      mState = NCS_Welcomed;
      LOG("'%s' was welcomed on client as player %d",
           mName.c_str(), mPlayerId);
   }
}
```

The server side of things is a bit more complex. First, the server has a hash map called `mAddressToClientMap` that it uses to track all known clients. The key for the map is the `SocketAddress`, and the value is a pointer to a `ClientProxy`. We'll discuss client proxies in more detail later in this chapter, but for now, you can think of it as a class that the server uses to track the state of all known clients. Keep in mind that because we are using the socket address directly, there could potentially be NAT traversal issues as previously discussed in Chapter 2. We will not worry about handling the traversal in the code for *Robo Cat*.

When the server first receives a packet, it performs a lookup into the address map to see whether or not the sender is known. If the sender is unknown, the server will then check to see if the packet is a hello packet. If the packet isn't a hello packet, it will simply be ignored.

Otherwise, the server will create a client proxy for the new client and send it a welcome packet. This is shown in Listing 6.3, though the code for sending a welcome packet is omitted as it is as straightforward as sending the hello packet.

Listing 6.3 Server Handling New Clients

```
void NetworkManagerServer::ProcessPacket
(
   InputMemoryBitStream& inInputStream,
   const SocketAddress& inFromAddress
)
{
   //do we know who this client is?
   auto it = mAddressToClientMap.find(inFromAddress);
   if(it == mAddressToClientMap.end())
   {
      HandlePacketFromNewClient(inInputStream, inFromAddress);
   }
   else
   {
      ProcessPacket((*it).second, inInputStream);
   }
}

void NetworkManagerServer::HandlePacketFromNewClient
(
   InputMemoryBitStream& inInputStream,
   const SocketAddress& inFromAddress
)
{
   uint32_t packetType;
   inInputStream.Read(packetType);
   if(packetType == kHelloCC)
   {
      string name;
      inInputStream.Read(name);

      //create a client proxy
      //...

      //and welcome the client...
      SendWelcomePacket(newClientProxy);

      //init replication manager for this client
      //...
   }
   else
   {
      LOG("Bad incoming packet from unknown client at socket %s",
           inFromAddress.ToString().c_str());
}
```

Input Sharing and Client Proxies

The implementation of replication for game objects in *Robo Cat Action* is very similar to the approach discussed in Chapter 5, "Object Replication." There are three replication commands: create, update, and destroy. Furthermore, a partial object replication system is implemented to reduce the amount of information sent in an update packet. Since the game uses an authoritative server model, objects are only ever replicated from the server to the client—thus the server is responsible for sending the replication update packets (assigned the literal `'STAT'`), and the client is responsible for processing the replication commands as necessary. There's a bit of work that needs to be done in order to ensure that the appropriate commands are sent to each of the clients, which will be covered later in this section.

For now, consider what the client needs to send to the server. Since the server is the authority, the client ideally should not be sending any replication commands for objects. However, in order for the server to accurately simulate each client, it needs to know what each client is trying to do. This leads to the concept of an input packet. In every frame, the client processes the input events. If any of these input events lead to something that needs to be processed server side—such as movement of the cat or throwing of a ball of yarn—the client will send the input events to the server. The server then accepts the input packet, and saves the input state into a **client proxy—**an object used by the server to track a particular client. Finally, when the sever updates the simulation, it will take into account any input stored in a client proxy.

The `InputState` class tracks a snapshot of the client input on a particular frame. Every frame, the `InputManager` class updates the `InputState` based on the client's input. What is stored in the `InputState` will vary from game to game. In this particular case, the only information stored is the desired movement offsets in each of the four cardinal directions, and whether or not the player pressed the button to throw a ball of yarn. This leads to a class with only a handful of members, as shown in Listing 6.4.

Listing 6.4 `InputState` Class Declaration

```
class InputState
{
public:
   InputState():
   mDesiredRightAmount(0),
   mDesiredLeftAmount(0),
   mDesiredForwardAmount(0),
   mDesiredBackAmount(0),
   mIsShooting(false)
   {}

   float GetDesiredHorizontalDelta() const
   {return mDesiredRightAmount - mDesiredLeftAmount;}
   float GetDesiredVerticalDelta() const
   {return mDesiredForwardAmount - mDesiredBackAmount;}
   bool IsShooting() const
   {return mIsShooting;}
```

```
    bool Write(OutputMemoryBitStream& inOutputStream) const;
    bool Read(InputMemoryBitStream& inInputStream);

private:
    friend class InputManager;
    float mDesiredRightAmount, mDesiredLeftAmount;
    float mDesiredForwardAmount, mDesiredBackAmount;
    bool mIsShooting;
};
```

The `GetDesiredHorizontalDelta` and `GetDesiredVerticalDelta` functions are helper functions that determine the overall offset on each axis. So for example, if the player holds both the A and D keys, the overall horizontal delta should be zero. The code for the `Read` and `Write` functions is not included in Listing 6.4—these functions just read and write the member variables to the provided memory bit stream.

Keep in mind that the `InputState` is updated every single frame by the `InputManager`. For most games, it would be impractical to send the `InputState` to the server at the same frequency. Ideally, the `InputState` over the course of several frames should be combined into a single move. To keep things simple, *Robo Cat Action* doesn't combine the `InputState` in any way—instead, every *x* seconds, it will grab the current `InputState` and save this as a `Move`.

The `Move` class is essentially a wrapper for the `InputState`, with the addition of two floats: one to track the timestamp of the `Move`, and one to track the amount of delta time between the current move and the previous move. This is shown in Listing 6.5.

Listing 6.5 `Move` Class

```
class Move
{
public:
    Move() {}
    Move(const InputState& inInputState, float inTimestamp,
         float inDeltaTime):
        mInputState(inInputState),
        mTimestamp(inTimestamp),
        mDeltaTime(inDeltaTime)
    {}

    const InputState& GetInputState() const {return mInputState;}
    float GetTimestamp() const {return mTimestamp;}
    float GetDeltaTime() const {return mDeltaTime;}
    bool Write(OutputMemoryBitStream& inOutputStream) const;
    bool Read(InputMemoryBitStream& inInputStream);
private:
    InputState mInputState;
```

```
    float mTimestamp;
    float mDeltaTime;
};
```

The `Read` and `Write` functions here will read and write both the input state and the timestamp from/to the provided stream.

> **note**
>
> Although the `Move` class is just a thin wrapper for `InputState` with additional time variables, the distinction is made in order to allow for cleaner code on a frame-to-frame basis. The `InputManager` polls the keyboard every frame, and saves the data into an `InputState`. Only when the client actually needs to create a `Move` does the timestamp matter.

Next, a series of moves is stored in a `MoveList`. This class contains, unsurprisingly, a list of moves, as well as the timestamp of the last move in the list. On the client side, when the client determines it should store a new move, it will add the move to the move list. Then the `NetworkManagerClient` will write out the sequence of moves into an input packet when it is time to do so. Note that the code for writing the sequence of moves optimizes the bit count by assuming that there will never be more than three moves to write at a time. It can make this assumption based on the constant factors that dictate the frequency of moves and input packets. The client code related to move lists is shown in Listing 6.6.

Listing 6.6 Client-Side Code for Move Lists

```
const Move& MoveList::AddMove(const InputState& inInputState,
                              float inTimestamp)
{
    //first move has 0 delta time
    float deltaTime = mLastMoveTimestamp >= 0.f ?
                      inTimestamp - mLastMoveTimestamp: 0.f;

    mMoves.emplace_back(inInputState, inTimestamp, deltaTime);
    mLastMoveTimestamp = inTimestamp;
    return mMoves.back();
}

void NetworkManagerClient::SendInputPacket()
{
    //only send if there's any input to send!
    MoveList& moveList = InputManager::sInstance->GetMoveList();

    if(moveList.HasMoves())
    {
        OutputMemoryBitStream inputPacket;
        inputPacket.Write(kInputCC);
```

```
        //we only want to send the last three moves
        int moveCount = moveList.GetMoveCount();
        int startIndex = moveCount > 3 ? moveCount - 3 - 1: 0;
        inputPacket.Write(moveCount - startIndex, 2);
        for(int i = startIndex; i < moveCount; ++i)
        {
           moveList[i].Write(inputPacket);
        }

        SendPacket(inputPacket, mServerAddress);
        moveList.Clear();
    }
}
```

Note that the code for `SendInputPacket` uses the array indexing operator on the `MoveList`. The `MoveList` internally uses a `deque` data structure, so this operation is constant time. In terms of redundancy, `SendInputPacket` really is not very fault tolerant. The client only ever sends the moves once. So for example, if an input packet contains a "throw" input command, but that packet never reaches the server, the client will never actually throw a ball of yarn. Clearly, this is not a tenable situation in a multiplayer game.

In Chapter 7, "Latency, Jitter, and Reliability," you will see how some redundancy can be added to the input packets. In particular, each move will be sent three times in order to give the server three opportunities to recognize the move. This adds a bit of complexity on the server side of things, because the server needs to recognize whether or not it has already processed a move when it receives it.

As previously mentioned, the client proxy is what the server uses to track the state of each client. Among one of the client proxy's most important responsibilities is that it contains a separate replication manager for each client. This allows the server to have a complete picture of what information it has or has not sent to each client. Since the server will most likely not send a replication packet to every client every frame, a separate replication manager for each client is necessary. This especially becomes important when redundancy is added, because it will allow the server to know the exact variables that need to be resent for a particular client.

Each client proxy also stores the socket address, name, and ID of each player. The client proxy is also where the move information for each client is stored. When an input packet is received, all of the moves associated with a client are added to the `ClientProxy` instance representing that client. Listing 6.7 shows a partial declaration of the `ClientProxy` class.

Listing 6.7 Partial Declaration of the `ClientProxy` Class

```
class ClientProxy
{
public:
```

```
    ClientProxy(const SocketAddress& inSocketAddress, const string& inName,
                int inPlayerId);
    // Functions omitted
    //...
    MoveList& GetUnprocessedMoveList() {return mUnprocessedMoveList;}
private:
    ReplicationManagerServer mReplicationManagerServer;
    // Variables omitted
    //...
    MoveList mUnprocessedMoveList;
    bool mIsLastMoveTimestampDirty;
};
```

Finally, the `RoboCatServer` class will use the unprocessed move data in its `Update` function, as shown in Listing 6.8. It is important to note that the delta time passed to each call of `ProcessInput` and `SimulateMovement` is based on the delta time between the moves, as opposed to the delta time of the server's frame. This is how the server can try to ensure the simulation stays as close to the client's actions as possible, even if it receives multiple moves in one packet. It also allows for the server and client to run at different frame rates. This can potentially add some complications for physics objects that must be simulated at set time steps. If this is the case for your game, you will want to lock the physics frame rate separate from other frame rates.

Listing 6.8 Updating the `RoboCatServer` Class

```
void RoboCatServer::Update()
{
    RoboCat::Update();
    // Code omitted
    //...

    ClientProxyPtr client = NetworkManagerServer::sInstance->
                                GetClientProxy(GetPlayerId());
    if( client )
    {
        MoveList& moveList = client->GetUnprocessedMoveList();
        for( const Move& unprocessedMove: moveList)
        {
            const InputState& currentState = unprocessedMove.GetInputState();
            float deltaTime = unprocessedMove.GetDeltaTime();
            ProcessInput(deltaTime, currentState);
            SimulateMovement(deltaTime);
        }

        moveList.Clear();
    }
    HandleShooting();
```

```
    // Code omitted
    //...
}
```

Implementing Peer-to-Peer

Robo Cat RTS is a real-time strategy game that supports up to four players. Each player is given a herd of three cats. Cats can be controlled by first left clicking to select a cat, and then right clicking on a target. If the target is a location, the cat will move to that location. If the target is an enemy cat, the cat will move into range of the enemy cat before beginning to attack. As in the action game, the cats attack each other by throwing balls of yarn. *Robo Cat RTS* is shown in action in Figure 6.5. The code for the initial version of the game is in `Chapter6/RoboCatRTS`.

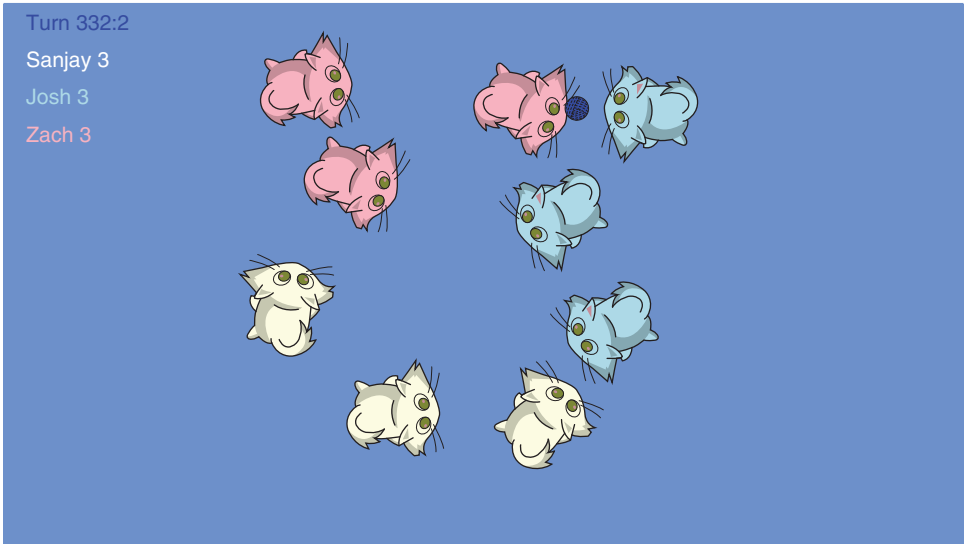

Figure 6.5 *Robo Cat RTS* in action

Although both games utilize UDP, the network model used for *Robo Cat RTS* is very different from *Robo Cat Action*. As with the action game, this initial version of the RTS assumes there is no packet loss. However, due to the nature of lockstep turns, the game will still function with some amount of latency—though there definitely is a degradation in the quality of the experience if the latency becomes too high.

Because *Robo Cat RTS* uses a peer-to-peer model, there is no need to separate the code into multiple projects. Each peer uses the same exact code. This reduces the number of files somewhat, and also means that the same executable is used by all players in the game.

> **note**
>
> There are two different ways to launch *Robo Cat RTS*, although both use the same executable. To initialize as a master peer, specify a port number and player name:
>
> ```
> RoboCatRTS 45000 John
> ```
>
> To initialize as a normal peer, specify the full address of the master peer (including the port number), as well as a player name:
>
> ```
> RoboCatRTS 127.0.0.1:45000 Jane
> ```
>
> Note that if the address specified is of a non-master peer, the player will still successfully connect, though it is faster if the master peer is specified.

However, *Robo Cat RTS* does employ the idea of a **master peer**. The primary purpose of the master peer is to provide a known IP address of a peer in the game. This is especially relevant when using a matchmaking service that maintains a list of known available games. Furthermore, the master peer is the only peer who is allowed to assign a player ID to a new player. This is mostly to avoid a race condition that could occur if multiple peers were contacted by two different new players simultaneously. Other than this one special case, the master peer behaves in the same manner as all of the other peers. Because each peer independently maintains the state of the entire game, the game can still continue if the master peer disconnects.

Welcoming New Peers and Game Start

The welcoming process for a peer-to-peer game is bit more complex than in a client-server game. As in *Robo Cat Action*, the new peer first sends a "hello" packet with their player name. However, the hello packet (`'HELO'`) can now have one of three responses:

1. **Welcome (`'WLCM'`)**—This means that the hello packet was received by the master peer, and the new peer is welcomed into the game. The welcome packet contains the new peer's player ID, the player ID of the master peer, and the number of players in the game (not including the new peer). Furthermore, the packet contains the names and IP addresses of all of the peers.
2. **Not joinable (`'NOJN'`)**—This means that either the game is already in progress, or the game is full. If the new peer receives this packet, the game exits.
3. **Not master peer (`'NOMP'`)**—This happens if the hello packet was sent to a peer who is not the master peer. In this instance, the packet will contain the address of the master peer so that the new peer can send a hello packet to the master peer.

However, once a new peer receives the welcome packet, the process is not complete. It is also the responsibility of the new peer to send an introduction packet (`"INTR"`) to every other peer in the game. This packet contains the new peer's player ID and name. This way, each peer in

the game is guaranteed to have the new peer stored in their data structures used to track the players in the game.

Because the addresses stored by each peer are based on addresses gleaned from incoming packets, there is potential for issue when one or more peer is connected on a local network. For example, suppose that Peer A is the master peer and Peer B is on the same local network as Peer A. This means that Peer A's map of peers will include the local network address of Peer B. Now suppose a new peer, Peer C, connects to Peer A via an external IP address. Peer A will welcome Peer C to the game, and give Peer C the address of Peer B. However, the address of Peer B that is provided is not reachable by Peer C, because Peer C is not on the same local network as Peer A and Peer B. Thus Peer C will fail to communicate with Peer B, and will not be able to properly join the game. This problem is shown in Figure 6.6a.

Recall that Chapter 2, "The Internet," described one such solution to this problem via NAT punchthrough. Other approaches involve the use of an external server in some way. In one approach, the external server, sometimes called a **rendezvous server**, only facilitates the initial connection between peers. In this way, it is guaranteed that every peer connects to every other peer via an externally reachable IP address. Use of a rendezvous server is illustrated in Figure 6.6b.

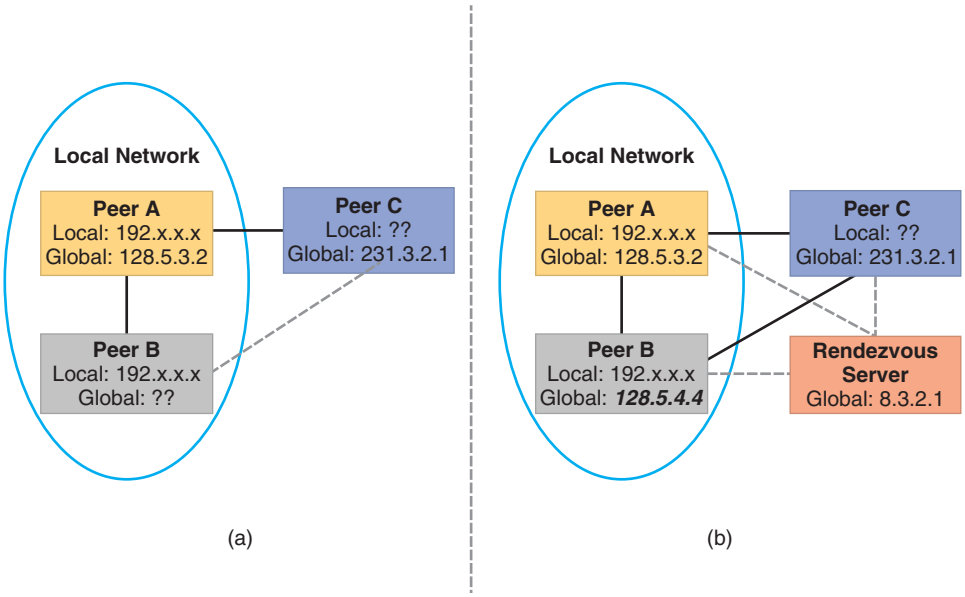

Figure 6.6 (a) Peer C is unable to connect to Peer B; (b) A rendezvous server facilitates initial communication between peers

Another approach used by some gaming services is to have a central server handle the entire packet routing between peers. What this means is that all peer traffic goes to the central server, and then is routed to the correct peer. Although this second approach requires a far more powerful server, it ensures that no peer will ever know the public IP address of any other peer. From a security standpoint, this may be preferred as it would, for instance, prevent one peer from trying to disconnect another peer via a distributed denial of service attack.

One other edge case worth considering is what should happen if a peer is only able to connect to some of the players in the game? This could happen even in the event of a rendezvous server or a central server routing packets. The simplest solution is to just not let this peer join the game, but you would need additional code to track this case. Since this chapter assumes that there are no connection issues to worry about, there is no code provided to handle this. But a commercial peer-to-peer game would absolutely need to include code to handle such a case.

When each peer is added to a game, their `NetworkManager` goes into a lobby state. When the master peer presses the return key, this will send a start packet (`'STRT'`) to every peer in the game. This will signal all peers to enter a 3-second countdown. Once the countdown hits zero, the game officially begins.

Note that this starting approach is naïve in that the timer does not really compensate for any latency between the master peer and the other peers. Thus, the master peer will always end up starting the game before the other peers. This does not affect the synchronization of the game due to the lockstep model, but it may mean that the master peer's game has to temporarily pause to allow the other peers to catch up. One way to solve this issue would be for each peer to subtract ½ RTT time from the timer duration. So if the master peer's RTT to Peer A were 100 ms, Peer A could subtract 50 ms from the total time duration, which should allow it to be better synchronized.

Command Sharing and Lockstep Turns

To simplify things, *Robo Cat RTS* runs at a locked 30 FPS, with a locked delta time of ~33 ms. This means that even if a particular peer takes greater than 33 ms to render a frame, the simulation still runs as if it were a 33-ms frame. *Robo Cat RTS* refers to each of these 33-ms ticks as a "sub-turn." There are three sub-turns per full turn. Thus, each full turn is 100 ms in length, or in other words, there are 10 turns per second. Ideally, the duration of sub-turns and full turns would be variable based on network and performance conditions. In fact, this is one of the topics of discussion in the Bettner and Terrano paper on *Age of Empires*. However, to keep things simple *Robo Cat RTS* never adjusts the length of turns or sub-turns.

In terms of replication, each peer runs a full simulation of the game world. This means that objects are not replicated in any way whatsoever. Instead, during gameplay only "turn" packets are transmitted. These packets contain a list of the commands issued by each peer on a particular turn, along with a couple of other key pieces of data.

It should be noted that there is a clear delineation between "commands" and input. For example, left clicking on a cat selects a particular cat. However, as this selection does not affect the game state in any way, it does not generate a command. On the other hand, if a cat is selected and the player right clicks, this means the player wants the cat to either move or attack. Since both of these actions would affect the game state, both will generate commands.

Furthermore, no command is executed the very instant it is issued. Rather, each peer queues up all commands issued on a particular turn. At the end of a turn, each peer will send its command list to every other peer. This command list is scheduled for execution on a *future* turn. Specifically, a command issued by a peer on turn x is not executed until turn $x + 2$. This allows for roughly 100 ms for turn packets to be received and processed by every peer. What this means is that in normal conditions, there is a delay of up to 200 ms from when a command is issued to when it is executed. However, because the delay is consistent, this does not really negatively affect the game experience, at least in the case of an RTS.

The concept of commands lends itself naturally to an inheritance hierarchy. Specifically, there is a base `Command` class, which is declared in Listing 6.9.

Listing 6.9 Declaration of the `Command` Class

```
class Command
{
public:
   enum ECommandType
   {
      CM_INVALID,
      CM_ATTACK,
      CM_MOVE
   };

   Command():
   mCommandType(CM_INVALID),
   mNetworkId(0),
   mPlayerId(0)
   {}

   //given a buffer, will construct the appropriate command subclass
   static shared_ptr<Command> StaticReadAndCreate(
                                    InputMemoryBitStream& inInputStream);

   //getters/setters
   //...

   virtual void Write(OutputMemoryBitStream& inOutputStream);
   virtual void ProcessCommand() = 0;
protected:
   virtual void Read(InputMemoryBitStream& inInputStream) = 0;
```

```
    ECommandType mCommandType;
    uint32_t mNetworkId;
    uint32_t mPlayerId;
};
```

The implementation of the `Command` class is mostly self-explanatory. There is an enum specifying the type of command, and an unsigned integer to store the network ID of the unit to whom the command was issued. The `ProcessCommand` pure virtual function is used when a command is actually executed. The `Read` and `Write` functions are used to read/write the commands to memory bit streams. The `StaticReadAndCreate` function first reads the command type enum from the memory bit stream. Then based on the value of the enum, it will construct an instance of an appropriate subclass and call the subclass' `Read` function.

There are only two subclasses in this case. The "move" command moves a cat to the targeted location. The "attack" command tells a cat to attack an enemy cat. In the case of the move command, it has an additional member variable that is a Vector3 containing the target of the move. Each subclass command also has a custom `StaticCreate` function that is used as a helper to create a `shared_ptr` to a command. The implementation `StaticCreate` and `ProcessCommand` for the move command is shown in Listing 6.10.

Listing 6.10 Select Functions from `MoveCommand`

```
MoveCommandPtr MoveCommand::StaticCreate(uint32_t inNetworkId,
                                         const Vector3& inTarget)
{
    MoveCommandPtr retVal;
    GameObjectPtr go = NetworkManager::sInstance->
                                    GetGameObject(inNetworkId);
    uint32_t playerId = NetworkManager::sInstance->GetMyPlayerId();

    //can only issue commands to this unit if I own it, and it's a cat
    if (go && go->GetClassId() == RoboCat::kClassId &&
        go->GetPlayerId() == playerId)
    {
        retVal = std::make_shared<MoveCommand>();
        retVal->mNetworkId = inNetworkId;
        retVal->mPlayerId = playerId;
        retVal->mTarget = inTarget;
    }
    return retVal;
}

void MoveCommand::ProcessCommand()
{
    GameObjectPtr obj = NetworkManager::sInstance->
                                     GetGameObject(mNetworkId);
```

```
    if (obj && obj->GetClassId() == RoboCat::kClassId &&
       obj->GetPlayerId() == mPlayerId)
    {
       RoboCat* rc = obj->GetAsCat();
       rc->EnterMovingState(mTarget);
    }
}
```

The `StaticCreate` function takes in the network ID of the cat who is receiving the command, as well as the target location. It also does some validation to ensure that the command is only being issued to a game object that exists, that the object is a cat, and that it is controlled by the peer issuing the command. The `ProcessCommand` function does some basic validation to ensure that the network ID it receives is that of a cat, and that the player ID corresponds to the player controlling the cat. The call to `EnterMovingState` simply tells the cat to start executing its moving behavior, which will occur over the course of one or more sub-turns. The moving state is implemented much like it would be in a single-player game, so it is not explained in this text.

Commands are stored in a `CommandList`. As with the `MoveList` class in the action game, the `CommandList` is just a wrapper for a `deque` of commands. It also has a `ProcessCommands` function that calls `ProcessCommand` on each command in the list.

Each peer's input manager has an instance of `CommandList`. When a local peer either uses the keyboard or mouse to request a command, the input manager adds the command to its list. A class called `TurnData` is used to encapsulate a peer's command list, as well as data related to synchronization, for each completed 100-ms turn. The network manager then has a `vector` where the index corresponds to a turn number. At each index, the network manager stores a map where the key is the player ID, and the value is the `TurnData` for that player. This way, for each turn, each player's turn data is separate. This is what allows the network manager to validate it has received data from each peer.

When each peer completes a sub-turn, it checks to see whether or not the full turn is over. If the turn is over, then it prepares turn packets to send to each peer. This function is a bit involved, and therefore is shown in Listing 6.11.

Listing 6.11 Sending Turn Packets to Each Peer

```
void NetworkManager::UpdateSendTurnPacket()
{
   mSubTurnNumber++;
   if (mSubTurnNumber == kSubTurnsPerTurn)
   {
      //create our turn data
      TurnData data(mPlayerId,
                    RandGen::sInstance->GetRandomUInt32(0, UINT32_MAX),
                    ComputeGlobalCRC(),
                    InputManager::sInstance->GetCommandList());
```

```
        //we need to send a turn packet to all of our peers
        OutputMemoryBitStream packet;
        packet.Write(kTurnCC);
        //we're sending data for 2 turns from now
        packet.Write(mTurnNumber + 2);
        packet.Write(mPlayerId);
        data.Write(packet);

        for (auto &iter: mPlayerToSocketMap)
        {
           SendPacket(packet, iter.second);
        }

        //save our turn data for turn + 2
        mTurnData[mTurnNumber + 2].emplace(mPlayerId, data);
        InputManager::sInstance->ClearCommandList();

        if (mTurnNumber >= 0)
        {
           TryAdvanceTurn();
        }
        else
        {
           //a negative turn means there's no possible commands yet
           mTurnNumber++;
           mSubTurnNumber = 0;
        }
    }
}
```

Two of the parameters passed to the `TurnData` constructor—the random value and the CRC—are discussed in the next section. The main item to note for now is that the peer prepares a turn packet that includes a list of all the commands to be executed two turns from now. This turn packet is then sent to all of the peers. Furthermore, the peer locally keeps its own turn data before clearing the input manager's command list.

Finally, there is code that checks for a negative turn number. When the game begins, the turn number is set to –2. This way, commands that are issued on turn –2 will be scheduled for execution on turn 0. This means that no commands are executed for the first 200 ms, but there is no way to avoid this initial delay—it is a property of the lockstep turn mechanism.

The `TryAdvanceTurn` function, shown in Listing 6.12, is named as such because it does not guarantee that the turn advances. This is because it is the responsibility of `TryAdvanceTurn` to enforce the lockstep nature of the turns. In essence, if it is currently turn *x*, `TryAdvanceTurn`

will only advance to turn $x + 1$ if all of the turn data for turn $x + 1$ has been received. If some turn data for turn $x + 1$ is still missing, the network manager will enter into a delay state.

Listing 6.12 `TryAdvanceTurn` Function

```
void NetworkManager::TryAdvanceTurn()
{
   //only advance the turn IF we received the data for everyone
   if (mTurnData[ mTurnNumber + 1].size() == mPlayerCount)
   {
      if (mState == NMS_Delay)
      {
         //throw away any input accrued during delay
         InputManager::sInstance->ClearCommandList();
         mState = NMS_Playing;
         //wait 100ms to give the slow peer a chance to catch up
         SDL_Delay(100);
      }

      mTurnNumber++;
      mSubTurnNumber = 0;

      if (CheckSync(mTurnData[mTurnNumber]))
      {
         //process all the moves for this turn
         for (auto& iter: mTurnData[mTurnNumber])
         {
            iter.second.GetCommandList().ProcessCommands(iter.first);
         }
      }
      else
      {
         //for simplicity, just end the game if it desyncs
         Engine::sInstance->SetShouldKeepRunning(false);
      }
   }
   else
   {
      //don't have all player's turn data, we have to delay:(
      mState = NMS_Delay;
   }
}
```

While in the delay state, objects in the world are not updated. Instead, the network manager will wait for the turn packets it still needs to receive. Every time a new turn packet is received while in delay, the network manager will again call `TryAdvanceTurn`, hoping that the new turn packet fills in the gap in turn data. This process will repeat until all necessary data is received. Similarly, if a connection is reset while in delay, the reset peer will be removed from the game and all other peers will attempt to continue.

Don't forget that this first version of *Robo Cat RTS* is assuming that all packets are eventually received. To account for packet loss, the delay state could be augmented so that while in delay, the peer determines whose command data is missing. It could then send a request to the peer in question to resend the command data. If several such resend requests are ignored, the peer would eventually be dropped. Furthermore, future turn packets could contain previous turn data, so in the event that a prior turn packet was dropped, a subsequent incoming turn packet may contain the required data.

Maintaining Synchronization

One of the largest challenges in designing a peer-to-peer game in which each peer simulates the game independently is ensuring that each instance of the game stays synchronized. Even minor discrepancies such as inconsistent positions can propagate into more serious issues in the future. If these discrepancies are allowed to persist, over time the simulations will diverge. At some point, the simulations may be so different that it seems like the peers are playing a different game! Clearly, this cannot be allowed, so ensuring and verifying synchronization is very important.

Synchronizing Pseudo-Random Number Generators

Some sources of desynchronization are more apparent than others. For example, using a **pseudo-random number generator** (**PRNG**) is the only way for a computer to acquire numbers that are seemingly random. Random elements are a cornerstone of many games, so eliminating random numbers altogether typically is not a viable option. However, in a peer-to-peer game, it is necessary to guarantee that on any particular turn, two peers will always receive the same results from a random number generator.

If you have ever used random numbers in a C/C++ program, you are likely familiar with the `rand` and `srand` functions. The `rand` function generates a pseudo-random number, while the `srand` function **seeds** the PRNG. Given a particular seed, a particular PRNG guarantees to always produce the same sequence of numbers. A typical approach is to use the current time as a seed passed to `srand`. In theory, this means that the numbers will be different every time.

In terms of keeping the peers in sync, this means there are two main things that need to be done in order to ensure each peer generates the same numbers:

- Each peer's random number generator should be seeded to the same initial value. In the case of *Robo Cat RTS*, the master peer selects a seed when it sends out the start packet. The seed is then included inside the start packet, so every peer will know what seed value to start the game with.
- It must be guaranteed that each peer will always make the same number of calls to the PRNG every turn, in the same order, and in the same location in the code. This means there cannot be different versions of the build that may use the PRNG more or less, such as for different hardware in cross-platform play.

However, there is a third issue that may not be apparent at first. It turns out that `rand` and `srand` are not particularly suitable for guaranteeing synchronization. The C standard does not specify which PRNG algorithm `rand` must use. This means that different implementations of the C library on different platforms (or even just in different compilers), are not guaranteed to use the same PRNG algorithm. If this is the case, it makes no difference whether or not the seeds are the same—different algorithms will give different results. Furthermore, because there are no guarantees regarding the PRNG algorithm used by `rand`, this means that the quality of the random numbers, or **entropy** of the values, is dubious.

In the past, the poorly defined nature of rand meant that most games implemented their own PRNG. Thankfully, C++11 introduced standardized and higher-quality pseudo-random number generators. Though the provided PRNGs are not considered cryptographically secure—meaning safe to use when random numbers are a part of security protocol—they are more than sufficient for the purposes of a game. Specifically, the code for *Robo Cat RTS* uses the C++11 implementation of the Mersenne Twister PRNG algorithm. The 32-bit Mersenne Twister, referred to as MT19937, has a period of 2^{19937}, meaning that the sequence of numbers will realistically never repeat during the course of a given game.

The interface for the C++11 random number generators is slightly more complex than the old `rand` and `srand` functions, so *Robo Cat RTS* wraps this in a `RandGen` class, as declared in Listing 6.13.

Listing 6.13 Declaration of the `RandGen` Class

```
class RandGen
{
public:
   static std::unique_ptr<RandGen> sInstance;

   RandGen();
   static void StaticInit();
   void Seed(uint32_t inSeed);
   std::mt19937& GetGeneratorRef() {return mGenerator;}

   float GetRandomFloat();
   uint32_t GetRandomUInt32(uint32_t inMin, uint32_t inMax);
   int32_t GetRandomInt(int32_t inMin, int32_t inMax);
   Vector3 GetRandomVector(const Vector3& inMin, const Vector3& inMax);
private:
   std::mt19937 mGenerator;
   std::uniform_real_distribution<float> mFloatDistr;
};
```

The implementation of a handful of the `RandGen` functions is likewise shown in Listing 6.14.

Listing 6.14 Select Functions from `RandGen`

```
void RandGen::StaticInit()
{
   sInstance = std::make_unique<RandGen>();
   //just use a default random seed, we'll reseed later
   std::random_device rd;
   sInstance->mGenerator.seed(rd());
}

void RandGen::Seed(uint32_t inSeed)
{
   mGenerator.seed(inSeed);
}

uint32_t RandGen::GetRandomUInt32(uint32_t inMin, uint32_t inMax)
{
   std::uniform_int_distribution<uint32_t> dist(inMin, inMax);
   return dist(mGenerator);
}
```

Note that when the `RandGen` is first initialized, it seeds using the `random_device` class. This will yield a platform-specific random value. Random devices are intended to be used for seeding a random number generator, but the device itself should not be used as a generator. The `uniform_int_distribution` class used in one of the functions simply is a way to specify a range of numbers, and receive a pseudo-random number within this range. This approach is preferable to the commonplace practice of taking an integer modulus of a random result. C++11 introduces several additional types of distributions.

To synchronize the random numbers, the master peer generates a random number to use as the new seed when the countdown begins. This random number is transmitted to all of the other peers to ensure that when turn –2 begins, all peers will have their generators seeded to the same value:

```
//select a seed value
uint32_t seed = RandGen::sInstance->GetRandomUInt32(0, UINT32_MAX);
RandGen::sInstance->Seed(seed);
```

Furthermore, when creating a turn packet at the end of a turn, each peer generates a random integer. This random integer is sent as part of the turn data inside the turn packet. This makes it easy for the peers to verify that all the random number generators remain in sync as the turns progress.

Keep in mind that if your game code requires random numbers that do not affect the game state in any way, it is possible to keep a different generator for these cases. One example is simulating random packet loss—this should not use the game's generator, because it means

every peer would simulate packet loss at the same time. However, be very careful when having multiple generators. You must make sure that any other programmers working on your game understand when to use which PRNG.

Verifying Game Synchronization

Other sources of desynchronization may not be as readily apparent as a PRNG. For example, while floating point implementations are deterministic, there can be discrepancies depending on the hardware implementations. For example, faster SIMD instructions may yield different results than regular floating point instructions. There typically are also different flags that can be set on a processor to change floating point behavior, such as whether or not it strictly follows the IEEE 754 implementation.

Other issues in synchronization may just be the result of an unintended error by a programmer. Perhaps the programmer wasn't aware how the synchronization worked, or perhaps they just made a mistake. Either way, it is important that the game has code that checks for synchronization on a regular basis. This way, desynchronization bugs can hopefully be caught soon after they are introduced.

A common approach is to utilize a **checksum**, much like how network packets use checksums in order to validate integrity of packet data. In essence, at the end of each turn, a checksum of the game state is computed. This checksum is transmitted inside the turn packet so that every peer can validate that all game instances arrive compute the same checksum at the end of a turn.

In terms of selecting an algorithm for the checksum, there are many different choices. *Robo Cat RTS* uses the **cyclic redundancy check** (**CRC**), which yields a 32-bit checksum value. Rather than implement a CRC function from scratch, this game uses the `crc32` function from the open-source zlib library. This was a matter of convenience, because zlib was already a dependency due to use of PNG image files. Furthermore, because zlib is designed to handle large amounts of data at once, it stands to reason that the CRC implementation is both vetted and performant.

In the spirit of further code reuse, the code for `ComputeGlobalCRC`, shown in Listing 6.15, uses the `OutputMemoryBitStream` class. Each game object in the world writes its relevant data into the provided bit stream via the `WriteForCRC` function. These objects are written in ascending order by network ID. Once every object has written its relevant data, the CRC is computed on the stream buffer as a whole.

Listing 6.15 `ComputeGlobalCRC` Function

```
uint32_t NetworkManager::ComputeGlobalCRC()
{
   OutputMemoryBitStream crcStream;

   uint32_t crc = crc32(0, Z_NULL, 0);
```

```
    for (auto& iter: mNetworkIdToGameObjectMap)
    {
       iter.second->WriteForCRC(crcStream);
    }

    crc = crc32(crc, reinterpret_cast<const Bytef*>
                (crcStream.GetBufferPtr()),
                 crcStream.GetByteLength());
    return crc;
}
```

There are a couple of additional items to consider regarding `ComputeGlobalCRC`. First, not every value for every game object is written into the stream. In the case of the `RoboCat` class, the values written are the controlling player ID, network ID, location, health, state, and target network ID. Some of the other member variables, such as the variable that tracks the cooldown between yarn throws, are not synchronized. This selectivity reduces the amount of time spent computing the CRC.

Furthermore, because the CRC function can be computed partially, it is not actually necessary to write all the data to a stream prior to computing the CRC. In fact, copying the data may be less efficient than computing the CRC of every value on the fly. It even would be possible to write an interface similar to `OutputMemoryBitStream`—essentially an instance of a class that only computes the CRC of values fed into it, but does not save it into a memory buffer. However, to keep the code simple, the existing `OutputMemoryBitStream` class was reused.

Back to the task at hand, recall that the `TryAdvanceTurn` function in Listing 6.12 makes a call to a `CheckSync` function when it advances the turn. This function loops through all of the random numbers and CRC values in every peer's turn data, and ensures that every peer computed the same random number and same CRC value when the turn packet was sent out.

In the event that `CheckSync` detects a desynchronization, *Robo Cat RTS* simply ends the game immediately. A more robust system would be to utilize a form of voting. Suppose there are four players in the game. In the event that players 1 to 3 computed checksum value A and player 4 computed checksum value B, this means that three of the players are still in sync. Thus it may be possible for the game to continue if player 4 is dropped from the game.

warning

When developing a peer-to-peer game with independent simulation, desynchronization is the source of much angst. Desynchronization bugs often are the most difficult to troubleshoot. In order to help debug these bugs, it is important to implement a logging system that can be enabled in order to see the commands executed by each peer in excruciating detail.

> While developing the sample code for *Robo Cat RTS*, a desynchronization would occur if a client went into the delay state while a cat was moving. It turned out that the cause was that when the delayed player resumed the game, they would skip a sub-turn. This was determined, thanks to a logging system that wrote when a peer was executing a sub-turn as well as the location of each cat at the end of each a sub-turn. This made it possible to see that one of the peers was skipping a sub-turn. Without the logging, it would have been much more time consuming to locate and fix this issue.

In the same scenario, a much more complex approach would be to actually replicate the entire game state back to player 4 in an effort to resynchronize them with the game. Depending on the amount of data in a game, this may be impractical. But it is something to keep in mind if it is important that players are not dropped from the game when they desync.

Summary

Selecting a network topology is one of the most important decisions made when creating a networked game. In the client-server topology, one game instance is denoted as the server, and it is generally the authority of the entire game. All other game instances are clients, and only communicate with the server. This usually means that object replication data is sent from the server to the client. In a peer-to-peer topology, each game instance is more or less on equal footing. One approach in a peer-to-peer game is to have each peer simulate the game independently.

The deep dive of the code for *Robo Cat Action* covered several different topics. To help modularize the code, the code was split up into three separate targets: a shared library, a server, and a client. The process of the server welcoming new clients involves transmission of a hello packet to the server, and a welcome packet back to the client. The input systems has the client sending input packets that contain "moves" executed by a client, including moving a cat around and throwing balls of yarn. The server maintains a client proxy for each client, both in order to track what replication data needs to be sent to each client and to have an object in which pending moves can be stored.

The section on *Robo Cat RTS* discussed many of the major challenges in designing a peer-to-peer game with independent simulation. The use of a master peer allows for a known IP address to be associated with a particular game. Each peer maintains a list of the addresses of all the other peers in the game. The welcoming of new peers is a bit more complex than a client-server game, because the new peer needs to inform all other peers of their existence. The peers maintain a lockstep by transmitting turn packets at the end of each 100-ms turn. The commands in these turn packets are scheduled for execution two turns later. Each peer continues on to the next turn only after all of turn data for the next turn has been received.

Finally, synchronizing random number generation and using checksums of the game state are necessary to keep each peer's game instance in sync.

Review Questions

1. In the client-server model, how do the responsibilities of a client differ from the responsibilities of the server?
2. What is the worst possible latency in a client-server game, and how does it compare to the worst possible latency in a peer-to-peer game?
3. How many connections does a peer-to-peer game require in comparison to a client-server game?
4. What is one approach to simulating the game state in a peer-to-peer game?
5. The current implementation of *Robo Cat Action* does not average the input state over several frames when creating a move. Implement this functionality.
6. In what manner could the start procedure be improved in *Robo Cat RTS*? Implement this improvement.

Additional Reading

Bettner, Paul and Mark Terrano. *1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond*. Presented at the Game Developer's Conference, San Francisco, CA, 2001.

This page intentionally left blank

CHAPTER 7

LATENCY, JITTER, AND RELIABILITY

Networked games live in a harsh environment, competing for bandwidth on aging networks, sending packets to servers and clients scattered throughout the world. This results in data loss and delay not typically experienced during development on a local network. This chapter explores some of the networking problems multiplayer games face and suggests workarounds and solutions for those problems, including how to build a custom reliability layer on top of the UDP transport protocol.

Latency

Your game, once released into the wild, must contend with a few negative factors not present in the tightly controlled environment of your local network. The first of these factors is **latency**. The word latency has different meanings in different situations. In the context of computer games, it refers to the amount of time between an observable cause and its observable effect. Depending on the type of game, this can be anything from the period between a mouse click and a unit responding to its orders in a real-time strategy (RTS) game, to the period between a user moving her head and a virtual reality (VR) display updating in response.

Some amount of latency is unavoidable, and different genres of games have different latency acceptability thresholds. VR games are typically the most sensitive to latency, because we as humans expect our eyes to see different things as soon as our heads swivel. In these cases, a latency of less than 20 ms is typically required for the user to remain present in the simulated reality. Fighting games, first-person shooters, and other twitchy action games are the next most sensitive to latency. Latency in these games can range from 16 to 150 ms before the user starts to feel, regardless of frame rate, that the game is sluggish and unresponsive. RTS games have the highest tolerance for latency, and this tolerance is often exploited to good effect, as described in Chapter 6, "Network Topologies and Sample Games." Latency in these games can grow as high as 500 ms without being detrimental to the user experience.

As a game engineer, decreasing latency is one manner in which you can improve your users' play experience. To do so, it helps to understand the many factors that contribute to this latency in the first place.

Non-Network Latency

It is a commonly held misconception that networking delay is the primary source for latency in gameplay. While packet exchange over the network is a significant source for latency, it is definitely not the only one. There are at least five other sources of latency, some of which are not under your control:

- **Input sampling latency.** The time between when a user pushes a button and when the game detects that button press can be significant. Consider a game running at 60 frames per second that polls a gamepad for input at the beginning of each frame, then updates all objects accordingly before finally rendering the game world. As shown in Figure 7.1a, if a user presses the jump button 2 ms after the game checks for input, it will be almost an entire frame before the game updates anything based on that button press. For inputs that drive view rotation, it is possible to sample the input again at the end of a frame and mildly warp the rendered output based on altered rotation, but this is typically only done in the most latency-sensitive applications. That means that on average, there is half a frame of latency between a button press and the game's response to that press.
- **Render pipeline latency.** GPUs do not perform draw commands the moment the CPU batches those commands. Instead, the driver inserts the commands into a command

buffer, and the GPU executes those commands at some point in the future. If there is a lot of rendering to do, the GPU may lag an entire frame behind the CPU before it displays the rendered image to the user. Figure 7.1b shows such a timeline common in single-threaded games. This introduces another frame of latency.

- **Multithreaded render pipeline latency.** Multithreaded games introduce even more latency into the render pipeline. Typically, one or more threads run the game simulation, updating a world that they pass to one or more render threads. These render threads then batch GPU commands while the simulation threads are already simulating the next frame. Figure 7.1c demonstrates how multithreaded rendering can add yet another frame of latency to the user experience.

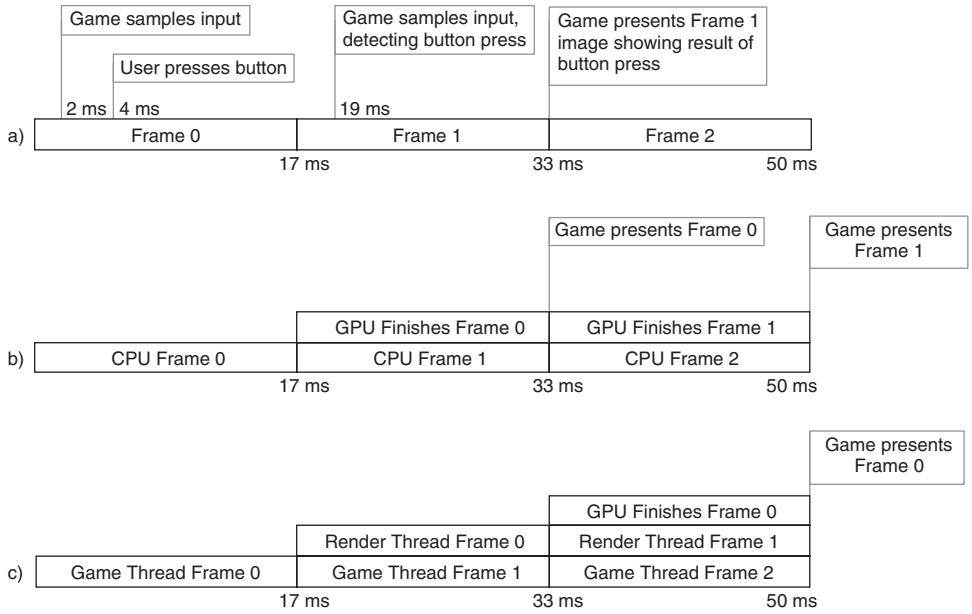

Figure 7.1 Latency timing diagrams

- **VSync.** To avoid screen tearing, it is common practice to change the image displayed by a video card only during a monitor's vertical blanking interval. This way, the monitor will never show part of one frame and part of the next frame at the same time. This means that a present call on the GPU must wait until the vertical blanking interval for the user's monitor, which is typically once every 1/60 of a second. If your game's frames take only 16 ms, this is not a problem. However, if a frame takes even 1 ms longer to render, the rendering will not be complete by the time the video card is ready to update the display. In this case, the command to present the back buffer to the front will stall, waiting an extra 15 ms until the next vertical blanking interval. When this happens, your user experiences an extra frame of latency.

> **note**
>
> Screen tearing is what happens when a GPU presents a back buffer while the monitor is in the middle of refreshing the image on its screen. Monitors typically update the image on screen one horizontal row at a time, from top bottom. If the image being drawn to the screen changes in the middle of the update, the user observes the bottom half of the screen showing the new image while the top half still shows the previous image. If the camera is scrolling rapidly across the world, this can result in a shearing effect that makes it look as if the image were printed on a piece of paper, torn in half, and then one-half slightly shifted.
>
> Most PC games give the user an option to disable vsync for enhanced performance, and some newer LCD monitors, known as G-SYNC, actually have variable refresh rates that can adjust to match frame rate and avoid the potential latency of vsyncing.

- **Display lag.** Most HDTVs and LCD monitors process their input to some extent before actually displaying an image. This processing can include de-interlacing, HDCP as well as other DRM, and image effects like video scaling, noise cancellation, adaptive luminance, image filtering, and more. This processing comes at a cost, and can easily add tens of milliseconds to the latency users perceive. Some televisions have a "game" mode that decreases video processing to minimize latency, but you cannot count on this to be enabled.
- **Pixel response time.** LCD displays have the additional problem that pixels just take time to change brightness. Typically this duration is in the single digits of milliseconds, but with older displays, this can easily add an extra half frame of latency. Luckily, the latency here presents more as image ghosting than absolute latency—the transition starts right away, but doesn't complete for several milliseconds.

Non-network latency is a serious issue and can negatively impact a user's perception of a game. John Carmack famously once tweeted "I can send an IP packet to Europe faster than I can send a pixel to the screen. How f'd up is that?" Given the amount of latency already present in a single-player game, there is strong pressure to mitigate any network-influenced latency as much as possible when introducing multiplayer functionality. To do that, it helps to understand the root causes of network latency.

Network Latency

Although there are many sources of latency, the delay experienced by a packet as it travels from a source host to its destination is usually the most significant cause of latency in multiplayer gaming. There are four main delays a packet experiences during its lifetime:

1. **Processing delay.** Remember that a network router works by reading packets from a network interface, examining the destination IP address, figuring out the next machine

that should receive the packet, and then forwarding it out of an appropriate interface. The time spent examining the source address and determining an appropriate route is known as the processing delay. Processing delay can also include any extra functionality the router provides, such as NAT or encryption.

2. **Transmission delay.** For a router to forward a packet, it must have a link layer interface that allows it to forward the packet along some physical medium. The link layer protocol controls the average rate at which bits can be written to the medium. For instance, a 1-MB Ethernet connection allows for roughly 1 million bits to be written to an Ethernet cable per second. Thus it takes about one millionth of a second (1 µs) to write a bit to a 1-MB Ethernet cable, and therefore 12.5 ms to write a whole 1500-byte packet. This time spent writing the bits to physical medium is known as the transmission delay.
3. **Queuing delay.** A router can only process a limited number of packets at a time. If packets arrive at a rate faster than the router can process them, they go into a receive queue, waiting to be processed. Similarly, a network interface can only output one packet at a time, so after a packet is processed, if the appropriate network interface is busy, it goes into a transmission queue. The time spent in these queues is known as the queuing delay.
4. **Propagation delay.** For the most part, regardless of the physical medium, information cannot travel faster than the speed of light. Thus, the latency when sending a packet is at least 0.3-ns times the number of meters the packet must travel. This means, even under ideal conditions, it takes at least 12 ms for a packet to travel across the United States. This time spent traveling is known as the propagation delay.

You can optimize some of these delays, and some you cannot. Processing delay is typically a minor factor, as most router processors these days are very fast.

Transmission delay is usually dependent on the type of link layer connection the end user employs. Because bandwidth capability typically increases as the packet moves closer to the backbone of the Internet, it is at the edges of the Internet where transmission delay is greatest. Making sure your servers use high-bandwidth connections is most important. After that, you can best reduce transmission delay by encouraging end users to upgrade to fast Internet connections. Sending packets that are as large as possible will also help, since you reduce the amount of bytes spent on headers. If those headers are a significant portion of your packet size, they translate to a significant portion of your transmission delay.

Queuing delay is a result of packets backing up waiting to be transmitted or processed. Minimizing processing and transmission delay will help minimize queuing delay. It's worth noting that because typical routing requires examining only the header of a packet, you can decrease the queuing delay for all your packets by sending few large packets instead of many small packets. For instance, a packet with a 1400-byte payload typically experiences as much processing delay as a packet with a 200-byte payload. If you send seven 200-byte packets, the final packet will have to sit in the queue during the processing of the six prior packets and thus will experience more cumulative network delay than a single large packet.

Propagation delay is often a very good target for optimization. Because it is based on the length of wire between hosts exchanging data, the best way to optimize it is to move the hosts closer together. In peer-to-peer games, this means prioritizing geographical locality when match making. In client-server games, this means making sure game servers are available close to your players. Be aware that sometimes physical locality isn't enough to ensure low-propagation delay: Direct connections between locations may not exist, requiring routers to route traffic in roundabout paths instead of via a straight line. It is important to take existing and future routes into account when planning the locations of your game servers.

> **note**
>
> In some cases, it is not feasible to disperse game servers throughout a geographical area, because you want all players on an entire continent to be able to play with each other. Riot games famously encountered such a situation with their title, *League of Legends*. Because dispersing game servers throughout the country was not an option, they took the reverse approach and built their own network infrastructure, peering with ISPs across North America to ensure they could control traffic routes and reduce network latency as much as possible. This is a significant undertaking, but if you can afford it, it is a clear and reliable way to reduce all four network delays.

In the context of networking, engineers sometimes use the term latency to describe the combination of these four delays. Because latency is such an overloaded term though, game developers more commonly discuss **round trip time**, or **RTT**. RTT refers to the time it takes for a packet to travel from one host to another, and then for a response packet to travel all the way back. This ends up reflecting not only the two-way processing, queuing, transmission, and propagation delays, but also the frame rate of the remote host, as this contributes to how quickly it can send the response packet. Note that traffic does not necessarily travel the same speed in each direction. The RTT is rarely exactly double the time it takes for a packet to go from one host to another. Regardless, games do often approximate one-way travel time by cutting the RTT in half.

Jitter

Once you have a good estimation of the RTT, you can take steps, as explained in Chapter 8, "Improved Latency Handling," to mitigate this delay and give clients the best experience possible for their given latency. However, when writing network code, you must be mindful that the RTT is not necessarily a constant value. For any two hosts, the RTT between them does typically hover around a certain value based on the average delays involved. However, any of these delays can change over time, leading to a deviation in RTT from the expected value. This deviation is known as **jitter**.

Any of the four network delays can contribute to jitter, although some are more likely to vary than others:

- **Processing delay.** As the least significant component of network latency, processing delay is also the least significant contributor to jitter. Processing delays may vary as routers dynamically adjust packet pathways, but this is a minor concern.
- **Transmission delay and propagation delay.** These two delays are both a function of the route a packet takes: Link layer protocols determine transmission delay and route length determines propagation delay. Thus these delays change when routers dynamically load balance traffic and alter routes to avoid heavily congested areas. This can fluctuate rapidly during times of heavy traffic and route changes can significantly alter round trip times.
- **Queuing delay.** Queuing delay is a function of the number of packets a router must process. Thus as the number of packets arriving at a router varies, the queuing delay will vary as well. Heavy traffic bursts can cause significant queuing delays and alter round trip times.

Jitter can negatively affect RTT mitigation algorithms, but even worse, it can cause packets to arrive completely out of order. Figure 7.2 illustrates how this can happen. Host A dispatches Packet 1, Packet 2, and Packet 3, in order, 5 ms apart, bound for a remote Host B. Packet 1 takes 45 ms to reach Host B, but due to a sudden influx of traffic on the route, Packet 2 takes 60 ms to reach Host B. Shortly after the traffic influx, the routers dynamically adjust the route causing Packet 3 to take only 30 ms to arrive at Host B. This results in Host B receiving Packet 3 first, then Packet 1, and then Packet 2.

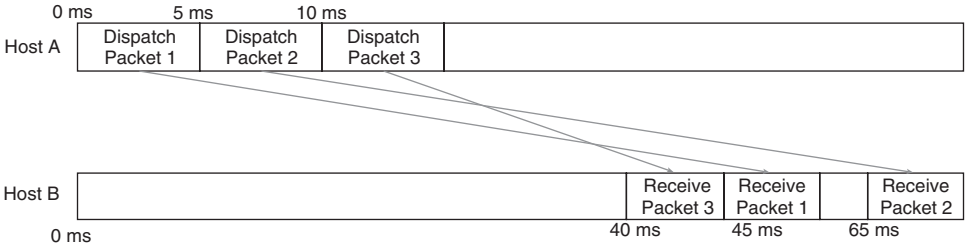

Figure 7.2 Jitter causing out of order packet delivery

To prevent errors due to packets arriving out of order, you must either use a reliable transport protocol, like TCP, that guarantees ordered packet delivery, or implement a custom system for ordering packets, as discussed in the latter half of this chapter.

Because of the problems jitter can cause, you should try to reduce it as much as possible to improve gameplay experience. Techniques that lower jitter are very similar to those that lower overall latency. Send as few packets as possible to keep traffic low. Locate servers near players to reduce the chance of encountering heavy traffic. Keep in mind that frame rate also affects RTT, so wild variations in frame rate will negatively impact clients. Make sure that complex operations are appropriately aggregated across multiple frames to prevent frame rate induced jitter.

Packet Loss

More significant than latency and jitter, the largest problem facing network game developers is packet loss. It's one thing if a packet takes a long time to get where it's going, but quite another if the packet never gets there at all.

Packets may drop for a variety of reasons:

- **Unreliable physical medium**. At its root, data transfer is the transfer of electromagnetic energy. Any external electromagnetic interference can cause corruption of this data. In the case of corrupted data, the link layer detects the corruption when validating checksums and discards the containing frames. Macroscale physical problems, such as a loose connection or even a nearby microwave oven, can also cause signal corruption and data loss.
- **Unreliable link layer**. Link layers have rules about when they can and cannot send data. Sometimes a link layer channel is completely full and an outgoing frame must be dropped. Because the link layer makes no guarantee of reliability, this is a perfectly acceptable response.
- **Unreliable network layer**. Recall that when packets arrive at a router faster than they can be processed, they are placed in a receiving queue. This queue has a fixed number of packets it can hold. When the queue is full, the router starts dropping either queued or incoming packets.

Dropped packets are a fact of life, and you must design your networking architecture to account for them. Regardless of packet loss mitigation, gameplay experience will be better with fewer dropped packets. When architecting at a high level, try to reduce the potential for packet loss. Use a data center with servers as close to your players as possible, because fewer routers and wires means a lower chance that one of them drops your data. Also, send as few packets as you can: Many routers base queue capacity on packet count, not total data. In these cases your game has a higher chance of flooding routers and overflowing queues if it sends many small packets than fewer large ones. Sending seven 200-byte packets through a clogged router requires there be seven free slots in the queue to avoid packet loss. However, sending the same 1400 bytes in a single packet only requires one free queue slot.

warning

Not all routers base queue slots on packet count—some allot queue space to individual sources based on incoming bandwidth, in which case smaller packets can actually be beneficial. If only one packet of the seven gets dropped due to bandwidth allocation instead of slot allocation, at least the other six get queued. It's worthwhile to know the routers in your data center and along heavily trafficked routes, especially because small packets waste bandwidth from headers, as mentioned in earlier chapters.

When its queues are full, a router does not necessarily drop each incoming packet. Instead, it may drop a previously queued packet. This happens when the router determines the incoming packet has higher priority or is more important than the queued one. Routers make priority decisions based on QoS data in the network layer header, and also sometimes on deeper information gleaned by inspecting the packet's payload. Some routers are even configured to make greedy decisions intended to reduce the overall traffic they must handle: They sometimes drop UDP packets before TCP packets because they know dropped TCP packets will just automatically be resent. Understanding the router configurations around your data centers and around ISPs throughout your target market can help you tune your packet types and traffic patterns to reduce packet loss. Ultimately, the simplest way to reduce dropped packets is to make sure your servers are on fast, stable Internet connections and are as close to your clients as possible.

Reliability: TCP or UDP?

Given the need for some level of networking reliability in almost every multiplayer game, an important decision to make early during development is that between TCP and UDP. Should your game rely on the existing reliability system built into TCP, or should you attempt to develop your own, customized reliability system on top of UDP? To answer this question, you need to consider the benefits and costs of each transport protocol.

The primary advantage of TCP is that it provides a time-tested, robust, and stable implementation of reliability. With no extra engineering effort, it guarantees all data will not only be delivered, but delivered in order. Additionally, it provides complex congestion control features which limit packet loss by sending data at a rate that does not overwhelm intermediate routers.

The major drawback of TCP is that everything it sends must be sent reliably and processed in order. In a multiplayer game with rapidly changing state, there are three different scenarios in which this mandatory, universally reliable transmission can cause problems:

1. **Loss of low-priority data interfering with the reception of high-priority data.** Consider a brief exchange between two players in a client-server first-person shooter. Player A on Client A and Player B on Client B are facing off against each other. Suddenly a rocket from some other source explodes in the distance, and the server sends a packet to Client A to play the distant explosion sound. Very shortly thereafter, Player B jumps in front of Player A and fires, and the server sends a packet containing this information to Client A. Due to fluctuating network traffic, the first packet gets dropped, but the second packet, containing Player B's movement, does not. The explosion sound is of low priority to Player A, whereas an enemy shooting him in the face is of high priority. Player A would probably not mind if the lost packet remained lost, and he never found out about the explosion. However, because TCP processes all packets in order, the TCP module will not pass the movement

packet to the game when it is received. Instead, it will wait until the server retransmits the lower-priority dropped packet before allowing the application to process the high-priority movement packet. This may, understandably, make Player A upset.

2. **Two separate streams of reliably ordered data interfering with each other.** Even in a game with no low-priority data, in which all data must be transmitted reliably, TCP's ordering system can still cause problems. Consider the prior scenario, but instead of an explosion, the first packet contains a chat message directed at Player A. Chat messages can be of critical importance so should be sent in some way that guarantees their receipt. In addition, chat messages need to be processed in order, because out of order chat messages can be quite confusing. However, chat messages only need to be processed in order relative to other chat messages. Player A would likely find it undesirable if the loss of a chat message packet prevented the processing of a headshot packet. In a game using TCP, this is exactly what happens.
3. **Retransmission of stale game state.** Imagine Player B runs all the way across the map to shoot Player A. She starts at position x = 0 and over the course of 5 seconds, runs to position x = 100. The server sends packets to Player A five times per second, each containing the latest x coordinate of Player B's position. If the server discovers that any or all of those packets get dropped, it will resend them. This means that while Player B is approaching her final position of x = 100, the server may be retransmitting old state data that had Player B closer to x = 0. This leads to Player A viewing a very stale position of Player B, and getting shot before receiving any information that Player B is nearby. This is not an acceptable experience for Player A.

In addition to enforcing mandatory reliability, there are a few other drawbacks to using TCP. Although congestion control helps prevent lost packets, it is not uniformly configurable on all platforms, and at times may result in your game sending packets at a slower rate than you'd like. The Nagle algorithm is a particularly bad offender here, as it can delay packets up to half a second before sending them out. In fact, games that use TCP as a transport protocol usually disable the Nagle algorithm to avoid this exact problem, though at the same time, giving up the benefit of the reduced packet count it provides.

Finally, TCP allocates a lot of resources to manage connections and track all data that may have to be resent. These allocations are usually managed by the OS and can be difficult to track or route through a custom memory manager if desired.

UDP, on the other hand, offers none of the built-in reliability and flow control that TCP provides. It does, however, present a blank canvas onto which you can paint any sort of custom reliability system your game requires. You can allow for the sending of reliable and unreliable data, or the interweaving of separately ordered streams of reliable data. You can also build a system that sends only the newest information when replacing dropped packets, instead of retransmitting the exact data that was lost. You can manage your memory yourself and have fine-grained control over how data is grouped into network layer packets.

All this comes at a cost of engineering and testing time. A custom spun implementation will naturally not be as mature and bug free as that of TCP. You can decrease some of this risk and cost by using a third-party UDP networking library, such as RakNet or Photon, though you may sacrifice some flexibility going that route. Additionally, UDP comes with an increased risk of packet loss, because routers may be configured to deprioritize UDP packets as described earlier. Table 7.1 sums up the differences between the protocols.

Table 7.1 Comparison of TCP to UDP

Column Heading	TCP	UDP
Reliability	Native. Everything is delivered and processed in the order it was sent.	None. Requires custom implementation but allows fine-grained reliability.
Flow control	Will automatically slow down rate of transmission if packets are getting dropped.	None. Requires custom flow and congestion control if desired.
Memory requirements	OS must keep copies of all data sent until it is acknowledged.	Custom implementation must decide what data to keep around and what to discard immediately. Memory managed at application level.
Router prioritization	May be prioritized over UDP packets.	May be dropped before TCP packets.

In most cases, the choice of which transport protocol to use comes down to this question: Does every piece of data the game sends need to be received, and does it need to be processed in a totally ordered fashion? If the answer is yes, you should consider using TCP. This is often true for turn-based games. Every piece of input must be received by every host and processed in the same order, so TCP is the perfect fit.

If TCP is not the absolute perfect fit for your game, and for most games it is not, you should use UDP with an application layer reliability system on top of it. This means either using a third-party middleware solution or building a custom system of your own. The rest of this chapter explores how you might go about building such a system.

Packet Delivery Notification

If UDP is the appropriate protocol for your game, you'll need to implement a reliability system. The first requirement for reliability is the ability to know whether or not a packet arrives at its destination or not. To do this, you'll need to build some kind of delivery notification module. The module's job is to help higher-level dependent modules send packets to remote hosts, and then to notify those dependent modules about whether the packets were received or not. By not implementing retransmission itself, it allows each dependent module to retransmit only the data it decides should be retransmitted. This is the main source of the

flexibility that UDP-based reliability provides that TCP does not. This section explores the `DeliveryNotificationManager`, which is one possible implementation of such a module, inspired by the *Starsiege: Tribes* connection manager.

The `DeliveryNotificationManager` needs to accomplish three things:

1. When transmitting, it must uniquely identify and tag each packet it sends out, so that it can associate delivery status with each packet and deliver this status to dependent modules in a meaningful way.
2. On the receiving end, it must examine incoming packets and send out an acknowledgment for each packet that it decides to process.
3. Back on the transmitting host, it must process incoming acknowledgments and notify dependent modules about which packets were received and which were dropped.

As an added bonus, this particular UDP reliability system will also ensure packets are never processed out of order. That is, if an old packet arrives at a destination after newer packets, the `DeliveryNotificationManager` will pretend the packet was dropped and ignore it. This is very useful, as it prevents stale data contained in old packets from accidentally overriding fresh data in newer packets. It is a slight overloading of the `DeliveryNotificationManager`'s purpose, but it is common and efficient to implement the functionality at this level.

Tagging Outgoing Packets

The `DeliveryNotificationManager` has to identify each packet it transmits, so that the receiving host has a way to specify which packet it acknowledges. Borrowing a technique from TCP, it does this by assigning each packet a sequence number. Unlike in TCP, however, the sequence number does not represent the number of bytes in a stream. It simply serves to provide a unique identifier for each transmitted packet.

To transmit a packet using the `DeliveryNotificationManager` the application creates an `OutputMemoryBitStream` to hold the packet, and then passes it to the `DeliveryNotificationManager::WriteSequenceNumber()` method, shown in Listing 7.1.

Listing 7.1 Tagging a Packet with a Sequence Number

```
InFlightPacket* DeliveryNotificationManager::WriteSequenceNumber(
    OutputMemoryBitStream& inPacket)
{
    PacketSequenceNumber sequenceNumber = mNextOutgoingSequenceNumber++;
    inPacket.Write(sequenceNumber);

    ++mDispatchedPacketCount;

    mInFlightPackets.emplace_back(sequenceNumber);
    return &mInFlightPackets.back();
}
```

The `WriteSequenceNumber` method writes the `DeliveryNotificationManager`'s next outgoing sequence number into the packet, and then increments the number in preparation for the next packet. In this way, no two packets sent in close succession should have the same sequence number, and each has a unique identifier.

The method then constructs an `InFlightPacket` and adds it to the `mInFlightPackets` container, which keeps track of all packets that have not yet been acknowledged. These `InFlightPacket` objects will be needed later when processing acknowledgments and reporting delivery status. After giving the `DeliveryNotificationManager` a chance to tag the packet with a sequence number, it is up to the application to write the payload of the packet and send it off to the destination host.

> **note**
>
> `PacketSequenceNumber` is a `typedef` so you can easily change the number of bits in a sequence number. In this case, it is a `uint16_t`, but depending on the number of packets you plan on sending, you might want to use more or fewer bits. The goal is to use as few bits as possible while minimizing the chance of wrapping the sequence number and then encountering a very old packet with a seemingly relevant sequence number from long before the wrap around. If you're pushing the bit count as low as possible, it can be very useful to include an unwrapped 32-bit sequence count during development for debugging and verification purposes. You'd then remove the extra sequence count when making release builds.

Receiving Packets and Sending Acknowledgments

When the destination host receives a packet, it sends an `InputMemoryBitStream` containing the packet's data to its own `DeliveryNotificationManager`'s `ProcessSequenceNumber()` method, shown in Listing 7.2.

Listing 7.2 Processing an Incoming Sequence Number

```
bool DeliveryNotificationManager::ProcessSequenceNumber(
    InputMemoryBitStream& inPacket)
{
    PacketSequenceNumber    sequenceNumber;

    inPacket.Read(sequenceNumber);
    if(sequenceNumber == mNextExpectedSequenceNumber)
    {
        //is this expected? add ack to the pending list and process packet
        mNextExpectedSequenceNumber = sequenceNumber + 1;
        AddPendingAck(sequenceNumber);
```

```
            return true;
        }
        //is sequence number < current expected? Then silently drop old packet.
        else if(sequenceNumber < mNextExpectedSequenceNumber)
        {
            return false;
        }
        //otherwise, we missed some packets
        else if(sequenceNumber > mNextExpectedSequenceNumber)
        {
            //consider all skipped packets as dropped, so
            //our next expected packet comes after this one...
            mNextExpectedSequenceNumber = sequenceNumber + 1;
            //add an ack for the packet and process it
            //when the sender detects break it acks, it can resend
            AddPendingAck(sequenceNumber);
            return true;
        }
    }
```

`ProcessSequenceNumber()` returns a `bool` indicating whether the packet should be processed by the application, or just completely ignored. This is how the `DeliveryNotificationManager` prevents out of order processing. The `mNextExpectedSequenceNumber` member variable keeps track of the next sequence number the destination host should receive in a packet. Because each transmitted packet has a consecutively increasing sequence number, the receiving host can easily predict which sequence number should be present in an incoming packet. Given that, there are three cases that might occur when the method reads a sequence number:

- **The incoming sequence number matches the expected sequence number.** In this case, the application should acknowledge the packet, and should process it. The `DeliveryNotificationManager` should increment its `mNextExpectedSequenceNumber` by 1.
- **The incoming sequence number is less than the expected sequence number.** This probably means the packet is older than packets that have already arrived. To avoid out of order processing, the host should not process the packet. It should also not acknowledge the packet, because the host should only acknowledge packets that it processes. There is an edge case that you must consider here. If the current `mNextExpectedSequenceNumber` is close to the maximum number representable by a `PacketSequenceNumber` and the incoming sequence number is close to the minimum, the sequence numbers may have wrapped around. Based on the rate at which your game sends packets, and the number of bits used in `PacketSequenceNumber`, this may or may not be possible. If it is possible, and the `mNextExpectedSequenceNumber` and incoming sequence number suggest it is likely, then you should handle this the same way as you would the following case.

- **The incoming sequence number is greater than the expected sequence number.** This is what happens when one or more packets get dropped or delayed. A different packet eventually gets through to the destination, but its sequence number is higher than expected. In this case, the application should still process the packet and should still acknowledge it. Unlike in TCP, the `DeliveryNotificationManager` does not promise to process every single packet sent in order. It only promises to process nothing out of order, and to report when packets are dropped. Thus it is perfectly safe to acknowledge and process packets that come in after previously transmitted packets were dropped. In addition, to prevent the processing of any old packets, should they arrive, the `DeliveryNotificationManager` should set its `mNextExpectedSequenceNumber` to the most recently received packet's sequence number plus one.

> **note**
>
> The first and third cases actually perform the exact same operation. They are called out separately in the code because they indicate different situations, but they could be collapsed into a single case by checking if `sequenceNumber ≥ mNextExpectedSequenceNumber`.

The `ProcessSequenceNumber()` method does not send any acknowledgments directly. Instead, it calls `AddPendingAck()` to track that an acknowledgment should be sent. It does this for efficiency. If a host receives many packets from another host, it would be inefficient to send an acknowledgment for each incoming packet. Even TCP is allowed to acknowledge only every other packet. In the case of a multiplayer game, the server may need to send several MTU-sized packets to a client before the client has to send any data back to the server. In cases like this, it is best to just accumulate all necessary acknowledgments and piggy back them on to the next packet the client sends to the server.

The `DeliveryNotificationManager` may accumulate several nonconsecutive acknowledgments. To efficiently track and serialize them, it keeps a `vector` of `AckRanges` in its `mPendingAcks` variable. It adds to them using the `AddPendingAck()` code shown in Listing 7.3.

Listing 7.3 Adding a Pending Acknowledgment

```
void DeliveryNotificationManager::AddPendingAck(
    PacketSequenceNumber inSequenceNumber)
{
    if(mPendingAcks.size() == 0 ||
       !mPendingAcks.back().ExtendIfShould(inSequenceNumber))
    {
        mPendingAcks.emplace_back(inSequenceNumber);
    }
}
```

An `AckRange` itself represents a collection of consecutive sequence numbers to acknowledge. It stores the first sequence number to acknowledge in its `mStart` member, and the count of how many sequence numbers to acknowledge in its `mCount` member. Thus, multiple `AckRange`s are only necessary when there is a break in the sequence. The code for `AckRange` is shown in Listing 7.4

Listing 7.4 Implementing `AckRange`

```
inline bool AckRange::ExtendIfShould
    (PacketSequenceNumber inSequenceNumber)
{
    if(inSequenceNumber == mStart + mCount)
    {
        ++mCount;
        return true;
    }
    else
    {
        return false;
    }
}

void AckRange::Write(OutputMemoryBitStream& inPacket) const
{
    inPacket.Write(mStart);
    bool hasCount = mCount > 1;
    inPacket.Write(hasCount);
    if(hasCount)
    {
        //let's assume you want to ack max of 8 bits...
        uint32_t countMinusOne = mCount - 1;
        uint8_t countToAck = countMinusOne > 255 ?
            255: static_cast<uint8_t>(countMinusOne);
        inPacket.Write(countToAck);
    }
}

void AckRange::Read(InputMemoryBitStream& inPacket)
{
    inPacket.Read(mStart);
    bool hasCount;
    inPacket.Read(hasCount);
    if(hasCount)
    {
        uint8_t countMinusOne;
        inPacket.Read(countMinusOne);
        mCount = countMinusOne + 1;
    }
```

```
    else
    {
        //default!
        mCount = 1;
    }
}
```

The `ExtendIfShould()` method checks if the sequence number is consecutive. If so, it increases the count and tells the caller the range was extended. If not, it returns false so the caller knows to construct a new `AckRange` for the nonconsecutive sequence number.

`Write()` and `Read()` work by first serializing the starting sequence number and then serializing the count. Instead of serializing the count directly, these methods take into account the fact that many games will typically only need to acknowledge a single packet at a time. Thus the methods use entropy encoding to efficiently serialize the count, with an expected value of 1. They also serialize the count as an 8-bit integer, assuming that more than 256 acknowledgments should never be needed. In truth, even 8 bits are high for the count, so this could easily be fewer.

When the receiving host is ready to send a reply packet, it writes any accumulated acknowledgments into the outgoing packet by calling `WritePendingAcks()` right after it writes its own sequence number. Listing 7.5 shows `WritePendingAcks()`.

Listing 7.5 Writing Pending Acknowledgments

```
void DeliveryNotificationManager::WritePendingAcks(
    OutputMemoryBitStream& inPacket)
{
    bool hasAcks = (mPendingAcks.size() > 0);
    inPacket.Write(hasAcks);
    if(hasAcks)
    {
        mPendingAcks.front().Write(inPacket);
        mPendingAcks.pop_front();
    }
}
```

Because not every packet necessarily contains acknowledgments, the method first writes a single bit to signal their presence. It then writes a single `AckRange` into the packet. It does this because packet loss is the exception, not the rule, and there will usually be only one `AckRange` pending. You could write all of the pending ranges, but this would require an extra indicator of how many `AckRanges` are present and could potentially bloat a packet. In the end, you want some flexibility, but not so much that it places an undue burden on your reply packet capacity. Studying the traffic pattern of your game will help you craft a system that is flexible enough for your edge cases but sufficiently efficient on average: For instance, if you feel confident that

your game will never need to acknowledge more than one packet at a time, you can remove the multi-acknowledgment system entirely and save a few bits per packet.

Receiving Acknowledgments and Delivering Status

Once a host sends out a data packet, it must listen for and process any acknowledgments accordingly. When the expected acknowledgments arrive, the `DeliveryNotificationManager` deduces that the corresponding packets arrived correctly and notifies the appropriate dependent modules of delivery success. When the expected acknowledgments do not arrive, the `DeliveryNotificationManager` deduces that packets were lost, and notifies the appropriate dependent modules of failure.

> **warning**
>
> Beware that the lack of an acknowledgment does not truly indicate the loss of a data packet. The data could have arrived successfully, but the packet containing the acknowledgment itself might have been lost. There is no way for the originally transmitting host to distinguish between these cases. In TCP, this is not a problem, because a retransmitted packet uses the exact same sequence number it used when originally sent. If a TCP module receives a duplicate packet, it knows to just ignore it.
>
> This is not the case for the `DeliveryNotificationManager`. Because lost data is not necessarily resent, every packet is unique and sequence numbers are never reused. This means a client module may decide to resend some reliable data based on the absence of an acknowledgment, and the receiving host may already have the data. In this case, it is up to the dependent module to uniquely identify the data itself to prevent duplication. For instance, if an `ExplosionManager` relies on the `DeliveryNotificationManager` to reliably send explosions across the Internet, it should uniquely number the explosions to ensure no explosion accidentally explodes twice on the receiving end.

To process acknowledgments and dispatch status notification, the host application uses the `ProcessAcks()` method, as shown in the Listing 7.6.

Listing 7.6 Processing the Acknowledgments

```
void DeliveryNotificationManager::ProcessAcks(
    InputMemoryBitStream& inPacket)
{
    bool hasAcks;
    inPacket.Read(hasAcks);
```

```
    if(hasAcks)
    {
        AckRange ackRange;
        ackRange.Read(inPacket);
        //for each InFlightPacket with seq# < start, handle failure...
        PacketSequenceNumber nextAckdSequenceNumber =
          ackRange.GetStart();
        uint32_t onePastAckdSequenceNumber =
            nextAckdSequenceNumber + ackRange.GetCount();
        while(nextAckdSequenceNumber < onePastAckdSequenceNumber &&
              !mInFlightPackets.empty())
        {
            const auto& nextInFlightPacket = mInFlightPackets.front();
            //if the packet seq# < ack seq#, we didn't get an ack for it,
            //so it probably wasn't delivered
            PacketSequenceNumber nextInFlightPacketSequenceNumber =
                 nextInFlightPacket.GetSequenceNumber();
            if(nextInFlightPacketSequenceNumber < nextAckdSequenceNumber)
            {
                //copy this so we can remove it before handling-
                //dependent modules shouldn't find it if seeing what's live
                auto copyOfInFlightPacket = nextInFlightPacket;
                mInFlightPackets.pop_front();
                HandlePacketDeliveryFailure(copyOfInFlightPacket);
            }
            else if(nextInFlightPacketSequenceNumber==
              nextAckdSequenceNumber)
            {
                HandlePacketDeliverySuccess(nextInFlightPacket);
                //received!
                mInFlightPackets.pop_front();
                ++nextAckdSequenceNumber;
            }
            else if(nextInFlightPacketSequenceNumber>
              nextAckdSequenceNumber)
            {
                //somehow part of this range was already removed
                //(maybe from timeout) check rest of range
                nextAckdSequenceNumber = nextInFlightPacketSequenceNumber;
            }
        }
    }
}
```

To process an `AckRange`, the `DeliveryNotificationManager` must determine which of its `InFlightPacket`s lie within the range. Because acknowledgments should be received in order, the method assumes that any packets with sequence numbers lower than the given range were dropped, and it reports their delivery as failed. It then reports any packets

within the range as successfully delivered. There can be quite a few packets in flight at any one time, but luckily it is not necessary to examine every single `InFlightPacket`. Because new `InFlightPacket`s are appended to the `mInFlightPackets` deque, all the `InFlightPacket`s are already ordered by sequence number. This means that when an `AckRange` comes in, the method can go through the `mInFlightPackets` in order, comparing each sequence number to the `AckRange`. Until it finds its first packet in the range, it reports all packets as dropped. Then, once it finds the first packet in the range, it reports its delivery as successful. Finally it needs only report success for the rest of the packets in the `AckRange` and it can exit without examining any other `InFlightPacket`s.

The final else-if clause handles the edge case in which the first known `InFlightPacket` is somewhere inside the `AckRange`, but not at the front. This can happen if a packet recently acknowledged was previously reported as dropped. In this case, `ProcessAcks()` just jumps to the packet's sequence number and reports all the remaining packets in range as successfully delivered.

You may wonder how a packet previously reported as dropped might later be acknowledged. This can happen if the acknowledgment took too long to arrive. Just as TCP resends packets when an acknowledgment is not prompt, the `DeliveryNotificationManager` should also be on the lookout for acknowledgments that have timed out. This is particularly useful when traffic is sparse, and there may not be a nonconsecutive acknowledgment to indicate a single dropped packet. To check for timed out packets, the host application should call the `ProcessTimedOutPackets()` method each frame, given in Listing 7.7.

Listing 7.7 Timing Out Packets

```
void DeliveryNotificationManager::ProcessTimedOutPackets()
{
    uint64_t timeoutTime = Timing::sInstance.GetTimeMS() - kAckTimeout;
    while( !mInFlightPackets.empty())
    {
        //packets are sorted, so all timed out packets must be at front
        const auto& nextInFlightPacket = mInFlightPackets.front();

        if(nextInFlightPacket.GetTimeDispatched() < timeoutTime)
        {
            HandlePacketDeliveryFailure(nextInFlightPacket);
            mInFlightPackets.pop_front();
        }
        else
        {
            //no packets beyond could be timed out
            break;
        }
    }
}
```

The `GetTimeDispatched()` method returns a timestamp set at creation time in the `InFlightPacket`'s constructor. Because the `InFlightPacket`s are sorted, the method only has to check the front of the list until it has found a packet that has not timed out. After that point, it is guaranteed all successive packets in flight have not timed out.

To track and report delivered and dropped packets, the aforementioned methods call `HandlePacketDeliveryFailure()` and `HandlePacketDeliverySuccess()` as shown in Listing 7.8.

Listing 7.8 Tracking Status

```
void DeliveryNotificationManager::HandlePacketDeliveryFailure(
    const InFlightPacket& inFlightPacket)
{
    ++mDroppedPacketCount;
    inFlightPacket.HandleDeliveryFailure(this);

}

void DeliveryNotificationManager::HandlePacketDeliverySuccess(
    const InFlightPacket& inFlightPacket)
{
    ++mDeliveredPacketCount;
    inFlightPacket.HandleDeliverySuccess(this);
}
```

These methods increment `mDroppedPacketCount` and `mDeliveredPacketCount`, accordingly. By doing so, the `DeliveryNotificationManager` can track packet delivery rates, and estimate packet loss rates for the future. If loss is too high, it can notify appropriate modules to decrease transmission rate, or the modules can notify the user directly that something might be wrong with the host's network connection. The `DeliveryNotificationManager` can also sum these values with the `mInFlightPackets` vector's size and assert they equal the `mDispatchedPacketCount`, incremented in `WriteSequenceNumber()`.

The aforementioned methods make use of `InFlightPacket`'s `HandleDeliveryFailure()` and `HandleDeliverySuccess()` methods to notify higher-level consumer modules about delivery status. To understand how they work, it's worth looking at the `InFlightPacket` class in Listing 7.9.

Listing 7.9 The `InFlightPacket` Class

```
class InFlightPacket
{
public:
```

```
    ....
    void SetTransmissionData(int inKey,
                             TransmissionDataPtr inTransmissionData)
    {
        mTransmissionDataMap[ inKey ] = inTransmissionData;
    }
    const TransmissionDataPtr GetTransmissionData(int inKey) const
    {
        auto it = mTransmissionDataMap.find(inKey);
        return (it != mTransmissionDataMap.end()) ? it->second: nullptr;
    }

    void HandleDeliveryFailure(
        DeliveryNotificationManager* inDeliveryNotificationManager) const
    {
        for(const auto& pair: mTransmissionDataMap)
        {
            pair.second->HandleDeliveryFailure
              (inDeliveryNotificationManager);
        }
    }
    void HandleDeliverySuccess(
        DeliveryNotificationManager* inDeliveryNotificationManager) const
    {
        for(const auto& pair: mTransmissionDataMap)
        {
            pair.second->HandleDeliverySuccess
              (inDeliveryNotificationManager);
        }
    }
private:
    PacketSequenceNumber mSequenceNumber;
    float mTimeDispatched;
    unordered_map<int, TransmissionDataPtr>    mTransmissionDataMap;
};
```

tip

Keeping the transmission data map in an `unordered_map` is clear and useful for demonstrative purposes. Iterating through an `unordered_map` is not very efficient and can lead to many cache misses. In production, if you have a small number of transmission data types, it can be better to just make a dedicated member variable for each type, or to store them in a fixed array with a dedicated index per type. If you need more than a few transmission data types, it might be worth it to keep them in a sorted vector.

Each `InFlightPacket` holds a container of `TransmissionData` pointers. `TransmissionData` is an abstract class with its own `HandleDeliverySucess()` and `HandleDeliveryFailure()` methods. Each dependent module that sends data via the `DeliveryNotificationManager` can create its own subclass of `TransmissionData`. Then, when a module writes reliable data into a packet's memory stream, it creates an instance of its customized `TransmissionData` subclass and uses `SetTransmissionData()` to add it to the `InFlightPacket`. When the `DeliveryNotificationManager` notifies the dependent module about a packet's success or failure, the module has a record of exactly what it stored in the given packet, allowing it to figure out how best to proceed. If the module needs to resend some of the data, it can. If it needs to send a newer version of the data, it can. If it needs to update custom variables elsewhere in the application, it can. In this way, the `DeliveryNotificationManager` provides a solid foundation on which to build a UDP-based reliability system.

> **note**
>
> Each pair of communicating hosts requires its own pair of `DeliveryNotificationManager`s. So in a client-server topology, if the server is talking to 10 clients, it needs 10 `DeliveryNotificationManager`s, one for each client. Then each client host uses its own `DeliveryNotificationManager` to communicate with the server.

Object Replication Reliability

You can use a `DeliveryNotificationManager` to send data reliably by resending any data that fails to reach its intended host. Simply extend `TransmissionData` with a `ReliableTransmissionData` class that contains all data sent in the packet. Then, inside the `HandleDeliveryFailed()` method, create a new packet and resend all the data. This is very close to how TCP implements reliability, however, and doesn't take full advantage of the `DeliveryNotificationManager`'s potential. To improve upon the TCP version of reliability, you do not have to resend the exact data that dropped. Instead, you can send only the latest version of the data that was dropped. This section will explore how to extend the `ReplicationManager` from Chapter 5 to support reliably resending the most recent data, inspired by the *Starsiege: Tribes* ghost manager.

The `ReplicationManager` of Chapter 5 has a very simple interface. Dependent modules create an output stream, prepare a packet, and then call `ReplicateCreate()`, `ReplicateUpdate()`, or `ReplicateDestroy()` to create, update, or destroy a remote object, accordingly. The problem with this methodology is that the `ReplicationManager` neither controls what data goes in which packets, nor keeps a record of that data. This does not lend itself well to supporting reliability.

To send data reliably, the `ReplicationManager` needs to be able to resend data whenever it learns that a packet carrying reliable data has dropped. To support this, the host application

needs to poll the ReplicationManager regularly, offering it a primed packet and asking if it has data it would like to write into the packet. This way, whenever the ReplicationManager knows about lost reliable data, it can write whatever it needs to into the provided packet. The host can pick the frequency at which it offers packets to the ReplicationManager based on estimated bandwidth, packet loss rate, or any other heuristic.

It's worthwhile to extend this mechanism further and consider how things could work if the only time the ReplicationManager wrote data into packets were when the client periodically offered it an outgoing packet to fill. This would mean that instead of gameplay systems creating a packet whenever they have changed data to replicate, they can instead just notify the ReplicationManager about the data, and the ReplicationManager can take care of writing the data at the next opportunity. This nicely creates a further layer of abstraction between gameplay systems and network code. The gameplay code no longer needs to create packets or care about the network. Instead, it just notifies the ReplicationManager about important changes, and the ReplicationManager takes care of writing those changes into packets periodically.

This also happens to create the perfect pathway for up-to-date reliability. Consider the three basic commands: create, update, and destroy. When the gameplay system sends a replication command for a target object to the ReplicationManager, the ReplicationManager can use that command and object to write the appropriate state into a future packet. It can then store the replication command, target object pointer, and written state bits as transmission data in the corresponding InFlightPacket record. If the ReplicationManager learns that a packet dropped, it can find the matching InFlightPacket, look up the command and object that it used when writing the packet originally, and then write fresh data to a new packet using the same command, object, and state bits. This is a vast improvement over TCP, because the ReplicationManager does not use the original, potentially stale data to write the new packet. It instead uses only the current state of the target object, which could be a 1/2 second newer than the original packet by this point.

To support such a system, the ReplicationManager needs to offer an interface that allows gameplay systems to batch replication requests. For each game object, a gameplay system can batch creation, a set of property updates, or destruction. The ReplicationManager keeps track of the latest replication command for each object, so it can write the appropriate replication data into a packet whenever it is offered one. It stores these ReplicationCommands in mNetworkReplicationCommand, a member variable mapping from an object's network identifier to the latest command for that object. Listing 7.10 shows the interface for batching commands, as well as the inner workings of the ReplicationCommand itself.

Listing 7.10 Batching Replication Commands

```
void ReplicationManager::BatchCreate(
    int inNetworkId, uint32_t inInitialDirtyState)
{
    mNetworkIdToReplicationCommand[inNetworkId] =
```

```
        ReplicationCommand(inInitialDirtyState);
}

void ReplicationManager::BatchDestroy(int inNetworkId)
{
    mNetworkIdToReplicationCommand[inNetworkId].SetDestroy();
}

void ReplicationManager::BatchStateDirty(
    int inNetworkId, uint32_t inDirtyState)
{
    mNetworkIdToReplicationCommand[inNetworkId].
        AddDirtyState(inDirtyState);
}

ReplicationCommand::ReplicationCommand(uint32_t inInitialDirtyState):
    mAction(RA_Create), mDirtyState(inInitialDirtyState) {}

void ReplicationCommand::AddDirtyState(uint32_t inState)
{
    mDirtyState |= inState;
}

void ReplicationCommand::SetDestroy()
{
    mAction = RA_Destroy;
}
```

Batching a create command maps an object's network identifier to a `ReplicationCommand` containing a create action, and state bits specifying all the properties that should be replicated, as described in Chapter 5. Batching an update command binary ORs additional state bits as dirty so that the `ReplicationManager` knows to replicate the changed data. Game systems should batch update commands whenever they change data that needs to be replicated. Finally, batching a destroy command finds the `ReplicationCommand` for the object's network identifier and changes its action to destroy. Note that if destruction is batched for an object, it supersedes any previously batched instructions, since in the latest state methodology, it makes no sense to send state updates for an object that has already been destroyed. Once commands have been batched, the `ReplicationManager` fills the next packet it is offered using the `WriteBatchedCommands()` method shown in Listing 7.11.

Listing 7.11 Writing Batched Commands

```
void ReplicationManager::WriteBatchedCommands(
    OutputMemoryBitStream& inStream, InFlightPacket* inFlightPacket)
{
    ReplicationManagerTransmissionDataPtr repTransData = nullptr;
    //run through each replication command and rep if necessary
```

```
    for(auto& pair: mNetworkIdToReplicationCommand)
    {
        ReplicationCommand& replicationCommand = pair.second;
        if(replicationCommand.HasDirtyState())
        {
            int networkId = pair.first;
            GameObject* gameObj =
                mLinkingContext->GetGameObject(networkId);
            if(gameObj)
            {
                ReplicationAction action =
                    replicationCommand.GetAction();
                ReplicationHeader rh(action, networkId,
                                     gameObj->GetClassId());
                rh.Write(inStream);

                uint32_t dirtyState =
                    replicationCommand.GetDirtyState();
                if(action == RA_Create || action == RA_Update)
                {
                    gameObj->Write(inStream, dirtyState);
                }
                //create transmission data if we haven't yet
                if(!repTransData)
                {
                    repTransData =
                std::make_shared<ReplicationManagerTransmissionData>(
                       this);
                    inFlightPacket->SetTransmissionData
                       ('RPLM',repTransData);
                }
                //now store what we put in this packet and clear state
                repTransData->AddReplication(networkId, action,
                                             dirtyState);
                replicationCommand.ClearDirtyState(dirtyState);
            }
        }
    }
}
void ReplicationCommand::ClearDirtyState(uint32_t inStateToClear)
{
    mDirtyState &= ~inStateToClear;
    if(mAction == RA_Destroy)
    {
        mAction = RA_Update;
    }
}
bool ReplicationCommand::HasDirtyState() const
```

```
{
    return (mAction == RA_Destroy) || (mDirtyState != 0);
}
```

`WriteBatchedCommand()` starts by iterating over the replication command map. If it finds a network identifier with a batched command, defined as having either nonzero dirty state or a destroy action, it writes the `ReplicationHeader` and state, just as it did in Chapter 5. Then, if it has not yet created a `ReplicationTransmissionData` instance, it creates one and adds it to the `InFlightPacket`. Instead of doing this at the top of the method, it does this only once it has determined that it has state to replicate. It then appends the network identifier, replication action, and dirty state bits to the transmission data so that it has a complete record of what it wrote into the packet. Finally, it clears the dirty state in the replication command, so that it will not attempt to replicate the data again until it changes. By the end of the call, the packet contains all the replication data that higher-level game systems have batched, and the `InFlightPacket` contains a record of the information used during replication.

When the `ReplicationManager` learns of the packet's fate from the `DeliveryNotificationManager`, it responds with one of the two methods in Listing 7.12.

Listing 7.12 Responding to Packet Delivery Status Notification

```
void ReplicationManagerTransmissionData::HandleDeliveryFailure(
    DeliveryNotificationManager* inDeliveryNotificationManager) const
{
    for(const ReplicationTransmission& rt: mReplications)
    {
        int networkId = rt.GetNetworkId();
        GameObject* go;
        switch(rt.GetAction())
        {
            case RA_Create:
                {
                    //recreate if not destroyed
                    go = mReplicationManager->GetLinkingContext()
                       ->GetGameObject(networkId);
                    if( go )
                    {
                        mReplicationManager->BatchCreate(networkId,
                                                         rt.GetState());
                    }
                }
                break;
            case RA_Update:
                    go = mReplicationManager->GetLinkingContext()
                        ->GetGameObject(networkId);
                    if(go)
                    {
```

```
                        mReplicationManager->BatchStateDirty(networkId,
                                                             rt.GetState());
                    }
                break;
            case RA_Destroy:
                mReplicationManager->BatchDestroy(networkId);
                break;
        }
    }
}

void ReplicationManagerTransmissionData::HandleDeliverySuccess
  (DeliveryNotificationManager* inDeliveryNotificationManager) const
{
    for(const ReplicationTransmission& rt: mReplications)
    {
        int networkId = rt.GetNetworkId();
        switch(rt.GetAction())
        {
            case RA_Create:
                //once ackd, can send as update instead of create
                mReplicationManager->HandleCreateAckd(networkId);
                break;
            case RA_Destroy:
                mReplicationManager->RemoveFromReplication(networkId);
                break;
        }
    }
}
```

`HandleDeliveryFailure()` implements the real magic of up-to-date reliability. If a dropped packet contains a creation command, it rebatches the creation command. If it contains a state update command, it marks the corresponding state as dirty so that the new state values will be sent at the next opportunity. Finally, if it contains a destroy command, it rebatches the destroy command. In the event of successful delivery, `HandleDeliverySuccess()` takes care of some housekeeping tasks. If the packet contained a creation command, it changes the creation command to an update command so that the object will not be replicated as a creation the next time a game system marks its state as dirty. If the packet contained a destroy command, the method removes the corresponding network identifier from the `mNetworkIdToReplicationCommandMap` because there should be no more replication commands batched by the game.

Optimizing from In-Flight Packets

There is a significant optimization worth making to the `ReplicationManager`, again taking a lead from the *Starsiege: Tribes* ghost manager. Consider the case of a car driving through the game world for 1 second. If a server sends state reliably to a client 20 times a

second, each packet will contain a different position of the car as it travels. If the packet sent at 0.9-second drops, it might be 200 ms later before the server's `ReplicationManager` realizes and attempts to resend new data. By that point, the car would have stopped. Because the server was sending constant updates while the car was driving, there would already be new packets on their way to the client, containing updated positions of the car. It would be wasteful for the server to resend the car's current position when a packet containing that very data was already in flight to the client. If there were some way for the `ReplicationManager` to know about the in-flight data, it could avoid sending redundant state. Luckily, there is! When the `ReplicationManager` first learns of the dropped data, it can search through the `DeliveryNotificationManager`'s list of `InFlightPackets` and check the `ReplicationTransmissionData` stored in each one. If it sees state data in flight for the given object and property, then it knows it does not need to resend that data: It's already on the way! Listing 7.13 contains an updated `RA_Update` case for the `HandleDeliveryFailure()` method that does just this.

Listing 7.13 Avoiding Redundant Retransmission

```
void ReplicationManagerTransmissionData::HandleDeliveryFailure(
    DeliveryNotificationManager* inDeliveryNotificationManager) const
{
    ...
    case RA_Update:
        go = mReplicationManager->GetLinkingContext()
            ->GetGameObject(networkId);
        if(go)
        {
            //look in all in flight packets,
            //remove written state from dirty state
            uint32_t state = rt.GetState();
            for(const auto& inFlightPacket:
                inDeliveryNotificationManager->GetInFlightPackets())
            {
               ReplicationManagerTransmissionDataPtr rmtdp =
               std::static_pointer_cast
                   <ReplicationManagerTransmissionData>(
                   inFlightPacket.GetTransmissionData('RPLM'));
               if(rmtdp)
               {
                   for(const ReplicationTransmission& otherRT:
                       rmtdp->mReplications )
                   {
                       if(otherRT.GetNetworkId() == networkId)
                       {
                           state &= ~otherRT.GetState();
                       }
                   }
               }
```

```
            }
            //if there's still any dirty state, rebatch it
            if( state )
            {
                mReplicationManager->BatchStateDirty(networkId, state);
            }
        }
        break;
    ...
}
```

The update case first captures the state that was dirty in the original replication. Then, it iterates through each of the `InFlightPacket`s stored by the `DeliveryNotificationManager`. In each packet, it tries to find the `ReplicationManager`'s transmission data entry. If it finds one, it searches through the contained `ReplicationTransmission`s. For each replication, if the network identifier matches the identifier in the original dropped replication, it unsets any bits in the original state that are set in the found state. This way, the `ReplicationManager` avoids resending any state already in flight. If no bits are still set in the state by the time the method finishes checking all packets, it doesn't need to rebatch any state at all.

The aforementioned "optimization" can require quite a bit of processing each time a full packet drops. However, given the typically low frequency of dropped packets, and the fact that bandwidth is often more dear than processing power, it can still be beneficial. As always, consider the tradeoffs in the specific context of your game.

Simulating Real-World Conditions

Given the hazards that await your game in the real world, it is important to create a test environment that can properly simulate latency, jitter, and packet loss. You can engineer a testing module to sit between a socket and the rest of your game and simulate real-world conditions. To simulate loss, decide the probability of a packet dropping that you'd like to simulate. Then, each time a packet comes in, use a random number to decide whether to drop the packet, or pass it on to the application. To simulate latency and jitter, decide the average latency and jitter distribution for the test. When a packet arrives, calculate the timestamp at which it would have arrived in the real world by adding its latency and jitter to the time at which it actually arrived. Then, instead of sending the packet to your game to be processed right away, stamp it with the simulated arrival time and insert it into a sorted list of packets. Finally, each frame of your game, examine the sorted list and only process those packets whose simulated arrival times are lower than the current time. Listing 7.14 gives an example of how to do so.

Listing 7.14 Simulating Loss, Latency, and Jitter

```
void RLSimulator::ReadIncomingPacketsIntoQueue()
{
    char packetMem[1500];
```

```
    int packetSize = sizeof(packetMem);
    InputMemoryBitStream inputStream(packetMem, packetSize * 8);
    SocketAddress fromAddress;

    while(receivedPackedCount < kMaxPacketsPerFrameCount)
    {
        int cnt = mSocket->ReceiveFrom(packetMem, packetSize, fromAddress);
        if(cnt == 0)
        {
            break;
        }
        else if(cnt < 0)
        {
            //handle error
        }
        else
        {
            //now, should we process the packet?
            if(RoboMath::GetRandomFloat() >= mDropPacketChance)
            {
                //we made it, queue packet for later processing
                float simulatedReceivedTime =
                    Timing::sInstance.GetTimef() +
                    mSimulatedLatency +
                    (RoboMath::GetRandomFloat() - 0.5f) *
                    mDoubleSimulatedMaxJitter;
                //keep list sorted by simulated receive time
                auto it = mPacketList.end();
                while(it != mPacketList.begin())
                {
                    --it;
                    if(it->GetReceivedTime() < simulatedReceivedTime)
                    {
                        //time comes after this element, so inc and break
                        ++it;
                        break;
                    }
                }
                mPacketList.emplace(it, simulatedReceivedTime,
                                    inputStream, fromAddress);
            }
        }
    }
}

void RLSimulator::ProcessQueuedPackets()
{
    float currentTime = Timing::sInstance.GetTimef();
    //look at the front packet...
    while(!mPacketList.empty())
```

```
    {
        ReceivedPacket& packet = mPacketList.front();
        //is it time to process this packet?
        if(currentTime > packet.GetReceivedTime())
        {
            ProcessPacket(packet.GetInputStream(),
                packet.GetFromAddress());
            mPacketList.pop_front();
        }
        else
        {
            break;
        }
    }
}
```

> **tip**
>
> For an even more accurate simulation, consider incorporating the fact that packets are usually dropped or delayed in groups of sequential packets. When a random check indicates packets should be dropped, you can use another random number to determine how many packets in a row should be affected.

Summary

The real world is a scary place for multiplayer games. Players want immediate feedback from their inputs, and the forces of nature act to prevent that. Even without a network component, video games have to deal with many sources of latency, including input sampling latency, rendering latency, and display-based latency. With physical networking added to the mix, multiplayer games must also deal with latency from propagation delay, transmission delay, processing delay, and queuing delay. As a game developer, there are actions you can take to reduce these delays, but they may be very expensive and out of scope for your game.

Fluctuating network conditions lead to packets arriving late, out of order, or not at all. To build an enjoyable game experience, you need some level of reliable transmission to mitigate these issues. One way to guarantee reliable transmission is to use the TCP transport protocol. Although TCP is a well-tested, turn-key reliability solution, it has a few disadvantages. It works for games that need absolutely all their data transported reliably, but is not suitable for typical games that care more about up-to-date data than perfectly reliable data. For these games, UDP is the best choice because of the flexibility it offers.

When using UDP you have the ability and requirement to build your own custom reliability layer. The foundation of this is usually a notification system that alerts your game when packets arrive successfully and when they drop. By keeping a record of the data in each packet, the game can then decide how to act when notified about a packet's fate.

You can build a variety of reliability modules on top of a delivery notification system. A very common module provides for redelivery of up-to-date object state in the event of packet loss, similar to the *Starsiege: Tribes* ghost manager. It does this by tracking the state sent in each packet, and then resending the latest version of any appropriate state not already in flight when notified of a lost packet.

It is important to test your reliability system in a controlled environment before exposing it to the harsh conditions of the real world. Using random-number generators and a buffer of incoming packets, you can build a system that simulates packet loss, latency, and jitter. You can then see how both your reliability system and your entire game perform under various simulated network conditions.

Once you have dealt with the low-level problems of the real world, you can begin to think about addressing latency on a higher level. Chapter 8, "Improved Latency Handling," addresses the challenge of giving networked players as close to a lag-free experience as possible.

Review Questions

1. What are five processes which contribute to non-network latency?
2. What are the four delays which contribute to network latency?
3. Give one manner to reduce each network delay.
4. For what does RTT stand and what does it mean?
5. What is jitter? What are some causes of jitter?
6. Extend the `DeliveryNotificationManager::ProcessSequenceNumber()` to function properly in the case of sequence numbers wrapping back to 0.
7. Expand the `DeliveryNotificationManager` so that all packets received on the same frame are buffered and then sorted before the `DeliveryNotificationManager` decides which packets are stale and should be dropped.
8. Explain how a `ReplicationManager` can use the `DeliveryNotificationManager` to provide improved reliability over TCP, and send up-to-date data to recover from dropped packets.
9. Use the `DeliveryNotificationManager` and `ReplicationManager` to implement a two-player tag game. Simulate real-life conditions to see how tolerant your logic is of packet loss, latency, and jitter.

Additional Readings

Almes, G., S. Kalidindi, and M. Zekauskas. (1999, September). *A One-Way Delay Metric for IPPM*. Retrieved from https://tools.ietf.org/html/rfc2679. Accessed September 12, 2015.

Carmack, John (2012, April). *Tweet*. Retrieved from https://twitter.com/id_aa_carmack/status/193480622533120001. Accessed September 12, 2015.

Carmack, John (2012, May). *Transatlantic ping faster than sending a pixel to the screen?* Retrieved from http://superuser.com/questions/419070/transatlantic-ping-faster-than-sending-a-pixel-to-the-screen/419167#419167. Accessed September 12, 2015.

Frohnmayer, Mark and Tim Gift (1999). *The TRIBES Engine Networking Model*. Retrieved from http://gamedevs.org/uploads/tribes-networking-model.pdf. Accessed September 12, 2015.

Hauser, Charlie (2015, January). *NA Server Roadmap Update: Optimizing the Internet for League and You*. Retrieved from http://boards.na.leagueoflegends.com/en/c/help-support/AMupzBHw-na-server-roadmap-update-optimizing-the-internet-for-league-and-you. Accessed September 12, 2015.

Paxson, V., G. Almes, J. Mahdavi, and M. Mathis. (1998, May). *Framework for IP Performance Metrics*. Retrieved from https://tools.ietf.org/html/rfc2330. Accessed September 12, 2015.

Savage, Phil (2015, January). *Riot Plans to Optimise the Internet for League of Legends Players*. Retrieved from http://www.pcgamer.com/riot-plans-to-optimise-the-internet-for-league-of-legends-players/. Accessed September 12, 2015.

Steed, Anthony and Manuel Fradinho Oliveira. (2010). *Networked Graphics*. Morgan Kaufman.

CHAPTER 8

IMPROVED LATENCY HANDLING

As a multiplayer game programmer, latency is your enemy. Your job is to make your players feel like they're playing on a server across the street, when it may really be across the country. This chapter explores some of the ways to make that happen.

The Dumb Terminal Client

On the topic of client-server network topology, Tim Sweeney famously once wrote, "The server is the man!" He was referring to the fact that in *Unreal's* networking system, the server itself is the only host that necessarily has a true and correct game state. This is a traditional requirement of any cheat-resistant client-server setup: The server is the only host running a simulation that matters. That means there is always some delay between the time when a player takes an action and the time when the player can observe the true game state that results from that action. Figure 8.1 illustrates this by showing the round trip of a packet.

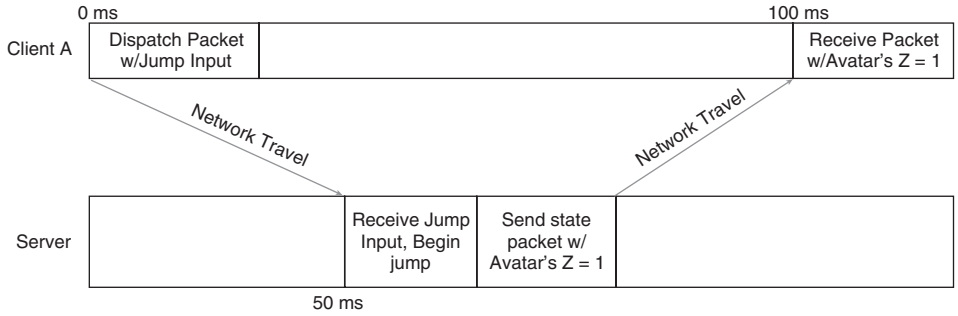

Figure 8.1 Packet round trip

In this example, the round trip time (RTT) between Client A and the server is 100 ms. At time 0, Player A's avatar on Client A is at rest, with a Z position of 0. Player A then pushes the jump button. Assuming roughly symmetric latency, it takes about 50 ms, or 1/2 RTT, for the packet carrying Player A's input to reach the server. When it receives the input, the server begins the player's jump, and sets her avatar's Z position to 1. It sends out new state, which reaches Client A another 50 ms, or 1/2 RTT, later. Client A updates Player A's avatar's Z position based on the state sent from the server and displays the results on screen. So finally, a full 100 ms after pushing the jump button, Player A gets to see the effect of the jump action.

From this demonstration, you can extract a useful conclusion: The true simulation running on the server is always 1/2 RTT ahead of the true simulation a remote player perceives. Put another way, if a player observes only the true simulation state replicated to the client, the player's perception of the state of the world is always at least 1/2 RTT older than the current true world state on the server. Depending on network traffic, physical distance and intermediate hardware, this can be as high as 100 ms or more.

Despite the noticeable lag between input and response, there were early multiplayer games that shipped with just this implementation. The original *Quake* was one game that endured despite its input latency. In *Quake*, and many of the other client-server games of the time, clients sent input to the server, and then the server ran the simulation and sent results back

to the client for display. Clients in these games were referred to as **dumb terminals** because they didn't need to understand anything about the simulation; their only purpose was to transmit input, receive the resulting state, and display it to the user. Because they showed only the state the server dictated, they never showed the user incorrect state. Although it might be delayed, whatever state a dumb terminal showed to a user was definitely a correct state at some recent point in time. Because the state throughout the system was always consistent and never incorrect, this method of networking can be classified as a **conservative algorithm**. At the expense of subjecting the user to noticeable latency, the conservative algorithm is at least never incorrect.

Besides just a feeling of latency, there is another problem with a pure dumb terminal. Figure 8.2 continues the example of Player A's jump.

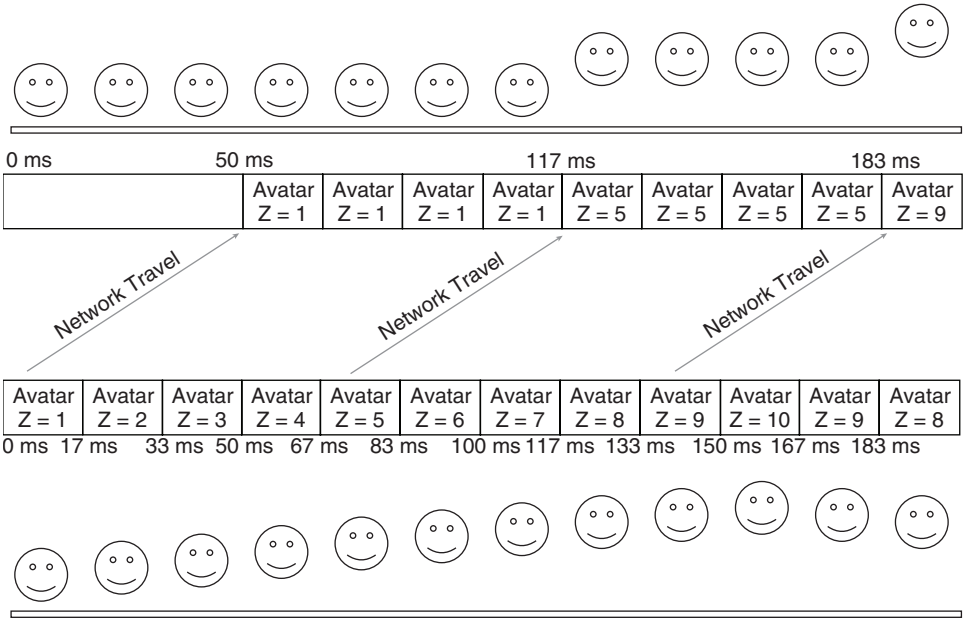

Figure 8.2 Jumping with 15 packets per second

Due to a high-powered GPU, Client A can run at 60 frames per second. The server can also run at 60 frames per second. However, due to bandwidth constraints on the connection between the server and Client A, the server can only send state updates 15 times per second. Assuming the player travels upward at 60 units per second at the start of her jump, the server smoothly increases her Z position by 1 unit each frame. However, it only sends state to the client every four frames. When Client A receives the state, it updates Player A's avatar's Z location, but then must render her at that Z location for four frames until new state from the server arrives. This means Player A sees the same picture on screen four frames in a row. Even though she spent good

money on a GPU that can render at 60 frames per second, she only gets the experience of playing at 15 frames per second due to network limitations. This would probably make her unhappy.

There is a third problem. Besides just causing a general feeling of unresponsiveness, this type of latency in a first-person shooter makes it difficult to aim at other players. Without an up-to-date representation of where players are, it can become an unpleasant challenge to figure out where to aim. It can be frustrating for players to think they are pulling off headshots, only to miss because their enemies were actually 100 ms ahead of where they were rendered. Too many experiences like that can cause players to switch to another game.

When building a client-server game, you cannot escape the issue of latency. However, you can reduce its impact on the player experience, and the following sections explore some common methods that multiplayer games use to handle latency.

Client Side Interpolation

The jumpiness brought on by infrequent state updates from the server can make players feel like their game is running slower than it actually is. One way to alleviate this is through **client side interpolation**. When using client side interpolation, the client game does not automatically teleport objects to their new positions sent by the server. Instead whenever the client receives new state for an object, it smoothly interpolates to that state over time using what's known as a **local perception filter**. Figure 8.3 illustrates the timing.

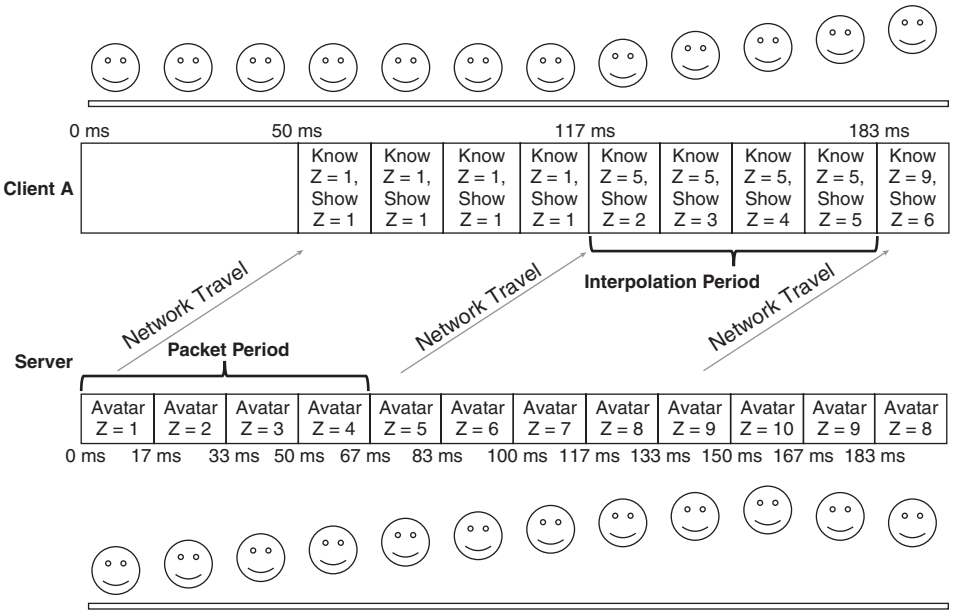

Figure 8.3 Timing of client side interpolation

Let IP represent the **interpolation period** in milliseconds, or how long the client takes to interpolate from old state to new state. Let PP represent the **packet period** in milliseconds, or how long the server waits between sending packets. The client finishes interpolating to a packet's state IP milliseconds after the packet arrives. Thus if IP is less than PP, the client will stop interpolating before a new packet has arrived, and the player may still experience a stutter. To make sure that the client state is changing smoothly each frame and the interpolation never stops, IP should be no less than PP. That way, whenever the client finishes interpolating to a given state, it will have already received the next state and can begin the process again.

Remember that a dumb terminal with no interpolation is always 1/2 RTT behind the server. If state arrives but the client does not display it right away, then the player's view of the world lags even further behind. Games using client side interpolation display state to players that is approximately 1/2 RTT + IP milliseconds behind the true state on the server. Thus, to minimize latency, IP should be as small as possible. This desire, combined with the fact IP must be greater than or equal to PP to prevent stutter, means it should be exactly equal to the PP.

The server can either notify the client how frequently it intends to send packets, or the client can compute the PP empirically by noting how rapidly packets arrive. Note that the server should set the packet period based on bandwidth, not latency. The server can send packets as frequently as it believes the network between the client and server can transmit them. This means that the latency perceived by players of games that use this type of client side interpolation is a factor of not only network latency, but also of network bandwidth.

Continuing the previous example, if the server sends 15 packets per second, the packet period is 66.7 ms. This means adding 66.7 ms of latency to 1/2 RTT that is already 50 ms. However, the game will look much smoother with interpolation than without, and it can make the experience more pleasant for the player such that latency is less of a concern.

Games that allow the player to manipulate the camera have a potential advantage here that can help reduce the feeling of extra latency. If the camera pose is not critical to the simulation, the game can handle it all client side. Walking around or shooting should require a trip to the server and back because they affect the simulation directly. Just aiming a camera might not affect the simulation in any manner, and if so, the client can update the renderer's view transform without waiting for a response from the server. Locally handling camera interaction gives the player instant feedback when she moves the camera. This combined with the smooth interpolation can help alleviate a lot of the unpleasant feelings associated with increased latency.

Client side interpolation still is considered a conservative algorithm: Although it may sometimes represent a state that the server did not replicate exactly, it only represents states that are between two states that the server did simulate. The client smoothens out the transition from state to state, but never guesses at what the server is doing, and therefore never ends up at a wildly incorrect state. This is not true about all methodologies, as the next section shows.

Client Side Prediction

Client side interpolation can smooth out your players' gameplay experiences, but it still won't bring them closer to what's actually happening on the server. Even with a tiny interpolation period, state is still at least 1/2 RTT old by the time the player sees it. To show game state that is any more current, your game needs to switch from interpolation to extrapolation. Through extrapolation, your client can take slightly old state received by the client and bring it approximately up to date before displaying it to the player. Techniques that perform this sort of extrapolation are often referred to as **client side prediction**.

To extrapolate the current state, the client must be able to run the same simulation code that the server runs. When the client receives a state update, it knows the update is 1/2 RTT ms old. To make the state more current, the client simply runs the simulation for an extra 1/2 RTT. Then, when the client displays the result to the player, it is a much closer approximation of the true game state currently simulating on the server. To maintain this approximation, the client continues running the simulation each frame and displaying the results to the player. Eventually, the client receives the next state packet from the server and internally simulates it for 1/2 RTT ms, at which point it ideally matches the exact state that the client has already calculated based on the previous received state!

To perform extrapolation by 1/2 RTT, the client must first be able to approximate the RTT. Because the clocks on the server and client are not necessarily in sync, the naïve approach of having the server timestamp a packet and then having the client check the age of the stamp will not work. Instead, the client must calculate the entire RTT and cut it in half. Figure 8.4 illustrates how to do so.

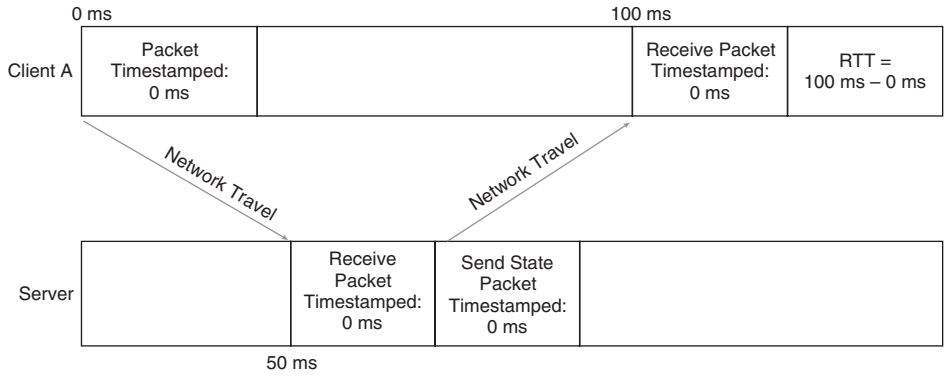

Figure 8.4 RTT calculation

The client sends a packet to the server containing a timestamp based on the client's own local clock. Upon receiving this packet, the server copies that timestamp into a new packet and sends it back to the client. When the client receives this new packet, it subtracts the old

timestamp, based on its clock, from the current time on its clock. This yields the exact amount of time between when the client first sent the packet and when it received the response—the definition of RTT. With this information, the client knows approximately how old the rest of the data in the packet is, and can use that information to extrapolate the contained state.

> **warning**
>
> Remember that 1/2 RTT is only an approximation of how old the data is. Traffic does not necessarily flow with the same speed in both directions, and thus the actual travel time from server to client may be more or less than 1/2 RTT. Regardless, 1/2 RTT is a good enough approximation for most real-time game purposes.

In *Robo Cat Action*, discussed in Chapter 6, the client already sends timestamped moves to the server, so the server just needs to send the timestamp from the most recent move back to the client when it sends state. Listing 8.1 shows the changes to the `NetworkManagerServer` which handle this.

Listing 8.1 Returning Client Timestamp to Client

```
void NetworkManagerServer::HandleInputPacket(
  ClientProxyPtr inClientProxy,
  InputMemoryBitStream& inInputStream)
{
  uint32_t moveCount = 0;
  Move move;
  inInputStream.Read(moveCount, 2);
  for(; moveCount > 0; --moveCount)
  {
    if(move.Read(inInputStream))
    {
      if(inClientProxy->GetUnprocessedMoveList().AddMoveIfNew(move))
      {
        inClientProxy->SetIsLastMoveTimestampDirty(true);
      }
    }
  }
}

bool MoveList::AddMoveIfNew(const Move& inMove)
{
  float timeStamp = inMove.GetTimestamp();
  if(timeStamp > mLastMoveTimestamp)
  {
    float deltaTime = mLastMoveTimestamp >= 0.f?
      timeStamp - mLastMoveTimestamp: 0.f;
    mLastMoveTimestamp = timeStamp;
```

```
      mMoves.emplace_back(inMove.GetInputState(), timeStamp, deltaTime);
      return true;
    }
    return false;
  }

  void NetworkManagerServer::WriteLastMoveTimestampIfDirty(
    OutputMemoryBitStream& inOutputStream,
    ClientProxyPtr inClientProxy)
  {
    bool isTimestampDirty = inClientProxy->IsLastMoveTimestampDirty();
    inOutputStream.Write(isTimestampDirty);
    if(isTimestampDirty)
    {
      inOutputStream.Write(
        inClientProxy->GetUnprocessedMoveList().GetLastMoveTimestamp());
      inClientProxy->SetIsLastMoveTimestampDirty(false);
    }
  }
```

For each incoming input packet, the server calls `HandleInputPacket`, which calls the move lists's `AddMoveIfNew` on each move in the packet. `AddMoveIfNew` checks each move's timestamp to see if it is newer than the most recently received move. If so, it adds the move to the move list and updates the list's most recent timestamp. If `AddMoveIfNew` added any moves, `HandleInputPacket` marks the most recent timestamp as dirty so that the `NetworkManager` will know the client should be sent this timestamp. When it is finally time for the `NetworkManager` to send a packet to the client, it checks to see if the timestamp for the client is dirty. If it is, it writes the cached timestamp from the move list into the packet. When the client receives this timestamp on the other end, it subtracts the timestamp from its current time, giving it an exact measure of how much time passed between when it sent its input to the server and when it received a corresponding response.

Dead Reckoning

Most aspects of a game simulation are deterministic, so the client can simulate them simply by executing a copy of the server's simulation code. Bullets fly through the air in the same way on both the server and the client. Balls bounce off walls and floors and obey the same laws of gravity. If the client has a copy of the AI code, it can even simulate AI-driven game objects to keep them in sync with the server. However, there is one class of objects that is completely nondeterministic and impossible to simulate perfectly: human players. There is no way the client can know what remote players are thinking, what actions they will initiate, or where they will move. This puts a kink in the extrapolation plan. In this scenario, the best solution is for the client to make an educated guess, and then correct this guess as necessary when an update arrives from the server.

In a networked game, **dead reckoning** is the process of predicting an entity's behavior based on the assumption that it will keep doing whatever it's currently doing. If this is a running player, it means assuming the player will keep running in the same direction. If it's a banking plane, it means assuming it will keep banking.

When the simulated object is controlled by a player, dead reckoning requires running the same simulation that the server is running, but in the absence of changing player input. This means that in addition to replicating the pose of player-controlled objects, the server must replicate any variables used by the simulation to calculate future poses. This includes velocity, acceleration, jump state, or more, depending on the specifics of your game.

As long as remote players continue doing exactly what they're doing, dead reckoning allows clients' games to accurately predict the current true world state on the server. However, when remote players take unexpected actions, the client simulation diverges from the true state, and must be corrected. Given that dead reckoning makes assumptions about behavior on the server before having all the facts, dead reckoning is not considered a conservative algorithm. It is instead known as an **optimistic algorithm**. It hopes for the best, guesses right most of the time, but sometimes is completely wrong and must adjust. Figure 8.5 illustrates this.

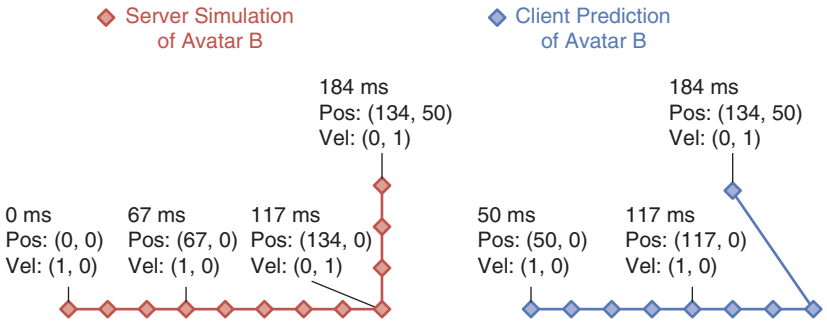

Figure 8.5 Dead reckoning misprediction

Assume an RTT of 100 ms and a frame rate of 60 frames per second. At time 50 ms, Client A receives information that Player B is at position (0, 0), running in the positive X direction at 1 unit per millisecond. Because this state is behind by 1/2 RTT, it simulates Player B's continued running at a constant speed for 50 ms before displaying Player B's position as (50, 0). Then, while waiting four frames for another state packet, it continues to simulate Player B's run each frame. By the fourth frame, at time 117 ms, it has predicted that Player B should be at (117, 0). It then receives a packet from the server replicating Player B's velocity as (1, 0) and pose as (67, 0). The client again simulates ahead for 1/2 RTT and finds that the position matches what it expected.

All is well. It continues the simulation for another four frames at which point it predicts Player B to be at (184, 0). However, at that point, it receives new state from the server dictating that Player B's position is (134, 0) but that his velocity has changed to (0, 1). Player B most likely stopped running forward and started strafing. Simulating ahead by 1/2 RTT yields a position of (134, 50), not at all what dead reckoning on the client previously predicted. Player B took an unexpected, unpredictable action, and as such, Client A's local simulation diverged from the true state of the world.

When a client detects that its local simulation has grown inaccurate, there are three ways it can remedy the situation:

- **Instant state update.** Simply update to the new state immediately. The player may notice the object jumping around, but that might be preferable to having inaccurate data. Remember that even after the immediate update, the state from the server is still 1/2 RTT old, so the client should use dead reckoning and the latest state to simulate it another 1/2 RTT.
- **Interpolation.** Taking a page from the client side interpolation method, your game can smoothly interpolate to the new state over a set number of frames. This could mean calculating and storing a delta to each incorrect state variable (position, rotation, etc.) that should be applied in each frame. Alternatively, you could just move the object part way to the corrected position and wait for future state from the server to continue the correction. One popular method is to use cubic spline interpolation to create a path that matches both position and velocity to transition smoothly from the predicted state to the corrected state. There is more in-depth information on this technique in the "Additional Readings" section.
- **Second-order state adjustment.** Even interpolation may be jarring if it suddenly ramps up the velocity of a near-stationary object. To be more subtle, your game can adjust second-order parameters like acceleration to very gently ease the simulation back in sync. This can be mathematically complex, but can provide the least noticeable corrections.

Typically, games will use a combination of these methods, based on the magnitude of the divergence and on the specifics of the game. A fast-paced shooter will usually interpolate for a small error and teleport for a large. A slower-paced game like a flight simulator or giant robot mech title might use second-order state adjustment for all but the largest errors.

Dead reckoning works well for remote players, because the local player doesn't actually know exactly what remote players are doing. When Player A watches Player B's avatar run across the screen, the simulation diverges every time Player B changes direction, but that's very hard for Player A to determine; without being in the same room as Player B, Player A doesn't actually know when Player B is changing input. For the most part, she sees the simulation as consistent, even though the client application is always guessing at least 1/2 RTT ahead of whatever the server has told it.

Client Move Prediction and Replay

Dead reckoning cannot hide latency for a local player. Consider the case of Player A, on Client A, starting to run forward. Dead reckoning uses state sent by the server to simulate, so from the time she pushes forward, it takes 1/2 RTT for the input to get to server, at which point the server adjusts her velocity. Then it takes 1/2 RTT for the velocity to get back to Client A, at which point the game can use dead reckoning. There's still a lag of RTT between when a player presses a button and when that player sees results.

There is a better alternative. Player A enters all her input directly into Client A, so the game on Client A can just use that input to simulate her avatar. As soon as Player A pushes a button to run forward, the client can start simulating her run. When the input packet reaches the server, it can begin the simulation as well, updating Player A's state accordingly. Not everything is so simple though.

A problem arises when the server sends a packet back to Client A containing Player A's replication state. Remember that when using client side prediction, all incoming state should be simulated an additional 1/2 RTT to catch up to the true state of the world. When simulating remote players, the client can just use dead reckoning and update assuming no change in input. Typically the updated incoming state will match the exact state the client has already predicted—if it doesn't, the client can smoothly interpolate the remote player into place. This won't work for local players. Local players know exactly where they are and will notice interpolation. They should not experience drifting or smoothing whenever they change their input. Ideally, moving around should feel to a local player like she is playing a single player, non-networked game.

One possible solution to this problem is to completely ignore the server's state for the local player. Client A can derive Player A's state solely from its local simulation, and Player A will have a smooth movement experience, with no latency. Unfortunately, this can cause Player A's state to diverge from the server's true state. If Player B bumps into Player A, there is no way for Client A to accurately predict the server's resolution of the collision. Only the server knows Player B's true position. Client A has a dead reckoned approximation of Player B's position, so cannot resolve the collision in exactly the same way the server would. Player A might end up in a pit of fire on the server, yet free and clear on the client, which can lead to much confusion. Because Client A ignores all incoming Player A state, there would be no way for the client and server to ever sync up again.

Luckily, there is a better solution. When Client A receives Player A's state from the server, Client A can use Player A's inputs to resimulate any state changes Player A instigated since the server calculated the incoming state. Instead of simulating the 1/2 RTT using dead reckoning, the client can simulate the 1/2 RTT using the exact input Player A used when the client side simulation originally ran. By introducing the concept of a **move**, input state tied to a timestamp, the client can keep track of what Player A was doing at all times. Whenever incoming state

arrives for a local player, the client can figure out which moves the server did not yet receive when calculating that state, and then apply those moves locally. Unless there was an encounter with an unexpected, remote player initiated event, this should end up with the same state the client had already locally predicted.

To extend *Robo Cat Action* with support for move replay, the first step is for the client to hold on to moves in the move list until the server has incorporated them into its simulation of state. Listing 8.2 shows the necessary changes to do so.

Listing 8.2 Retaining Moves

```
void NetworkManagerClient::SendInputPacket()
{
  const MoveList& moveList = InputManager::sInstance->GetMoveList();
  if(moveList.HasMoves())
  {

    OutputMemoryBitStream inputPacket;
    inputPacket.Write(kInputCC);
    mDeliveryNotificationManager.WriteState(inputPacket);
    //write the 3 latest moves for added reliability!
    int moveCount = moveList.GetMoveCount();
    int firstMoveIndex = moveCount - 3;
    if(firstMoveIndex < 3)
    {
      firstMoveIndex = 0;
    }
    auto move = moveList.begin() + firstMoveIndex;
    inputPacket.Write(moveCount - firstMoveIndex, 2);
    for(; firstMoveIndex < moveCount; ++firstMoveIndex, ++move)
    {
      move->Write(inputPacket);
    }
    SendPacket(inputPacket, mServerAddress);
  }
}
void
NetworkManagerClient::ReadLastMoveProcessedOnServerTimestamp(
  InputMemoryBitStream& inInputStream)
{
  bool isTimestampDirty;
  inInputStream.Read(isTimestampDirty);
  if(isTimestampDirty)
  {
    inPacketBuffer.Read(mLastMoveProcessedByServerTimestamp);
    mLastRoundTripTime = Timing::sInstance.GetFrameStartTime()
      - mLastMoveProcessedByServerTimestamp;
```

```
    InputManager::sInstance->GetMoveList().
      RemovedProcessedMoves(mLastMoveProcessedByServerTimestamp);
  }
}

void MoveList::RemovedProcessedMoves(
  float inLastMoveProcessedOnServerTimestamp)
{
  while(!mMoves.empty() &&
      mMoves.front().GetTimestamp() <=
       inLastMoveProcessedOnServerTimestamp)
  {
    mMoves.pop_front();
  }
}
```

Notice how `SendInputPacket` no longer clears the move list as soon as it sends the packet. Instead, it holds on to the moves so it can use them for move replay after receiving server state. As an added bonus, because moves now persist for more than a packet, the client sends the three most recent moves in the list. That way, if any input packets are dropped on the way to the server, the moves will have two more chances to make it through. This doesn't guarantee reliability but it significantly increases the chances.

When the client receives a state packet, it uses `ReadLastMoveProcessedOnServerTimestamp` to process any move timestamp the server might have returned. If it finds one, it subtracts the timestamp from the current time to measure RTT, which is useful for dead reckoning. It then calls `RemovedProcessedMoves` to remove any moves marked as at or before that timestamp. That means that after `ReadLastMoveProcessedOnServerTimestamp` completes, the client's local move list contains only moves which the server has not yet seen, and thus should be applied to any incoming state from the server. Listing 8.3 details the additions to the `RoboCat::Read()` method.

Listing 8.3 Replaying Moves

```
void RoboCatClient::Read(InputMemoryBitStream& inInputStream)
{
  float oldRotation = GetRotation();
  Vector3 oldLocation = GetLocation();
  Vector3 oldVelocity = GetVelocity();

  //...Read State Code Omitted...
  bool isLocalPlayer =
    (GetPlayerId() == NetworkManagerClient::sInstance->GetPlayerId());
  if(isLocalPlayer)
  {
```

```
    DoClientSidePredictionAfterReplicationForLocalCat(readState);
  }
  else
  {
    DoClientSidePredictionAfterReplicationForRemoteCat(readState);
  }
  //if this is not a create packet, smooth out any jumps
  if(!IsCreatePacket(readState))
  {
    InterpolateClientSidePrediction(
      oldRotation, oldLocation, oldVelocity, !isLocalPlayer);
  }
}

void RoboCatClient::DoClientSidePredictionAfterReplicationForLocalCat(
  uint32_t inReadState)
{
  //replay moves only if we received new pose
  if((inReadState & ECRS_Pose) != 0)
  {
    const MoveList& moveList = InputManager::sInstance->GetMoveList();

    for(const Move& move : moveList)
    {
      float deltaTime = move.GetDeltaTime();
      ProcessInput(deltaTime, move.GetInputState());

      SimulateMovement(deltaTime);
    }
  }
}

void RoboCatClient::DoClientSidePredictionAfterReplicationForRemoteCat(
  uint32_t inReadState)
{
  if((inReadState & ECRS_Pose) != 0)
  {
    //simulate movement for an additional RTT
    float rtt = NetworkManagerClient::sInstance->GetRoundTripTime();

    //split into framelength sized chunks so we don't run through walls
    //and do crazy things...
    float deltaTime = 1.f / 30.f;
    while(true)
    {
      if(rtt < deltaTime)
      {
        SimulateMovement(rtt);
        break;
      }
```

```
        else
        {
          SimulateMovement(deltaTime);
          rtt -= deltaTime;
        }
      }
    }
  }
```

The `Read` method begins by storing the current state of the object, so that the method can know later if any adjustments requiring smoothing occurred. It then updates state by reading it in from the packet as described in earlier chapters. After the update, it applies client side prediction to advance the replicated state by 1/2 RTT. If the replicated object is controlled by a local player, it calls `DoClientSidePredictionAfterReplicationForLocalCat` to run move replay. Otherwise, it calls `DoClientSidePredictionAfterReplicationForRemoteCat` to run dead reckoning.

`DoClientSidePredictionAfterReplicationForLocalCat` first checks to make sure that a pose was replicated. If not, there is no need to advance the simulation. If there was a pose, the method iterates through all remaining moves in the move list and applies them to the local `RoboCat`. This simulates all player actions that the server has not factored into its simulation yet. If nothing unexpected happened on the server, this function should leave the local cat's state exactly how it was before the `Read` method processed the packet in the first place.

If the cat being replicated is remote, the `DoClientSidePredictionAfterReplicationForRemoteCat` method advances the simulation using the latest known state for the cat. This consists of calling `SimulateMovement` for the appropriate amount of time without any associated `ProcessInput` calls. Again, if nothing unexpected happened on the server, this should also result in state that matches the state before the `Read` method began. However, unlike for local cats, it is very likely that something unexpected happened; remote players are always performing actions such as changing direction, speeding up or slowing down, and so on.

After performing client side prediction, the `Read()` method finally calls `InterpolateClientSidePrediction()` to handle any state that may have changed. By passing in old state, the interpolation method can decide how much, if at all, it should smooth out the change from old state to new state.

Hiding Latency through Tricks and Optimism

Delayed movement is not the only indication of latency to a player. When a player presses the button to shoot a gun, she expects her gun to fire immediately. When she tries to cast an attack spell, she expects her avatar to throw a big ball of fire. Move replay does not handle a situation

like this, so something else is necessary. It's usually too complicated for the client to create projectiles in a way that the server can take over replicating their state once it creates them itself—there is a simpler solution.

Almost all video game actions have **tells**, or visual cues that indicate something is happening. Muzzle flashes precede plasma blasts, and mages wave their hands and mumble before spraying fire. These tells usually last at least as long as a round trip to the server and back. This means that, optimistically, the client application can give a local player instant feedback to any input by playing the appropriate animation and effects locally, while waiting for the true simulation to be updated on the server. This doesn't mean that the client spawns projectiles, but it does start playing the spell casting animation and sound. If all is well, during the spell casting, the server receives the input packet, spawns the fire ball, and replicates it to the client, in time to show up as a result of the spell casting. Dead reckoning code advances the projectile forward by 1/2 RTT and it looks to the player as if she threw a fireball with no latency. If there is a problem, for instance, if the server knows that the player was recently silenced but hasn't yet replicated that to the player, the optimism proves unwarranted and the spell casting animation fires without a projectile appearing. This is a rare case though, and well worth the benefit typically provided.

Server Side Rewind

Using these various client side prediction techniques, your game can provide a fairly responsive experience to players, even in the presence of moderate latency. However, there is still one common type of game action which client side prediction does not handle perfectly: the long range, instant-hit weapon. When a player equips a sniper rifle, perfectly positions the reticle over another player, and pulls the trigger, she expects a perfect hit. However, due to the inaccuracies of dead reckoning, it is possible that a perfectly lined up shot on the client is not a perfectly lined up shot on the server. This can be a problem for games that rely on realistic, instant-hit weapons.

There is a solution to this, made popular by Valve's Source Engine, and responsible for the accuracy players feel when firing weapons in games like *Counter-Strike*. At its core, it works by rewinding state on the server to exactly the state the player perceived when lining up a shot and firing. That way, if the player perceived that she aimed perfectly, her shot will hit 100% of the time.

To accomplish this feat, the game must make a few adjustments to the client side prediction methods discussed earlier:

- **Use client side interpolation without dead reckoning for remote players.** The server needs to have accurate knowledge of exactly what client players see at any time. Because dead reckoning relies on the client advancing the simulation based on its assumptions, it

would cause extra complexity for the server, and thus should be turned off. To prevent any jerkiness or stuttering between packets, the client instead uses client side interpolation as described earlier in this chapter. The interpolation period should be exactly equal to the packet period, which is tightly controlled by the server. Client side interpolation introduces additional latency, but it turns out this is not significantly noticed by the player because of move replay and the server side rewind algorithm.

- **Use local client move prediction and move replay.** Although client side prediction is disabled for remote players, it must remain on for the local player. Without local move prediction and move replay, the local player would instantly notice both the latency from network traffic and the increased latency from the client side interpolation. However, by simulating player moves immediately, the local player never feels lagged, regardless of how much latency there is.
- **Record the client's view in each move packet sent to the server.** The client should stamp every input packet sent with the IDs of the frames between which the client is currently interpolating, and the percentage of the interpolation that is complete. This gives the server an exact indication of the client's perception of the world at the time.
- **On the server, store the poses of every relevant object for the last several frames.** When a client input packet comes in containing a shot, look up the two stored frames between which the client was interpolating at the time of the shot. Use the interpolation percentage in the packet to rewind all relevant objects to exactly where they were when the client pulled the trigger. Then perform a ray cast from the client's position to determine if the shot landed.

Server side rewind guarantees that if the client player lined up a shot correctly, it will land on the server. This gives a very satisfying feeling to the shooting player. However, it does not come without drawbacks. Because it rewinds server time by an amount based on the latency between server and client, it can end up causing some unexpected and frustrating experiences for the victims of the shots. Player A may think she has safely ducked around a corner, taking refuge from Player B. However, if Player B is on a particularly laggy network connection, he might have a view of the world that is 300 ms behind that of Player A. Thus on his computer, Player A may not have ducked behind the corner yet. If he lines up the shot and fires, the server will credit a hit to him and alert Player A that she was shot, even though she believed she was safely around a corner. As for all things in game development, it is a tradeoff. Only use these techniques if it is appropriate based on the specifics of your game.

Summary

Although stuttering and lag can ruin a multiplayer game experience, there are several strategies which help mitigate the problems. These days, it is practically required that a multiplayer game make use of one or more of these techniques.

Client side interpolation with a local perception filter smoothens out incoming state updates by interpolating to them instead of immediately presenting them to the client. An interpolation period equal to the period between state updates will provide the player with a consistently updating state, but will increase the player's perception of latency. It will never show the user an incorrect state.

Client side prediction uses extrapolation instead of interpolation to mask latency and keeps the client's game state in sync with the server's true game state. State updates are at least 1/2 RTT old by the time they reach the client, so the client can approximate the true game state by extrapolating the simulation for a duration of 1/2 RTT past the incoming state.

Through dead reckoning, a client uses the last known state of an object to extrapolate future state. It optimistically assumes remote players have not changed their input. Inputs change often, though, so the server does frequently send state to the client that differs from its approximation. When this happens, the client has many ways to factor this changed state into its own simulation, and update what it shows to the player.

Through move prediction and replay, a client can instantly simulate the results of local player input. When receiving local player state from the server, the client advances the state 1/2 RTT by replaying any move the player has made that the server has not yet processed. In most cases, this brings the replicated state into sync with the simulated client state. In the case of unexpected, server side events, like collisions with other players, the client can smooth the replicated, corrected state back into its local simulation.

For the ultimate in lag compensation when dealing with instant-hit weapons, games can employ server side rewind. The server buffers object positions for several frames and actually rewinds state to match the client's view when processing instant-hit weapon fire. This gives an increased feeling of precision to the shooter, but can result in targeted players taking damage even after they perceive they have safely taken cover.

Review Questions

1. What is meant by the term dumb client? What is the main benefit of a game which uses dumb clients?
2. What is the main advantage of client side interpolation? What is the main drawback?
3. On a dumb client, the state presented to the user is at least how much older than the true state running on the server?
4. What is the difference between a conservative algorithm and an optimistic algorithm? Give an example of each.
5. When is dead reckoning useful? How does it predict the positions of objects?
6. Give three ways to correct predicted state when it turns out to be incorrect.

7. Explain a system which allows a local player to experience no lag at all regarding their own movement.
8. What problem does server side rewind solve? What is its main advantage? What is its main disadvantage?
9. Expand *Robo Cat Action* with an optional instant-hit yarn ball and implement server side rewind hit detection.

Additional Readings

Aldridge, David. (2011, March). *Shot You First: Networking the Gameplay of HALO: REACH*. Retrieved from http://www.gdcvault.com/play/1014345/I-Shot-You-First-Networking. Accessed September 12, 2015.

Bernier, Yahn W. (2001) *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization*. Retrieved from https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization. Accessed September 12, 2015.

Caldwell, Nick. (2000, February) *Defeating Lag with Cubic Splines*. Retrieved from http://www.gamedev.net/page/resources/_/technical/multiplayer-and-network-programming/defeating-lag-with-cubic-splines-r914. Accessed September 12, 2015.

Carmack, J. (1996, August). *Here is the New Plan*. Retrieved from http://fabiensanglard.net/quakeSource/johnc-log.aug.htm. Accessed September 12, 2015.

Sweeney, Tim. *Unreal Networking Architecture*. Retrieved from https://udn.epicgames.com/Three/NetworkingOverview.html. Accessed September 12, 2015.

This page intentionally left blank

CHAPTER 9

SCALABILITY

Scaling up a networked game introduces a host of new challenges that don't exist for a game of a smaller scale. This chapter takes a look at some of the issues that crop up as the scale of a game increases, and some solutions to these issues.

Object Scope and Relevancy

Recall that the discussion of the *Tribes* model in Chapter 1 mentioned the concept of the **scope** or **relevancy** of an object. In this context, an object is considered *in scope* or *relevant* for a particular client when that client should be informed about updates to the object in question. For a smaller game, it may be viable to have all objects always be in scope or relevant to all clients in the game. This naturally means that all updates to objects on the server will be replicated to all clients. However, such an approach is not realistic for a larger game, both in terms of bandwidth and in terms of processing time for the client. In a game with 64 players, it may not be important to know about a player several kilometers away. In this case, sending information about this far away player would be a waste of resources. It therefore makes sense that if the server deems that Client A is too far away from object J, there is no need to send any updates to Client regarding the object. An additional benefit of reducing the replication data sent to each client is that it reduces the potential for cheating, a topic that is discussed in detail in Chapter 10, "Security."

However, object relevancy is rarely a binary proposition. For example, suppose object J is actually the avatar representing another player in the game. Suppose the game in question has a scoreboard that displays the health of every player in the game, regardless of the distance. In this scenario, the health of every player object is always relevant, even if other information regarding the player object is not. Thus it makes sense that the server will always send the health of other players, even if the rest of their object data may not be relevant. Furthermore, different objects could have different update frequencies based on their priority, which adds further complexity. In the interest of simplification, this section will consider relevancy of objects on a binary basis. But one should remain cognizant of the fact that relevancy in a commercial game rarely will be entirely binary in nature for every object in the game.

Returning to the example of the game with 64 players, the idea of deeming objects far away as out of scope is considered a **spatial** approach. Although simple distance checking is a very quick way to determine relevancy, typically it is not robust enough to be the sole mechanism of relevancy. To understand why this is the case, consider the example of a player in a first-person shooter. Suppose that the initial design of the game supports two different weapons: a pistol and an assault rifle. The network programmer thus decides to scope objects based on their distance—anything further than the assault rifle's range is deemed out of scope. In testing, the amount of bandwidth consumption is right at an acceptable limit. However, if the designers later decide to add a sniper rifle with a scope, with twice the range of the assault rifle, the number of relevant objects will increase greatly.

There are other issues related to only using distance to eliminating objects. A player in the middle of a level is more likely to be in range of objects than a player on the outskirts of the level. Furthermore, considering only distances assigns equal weight to objects in front of and behind a player, which is counterintuitive. Although a distance-based approach to object scope is simple, all objects around the player are deemed relevant, even those that may be behind a wall. These issues are shown in Figure 9.1.

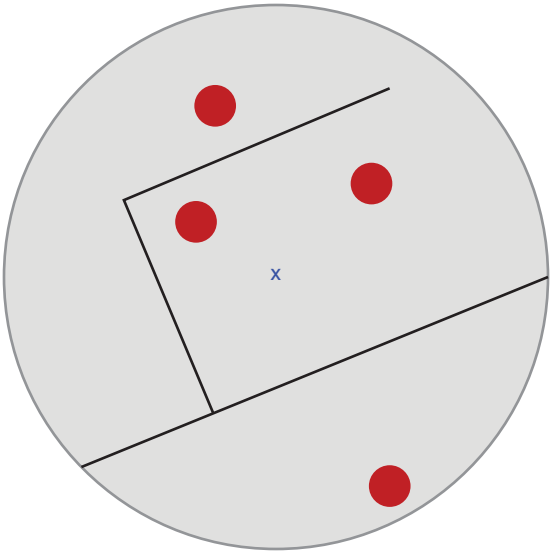

Figure 9.1 The player, designated by the X, in relation to relevant objects

The remainder of this section focuses on approaches more complex than simple distance checking. Many of these techniques are also commonly used in **visibility culling**, a category of rendering optimizations that try, as early as possible in the rendering process, to eliminate objects that are not visible. However, given the nature of latency in a networked game, some modifications are typically necessary to make a visibility culling approach suitable for object relevancy.

Static Zones

One approach to reducing the number of objects that are relevant is to break the world up into **static zones**. Only objects in the same static zone as the player are considered relevant. This kind of approach is often used in shared world games such as MMORPGs. For example, a town where players can meet with each other to trade goods might be one zone, whereas a forest where the players can fight monsters might be another zone. In this case, it makes no sense for players in the forest to be sent replication information about the players trading in town.

There are a couple of different ways to handle transitions over zone boundaries. One approach is to invoke a loading screen when traveling between zones. This provides enough time for the client to receive replication information regarding all of the objects in the new zone. For a more seamless transition, it may be more desirable to have objects fade in and out as their relevancy changes upon a zone transition. Assuming that the terrain for a zone never changes, the terrain could simply be stored on the client so that the zone behind a player doesn't completely

disappear upon crossing a zone boundary. However, keep in mind that storing terrain on the client may present some security issues. One solution would be to encrypt the data, a topic covered in Chapter 10, "Security."

One drawback of static zones is they are designed around the premise that players will be roughly evenly distributed between the zones in the game. This can be very tough to guarantee in most MMORPGs. Meeting places such as towns will always have a higher concentration of players than an out-of-the-way zone for high-level characters. This problem can be exasperated by in-game events that encourage a large number of players to gather at one specific location—such as in order to fight an especially tough enemy creature. With a high concentration of players in one zone, the experience may be degraded for all the players in the zone.

Solutions to an overcrowded zone may vary by the game. In the MMORPG *Asheron's Call*, if a player attempts to enter a zone with too many players, they are teleported to a neighboring zone. Although perhaps not ideal, this approach is superior to the game crashing due to too many players in one zone. Other games may actually split the zone into multiple instances, a topic discussed later in this chapter.

While viable for shared world games, static zones typically are not used for action games for two main reasons. First, most action games feature combat in a much smaller area than might be seen in an MMO game, though there are some notable exceptions, such as *PlanetSide*. Second, and perhaps more importantly, the pace of most action games means that the delay caused by traversing a zone boundary may be considered unacceptable.

Using the View Frustum

Recall that for a 3D game, the **view frustum** is a trapezoidal prism representing the area of the world that is projected into a 2D image for display. The view frustum is described in terms of an angle representing the horizontal field of view, an aspect ratio, and the distances to the near and far planes. When the projection transform is applied, objects fully enclosed by or intersecting the frustum are visible, whereas all other objects are not.

The view frustum is commonly used in visibility culling. Specifically, if an object is outside the frustum, it is not visible, so no time should be spent sending the object's triangles to the vertex shader. One way to implement frustum culling is to represent the view frustum as the six planes comprising the sides of the frustum. Then a simplified representation of an object, such as a sphere, can be tested against the frustum planes to determine whether or not the object in question is inside or outside the frustum. A detailed discussion of the math behind frustum culling is found in (Ericson 2004).

While visibility culling based on the view frustum makes a great deal of sense, using only the frustum for object scoping in a network game presents some issues when taking into account

latency. For example, if only the frustum is used, objects immediately behind the player would be considered out of scope. This may be problematic if the player quickly turns 180 degrees. It will take some time for a quick turn to be propagated to the server, and for the server to correspondingly send replication updates for objects that would suddenly scope in. One could imagine this would create some unacceptable latency, especially if the object behind the player happens to be an enemy player character. Furthermore, walls are still ignored in this approach. This issue is shown in Figure 9.2.

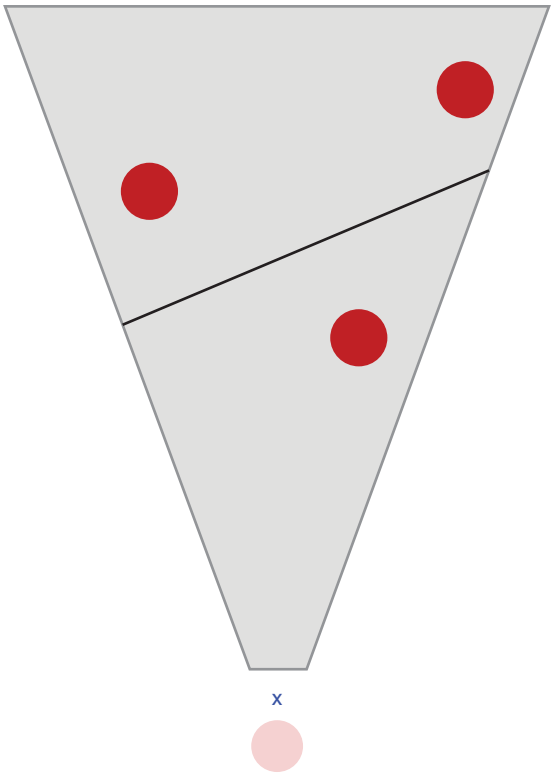

Figure 9.2 An out-of-scope object directly behind the player, X

One solution is to use both the view frustum *and* a distance-based system. Specifically, a distance closer than the far plane could be combined with the frustum. Then any objects that are either within the distance or within the frustum would be considered in scope, and everything else would be out of scope. This means that on a quick turn, far away objects would still go in and out of scope and walls would be ignored, but the scoping of closer objects would not change. An illustration of this approach is shown in Figure 9.3.

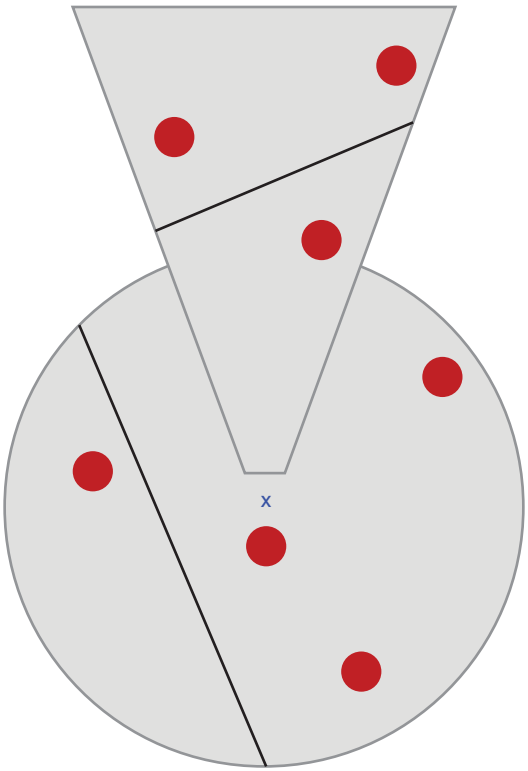

Figure 9.3 Combining a view frustum with a smaller radius to determine relevancy of objects

Other Visibility Techniques

Consider a networked racing game that features a track winding through a city. As would be apparent to anyone who has rode in a car, the amount of road that is visible can vary greatly. On a straight road with flat elevation, it is possible to see far into the distance. However, if the car is turning, the visibility is greatly reduced. Similarly, traveling uphill has lower visibility than traveling downhill. This idea of road visibility can be directly translated into the networked racing game. Specifically, if the server knows the position of a player's car, it can know how far ahead on the track the player can see. This area will likely be much smaller than the area intersecting the view frustum, which will ideally lead to a reduction in the number of objects in scope.

This leads to the concept of a **potentially visible set** (**PVS**). Using a PVS answers the following question: From each location in the world, what is the set of regions that are potentially visible? While this may seem similar to the static zone approach, the region sizes in PVS are typically much smaller than separate zones. A static zone might be a town of several buildings, while a PVS region would be an individual room inside of a building. Furthermore, in a static zone

approach, only objects within the same static zone are considered relevant. This is in contrast to PVS, where neighboring regions that are deemed potentially visible will contain relevant objects.

In a typical implementation of PVS, the world can be divided into a set of convex polygons (or if necessary, a 3D convex hull). An offline process then computes, for each convex polygon, the set of the other convex polygons that are potentially visible. At runtime, the server determines which convex polygon a player is located in. From this convex polygon, the pregenerated sets can be used to determine the set of all objects that are potentially visible. These objects can then be flagged as relevant to the player in question.

Figure 9.4 illustrates what the PVS for the hypothetical racing game might look like. Given the player's location marked by an *X*, the shaded region represents the area that is potentially visible. In an actual implementation, it would be advisable to add a bit of slack in both directions. This way, objects a little bit beyond the potentially visible area would also be marked as in scope. Especially in a racing game where the cars are moving quickly, making sure to account for the latency in the server updating the scoped objects is important.

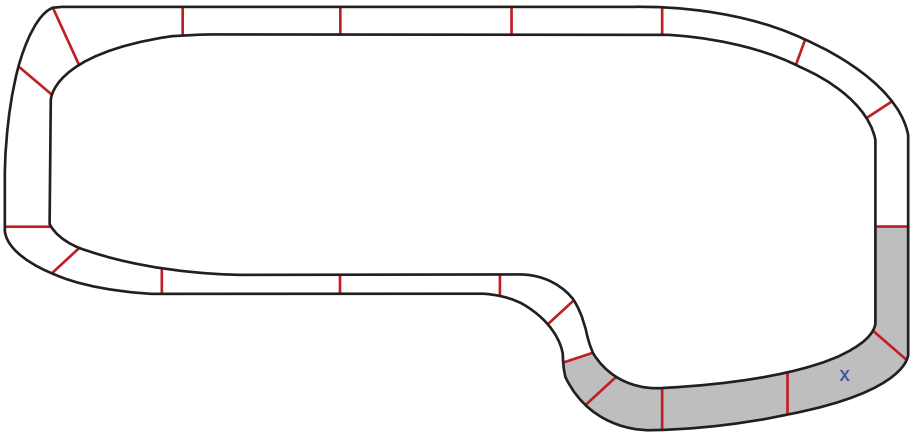

Figure 9.4 A sample PVS in a racing game

The PVS system also works well for a corridor-based first-person shooter, in the vein of *Doom* or *Quake*. For this type of game, it may also be desirable to use a related technique called **portals**. In a portal culling system, each room is a region and each door or window is considered a portal. The frustums created by the portals can be combined with the view frustum to greatly reduce the number of relevant objects. This system requires a greater amount of runtime processing than a PVS, but if your game is already using portals to reduce overdraw on the client, it may not be too difficult to extend the code to work for server-side object scoping.

In a similar vein, some games may merit consideration of hierarchical culling approaches such as BSP, quadtree, or octree. Each of these hierarchical culling techniques partition the objects

in the world using tree data structure. An in-depth discussion of these techniques can be found in (Ericson 2004). Keep in mind that using any of these more advanced techniques for object scoping will significantly increase the amount of time it takes. This is especially true given that the scoping process must be run separately for each client connected to the server. Unless you find your game really struggling to keep up with the volume of object replication, it probably is extreme to use these hierarchical culling systems for object scoping. A well-implemented PVS system should be more than sufficient for most action-oriented games, and many games may not even require the level of detail a PVS system provides.

Relevancy When Not Visible

It is important to note that visibility may not, in all instances, directly correlate with the relevancy of a particular object. Take the example of an FPS where players can throw a grenade. If a grenade explodes in a nearby room, it is important that the grenade be replicated to all clients nearby, even if it is not visible. This is because the client expects to hear the sound of a grenade explosion, even if the grenade is not visible at the moment of explosion.

One approach to solving this issue is to treat grenades differently from other objects. For example, they could be replicated by radius rather than by visibility. Another option is to replicate the explosion effect via RPC to the clients to whom the grenade itself is not relevant. This second approach may reduce the amount of data sent to the clients that need to know about the explosion sound (and potentially the particle effect), but don't need to replicate the actual grenade. This may mean that the grenade explosion information will be replicated to clients that can't actually hear it, but as long as this is a special case and not abused for a large amount of objects, it should not significantly increase bandwidth usage.

If the game is very much audio-based, it may even be possible to compute sound occlusion information on the server in order to determine relevancy. However, realistically such computation is generally done on the client side—it's unlikely a commercial game would actually need to compute audio relevancy with such a degree of accuracy on the server. A radial or RPC-based approach should be fine for most games.

Server Partitioning

Server partitioning or **sharding** is the concept of running multiple server processes simultaneously. Most action games inherently use this approach because each active game has a cap on the number of active players—often within the range of 8 to 16 players. The number of players supported per game is largely a game design decision, but there is also an undeniable technical benefit to such a system. The idea is that by having separate servers, the load on any one particular server should not be overwhelming.

Examples of games that use server partitioning include *Call of Duty*, *League of Legends*, and *Battlefield*. Since each server runs a separate game, there is no gameplay interaction between

the players of two separate games. However, many of these games still have statistics, experience, levels, or other information that is written to a shared database. This means that each server process will have access to some backend database, which can be considered part of the gamer services, a concept covered in more detail in Chapter 12, "Gamer Services."

In a server partitioning approach, it is a common occurrence that one machine is actually capable of running several server processes simultaneously. In many big-budget games, the developer provisions machines in a data center for the purpose of running several server processes. For these games, part of the game's architecture needs to handle distribution of processes to each machine. One approach is to have a master process that decides when server processes should be created, and on which machine. When a game ends, the server process can write any persistent data before exiting. Then when players decide to start a new match, the master process can determine which machine is under the least load, and have a new server process be created on that machine. It is also possible for developers to use cloud hosting for their servers, a configuration discussed in Chapter 13, "Cloud Hosting Dedicated Servers."

Server partitioning is also used as an extension to the static zone approach used in MMOs. Specifically, each static zone, or a collection of static zones, can be run as a separate server process. For example, the popular MMORPG *World of Warcraft* features multiple continents. Each continent runs on a separate server process. When a player transitions from one content to another, the client displays a loading screen while their character state is transferred to the server process for the new continent. Every continent is composed of several different static zones. Unlike changing continents, crossing the boundary between two zones is seamless, because all of the zones on the continent are still running on the same server process. Figure 9.5 illustrates what

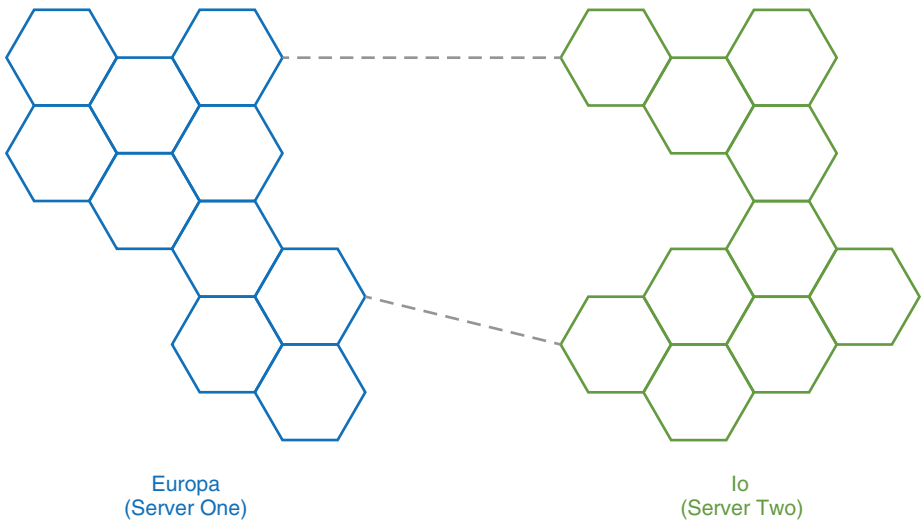

Figure 9.5 Use of server partitioning for separate continents, but not zones, in a hypothetical MMORPG

this type of configuration might look like for a hypothetical MMORPG. Each hexagon represents a static zone, and the dotted lines represent travel points between the two continents.

As with static zones, server partitioning only works well if the players are roughly evenly distributed between each server. If there are too many players on one server, the server can still encounter performance issues. This is not an issue in a game with a fixed player cap, but it can certainly be an issue in an MMO. Depending on the game, there are many different potential solutions to this problem. Some games simply have a server cap and force players to wait in a queue if a server becomes too full. In the case of *Eve Online*, the server slows down the game's time step. This slow-motion mode, called **time dilation**, allows the server to keep all players connected in a situation that it otherwise would not be able to maintain.

Instancing

In **instancing**, one shared game supports several separate instances at once. This term is usually applied to shared world games where all the characters reside on the same server, but may not be playing in the same instance at the same time. For example, many MMORPGs use instancing for dungeon content designed for a fixed number of players. This way, groups of players can experience highly scripted content, free from the interference of other players. In most games that implement this sort of instancing, there is a portal or similar construct that transitions the players from a shared zone into an instance.

Sometimes instancing is also used as a solution for overcrowded zones. For example, *Star Wars: The Old Republic* sets a cap on the number of players that can be in one particular zone. If the player count becomes too high, a second instance of the zone will be forked from the original instance. This does introduce some complexity for players. If two players try to meet in one zone, they might actually end up in two different instances of the zone. In the case of *The Old Republic*, the solution is to allow a player to teleport into a group member's instance, in the event it is different.

From a design perspective, instancing allows for content more in line with single-player or smaller multiplayer games, all while still having characters tied to a shared world. Some games even use instancing as a way to allow for a zone to evolve throughout the course of a quest line. However, the counterargument is that instancing makes the world feel less shared than it might otherwise.

From a performance standpoint, as long as the cost of spinning up an instance is properly managed, instancing can be beneficial. Instancing can guarantee that no more than *X* players are ever relevant at one point in time, especially if the zones can spawn separate instances. It is even possible to combine instancing with server partitioning in order to further decrease the load on specific server processes. Because entering an instance will almost always involve a loading screen for the client, there is no reason the client could not be transferred to a separate server, much how the continents in *World of Warcraft* run on separate server processes.

Prioritization and Frequency

For some games, the performance of the server is not the main bottleneck. Instead, the issue is the amount of data transmitted over the network to the clients. This may especially be an issue for mobile games that need to support a plethora of network conditions. Chapter 5 discussed some ways to solve this problem, such as using partial object replication. However, if testing determines that the amount of bandwidth the game is using is still too high, then there are some additional techniques to consider.

One approach is to assign a priority to different objects. Objects with a higher priority can be replicated first, and lower-priority objects are only replicated if there are no higher-priority objects left to replicate. This can be thought of as a way to ration bandwidth—there is only a limited amount of bandwidth available, so it may as well be used for the most important objects.

When using prioritization, it generally is important to still allow lower-priority objects through on occasion. Otherwise, lower-priority objects will never be updated on clients. This can be accomplished by allowing different objects to have different replication frequencies. For example, important objects might be updated a couple of times per second, but less important objects might only be updated every couple of seconds. The frequency could also be combined with base priority to compute some sort of dynamic priority—in essence, increasing the priority of a lower-priority object if it has been too long since the previous update.

This same sort of prioritization can also be applied to remote procedure calls. If certain RPCs are ultimately irrelevant to the game state, they can be dropped from transmission if there is not enough bandwidth to send them. This is similar to how packets can be sent reliably or unreliably, as discussed in Chapter 2.

Summary

Reducing the volume of data sent to any one client is important as a networked game scales up in size. One way to achieve this is to reduce the total number of objects in scope to a particular client. A simple approach is to deem objects too far away from a client as out of scope, though this one-size-fits-all approach may not work well in all scenarios. Another approach, especially popular in shared world games, is to partition the world into static zones. This way, only players in the same zone are relevant to each other.

It is also possible to leverage visibility culling techniques to reduce the number of relevant objects. While relying solely on the view frustum is not recommended, combining it with a smaller radius can work well. Other games that have clear sectioning of levels, such as corridor-based shooters or racing games might use PVS. With PVS, it is possible to determine which regions are visible from any location in the level. Still other visibility techniques such as portals may see some use on a case-by-case basis. Finally, there are instances where visibility should not be the only criteria for relevancy, such as when a grenade explodes.

Server partitioning can be used to reduce the load on any one server. This can be done both for action games with fixed player caps, and for large shared world games where zones can be placed on separate server processes. Similarly, instancing is a method that forks a shared world into areas that are more manageable from a performance or design standpoint.

There are other techniques, not related to object relevancy, that can be used to limit bandwidth usage of a networked game. One is to assign priority to different objects or RPCs so that the most important information is prioritized first. Another approach is to reduce the frequency that replication updates are sent for all but the most important objects.

Review Questions

1. What are the drawbacks of using only distances to determine object relevancy?
2. What is a static zone, and what are its potential benefits?
3. How can the view frustum be represented for the purposes of culling? What happens if only the frustum is used to determine object relevancy?
4. What is a potentially visible set, and how does this approach differ from static zones?
5. If a shared world game suffers from zone overcrowding, what are some potential solutions to this problem?
6. What are some approaches, other than reducing the number of relevant objects, to reduce the bandwidth requirements of a networked game?

Additional Readings

Ericson, Christer. *Real-Time Collision Detection*. San Francisco: Morgan Kaufmann, 2004.

Fannar, Hallidor. "The Server Technology of EVE Online: How to Cope With 300,000 Players on One Server." Presented at the Game Developer's Conference, Austin, TX, 2008.

CHAPTER 10

SECURITY

Since the first networked games, players have devised ways to gain an unfair advantage. As networked games have become increasingly popular, combating security vulnerabilities has become an important part of providing a safe and fun environment for all players. This chapter takes a look at some of the most common vulnerabilities and the preventative measures that can be taken against them.

Packet Sniffing

In normal network operation, packets are routed through several different computers on their path from the source to destination IP address. At the very least, the routers along the way need to read the header information in the packets in order to determine where to send the packet. And as covered in Chapter 2, sometimes the header addresses may be rewritten for network address translation. However, given the open nature of the data that is transmitted, there is nothing that prevents any of the machines on the route from inspecting all of the data in a particular packet.

Sometimes inspecting the payload contained in a packet might be done in the name of normal network operation. For example, some consumer routers employ **deep packet inspection** in order to implement **quality of service—**a system that prioritizes some packets over others. Quality of service needs to read the packets to determine what it contains. This way, if a packet can be determined to contain peer-to-peer file sharing data, it may be given a lower priority than a packet containing data for a voice-over IP (VoIP) call.

But there also is a form of inspecting these packets that is not necessarily as benign. **Packet sniffing** is a term generally used for the reading of packet data for a purpose other than normal network operation. This can be done for many different purposes including attempting to steal login information or cheating in networked games. The remainder of this section focuses on the specific ways various types of packet sniffing can be combatted in networked games.

Man-in-the-Middle Attack

In a **man-in-the-middle attack**, a computer somewhere on the route from source to destination is sniffing packets, without the knowledge of the source and destination computers. This is shown in Figure 10.1. Practically speaking, there are a few different ways this can occur. Any computer using an unsecured or public Wi-Fi network could have all of its packet information read by another machine on that network. (This is why it is generally a good idea to use an encrypted VPN when on a Wi-Fi network at the local coffee shop). If on a wired

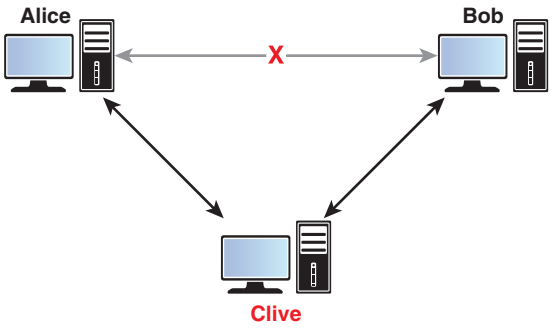

Figure 10.1 A man-in-the-middle attack, with a message between Alice and Bob being read by Clive

network, it could be that a gateway machine is sniffing packets—either because of some sort of malware, or due to a nosy system administrator. And if, for some reason, government agents are targeting your game, it is also possible that software installed at an ISP is attempting to gain access to the data.

Technically, a player could intentionally set up a man-in-the-middle for the purposes of sniffing the game. This may be a concern on a closed platform such as a console, but at least on PC or Mac, you should assume that the player always has access to all of the data transmitted over the network, anyway. So for the rest of this discussion of man-in-the-middle, we will assume that the "man" is a third party unknown to both the source and destination computer.

The general approach to combatting the man-in-the-middle is to encrypt all transmitted data. In the case of a networked game, prior to implementing any sort of encryption system, one should consider whether the game in question contains any sensitive data that needs to be encrypted. If your game contains any microtransactions where a player can purchase in-game items, it absolutely needs to encrypt any data related to purchases. If you are storing or even just processing credit card information, the Payment Card Industry Data Security Standard (PCI DSS) may be a legal requirement. However, even if there are no in-game purchases, any game where a player logs in to an account that saves progress, such as a MOBA or MMO, should encrypt data related to the login process. In both of these cases, there is a monetary incentive for a third party to steal information—whether credit card or login. So it is imperative that your game protects player's valuable data from a man-in-the-middle.

On the other hand, if the only data your game transmits over the network is replication data (or the like), it doesn't really matter if the man-in-the-middle intercepts this data. Thus, you could leave the data unencrypted and it wouldn't be a big issue. That being said, there may still be some value in encrypting the data to prevent host packet sniffing, which is discussed shortly.

If you come to the conclusion that your game does send sensitive data that needs to be protected from outside parties, then using a proven encryption system is the recommended course of action. Specifically, you will want to use **public key cryptography**, a type of cryptography well suited for transmitting secure information. Suppose that Alice and Bob want to transmit encrypted messages to each other. First, before they begin talking to each other, Alice and Bob both generate different private and public keys. The private keys remain private to whoever generated the key—they should never be shared with anyone else. When Alice and Bob first handshake with each other, they will exchange their public keys. Then when Alice sends a message to Bob, she will encrypt the message using Bob's public key. This message can then only be decrypted using Bob's private key. In essence, this means that Alice can send messages to Bob that only he can read, and Bob can send messages to Alice that only she can read. This is the essence of public key cryptography, and is illustrated in Figure 10.2.

In the case of a networked game where there's a login server, the client would have access to the server's public key. When the client wishes to log in to the server, their login and password

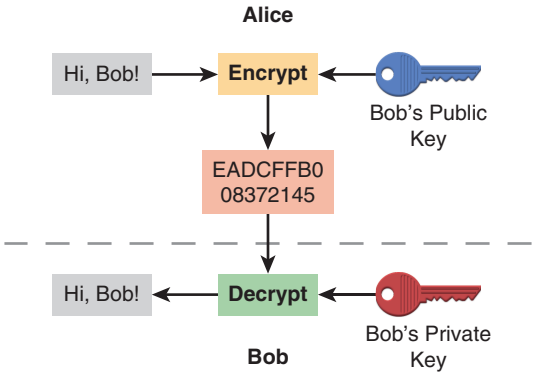

Figure 10.2 Alice and Bob communicate via public key cryptography

are encrypted using the server's public key. This login packet can then only be decrypted by the server's private key, which hopefully only the server knows!

Arguably the most popular public key cryptography system in use today is the RSA system designed in 1977 by Rivest, Shamir, and Adelman. In RSA, the public key is based on a very large number that is a **semiprime**, meaning it is the product of two prime numbers. The private key is then based on the prime number factorization of the semiprime. The system works because no known polynomial-time algorithm exists for integer factorization, and brute-forcing the factorization of a 1024- or 2048-bit number that is the product of two large prime numbers, at this time, is likely impossible even on the most powerful supercomputer in the world.

BREAKING RSA

There are a few scenarios where RSA could be broken, and any of these would be disastrous in the near-term. The first scenario would be the creation of a sufficiently powerful quantum computer. Shor's algorithm is a quantum computer algorithm that can factor integers in quantum polynomial time. However, at time of writing, the most powerful quantum computer in the world can only handle factorization of 21 into 7 and 3, so it may be a few years before a quantum computer factors a 1024-bit number. The other scenario is that a polynomial-time algorithm for integer factorization for standard computers is devised.

The reason why this would be disastrous is because a great deal of secure communication on the Internet relies on RSA or related algorithms. If RSA is broken, this means that many keys used for HTTPS, SSH, and similar protocols would no longer be secure. Most cryptographers are resigned to the fact that RSA will eventually be broken, which is why there is active cryptography research today on systems that even a quantum computer would not be able to solve in polynomial time.

Since RSA is such a well-established cryptography system, it would be a waste of resources to attempt to implement it on your own. Instead, use a trusted open-source implementation of RSA such as the implementation provided in OpenSSL. Because OpenSSL is released under a free software license, even commercial projects should have no issue with using it.

Packet Sniffing on a Host Machine

While only games that transmit sensitive data need to worry about a man-in-the-middle attack, every networked game is susceptible to a host machine intentionally sniffing packets. In this case, encrypting the data is a deterrent but is not a foolproof measure. The reason for this is a game executable on any platform can always be hacked, so encrypting the game data won't prevent someone from learning how to decrypt the data. Somewhere, there must be code within the executable that knows how to decrypt the data the executable is to receive. Once the decryption scheme is determined, the packet data can be read as if it weren't encrypted.

That being said, reverse engineering the decryption code and finding the private key stored in the client does take some time. So one way to make it more difficult for potential cheaters is to still encrypt the data, but change the encryption keys and memory offsets to those keys on a regular basis. This will then require someone to repeat the reverse engineering process every time your game is updated. Similarly, if your game changes the format and ordering of packets on a regular basis, this renders cheats that rely on a specific-packet format obsolete. Once again, this makes players spend time to learn the new format and get the cheats to work again. So, changing the encryption or packet format regularly will make developing cheats for your game more annoying. Hopefully, this means most players give up in developing cheats. But either way, you still have to accept the fact that you will never be able to prevent dedicated individuals from sniffing all of the packets on a host machine.

It's worthwhile to consider what exactly a player packet sniffing on the host machine seeks to accomplish. The player on the host machine is generally trying to utilize an **information cheat**, meaning he or she is trying to glean information that he or she should not know. A common refrain to prevent cheating in this case is to limit the amount of information transmitted to each host. In a client-server game, it is very much possible for the server to limit the data it sends to each client. For example, suppose a networked game supports players moving undetected in stealth mode. If the server still sends replication updates on a character in stealth, then a player could absolutely glean the position of these stealth players from the packets. On the other hand, if replication updates for position pause while a character is in stealth, there will be no way for the client to know the current position of the character.

In general, you should assume any data sent to each host can be examined by a player trying to cheat. Thus if the game ensures that only the critical information relevant to each host is transmitted, then it will minimize the potential for cheating. This will be much easier to enforce in a client-server topology than on a peer-to-peer topology, since peer-to-peer can only work if

all data relevant to the game is sent to every peer. Thus a peer-to-peer game needs to use other approaches to combat cheating.

Input Validation

In contrast to the packet sniffing techniques just covered, **input validation** strives to ensure that no player performs an action that is invalid. This method of cheat prevention can work equally as well for both client-server games and peer-to-peer games. The implementation of input validation boils down to the simple premise that the game should never blindly execute an action from a packet sent over the network. Instead, the action should first be validated to ensure that it is valid at the point in time in question.

For example, suppose that a packet is sent over the network requesting that Player A fire their gun. The receiving machine should never assume that it is valid for Player A to fire. It should first be confirmed that Player A has a weapon, the weapon has bullets, and the weapon is not on a cool down. If any of these conditions are not met, the fire request should be rejected.

It should further be confirmed that when receiving an action for Player A, it is being sent from the client who is responsible for Player A. Recall that the code for both versions of *Robo Cat* in Chapter 6 performed this validation. In the case of the client-server action game, each host address was associated with a client proxy. This way, when moves are received over the network, the server only allows those moves to be applied to that host's corresponding proxy. For the peer-to-peer RTS game, each command is issued by a specific player. When command packets are received over the network, they are associated with that specific peer. When it is time to execute the commands, the peers will reject any commands for units not owned by the peer issuing the commands.

If invalid actions are detected, it may be tempting to boot the offending player. However, you should consider the possibility that the invalid input was accidental, perhaps due to latency or packet loss. For example, suppose that players can cast spells in a particular game. In said game, let's suppose that it is also possible for players to "silence" other players, meaning they cannot cast spells for the duration of the silence. Now suppose that Player A is silenced, which means the server will send an update packet to Player A. It is possible that in the interval prior to receiving the silence packet, Player A transmits a spell cast action. Thus Player A would be transmitting an invalid action, but not due to any nefarious reason. Because of this, it would be a mistake to boot Player A. In general, a more conservative approach of simply rejecting the invalid input will be the proper course of action.

While input validation works well for the server validating a client and a peer validating another peer, it is not particularly easy for the client to validate commands from the server. This wouldn't be an issue for games that run on developer-hosted servers, but it could be an issue for servers that are hosted by players.

In an authoritative server model, only the server has a complete picture of the game state. So if the server tells a client that the client should take damage, the client will have a difficult time validating whether or not this damage is legitimate. This is doubly the case because in a typical configuration, the client has no way to directly communicate with the other clients. Thus, Client A has no way of verifying whether a command actually came from Client B—it has to trust that the server is sending it valid information.

The simplest and only foolproof solution to the problem of bad data from the server is to not allow players to host games. With the advent of cloud hosting, it is viable for even lower budget games to host servers in the cloud. Though there still is a cost, it is substantially less than it would be to run physical servers in a data center. Chapter 13 covers an approach to using cloud hosting for dedicated servers.

However, if your game either does not have the budget for this, or you simply want to give players the option to run their own servers, the solutions become more complex. One approach that has limited success is to maintain peer-to-peer connections between clients. This will increase the complexity of the code base and the runtime bandwidth requirements, but it would allow for some validation of the server's information.

To see how this would work, consider a hypothetical multiplayer dodge ball game. In the standard client-server model, if Client B throws a dodge ball at Client A, this information is first sent from Client B to the server, and then from the server to Client A. To add an additional layer of validation, when Client B throws the dodge ball, it could also send a packet to all of the other clients, notifying the other clients that it is throwing a dodge ball. Then when Client A receives a packet from the server regarding the ball throw, it can validate against the packet it should have received from Client B.

Unfortunately, there is no guarantee such a peer-to-peer validation system for the server will always work. For one, just because each client is able to reach the server does not necessarily mean that each client will be able to reach each other client. This is especially the case when dealing with NAT traversals, firewalls, and so on. Second, even if all clients are reachable to each other, there is no guarantee that the peer-to-peer packets will arrive faster than the packets from the server. So if Client A has to make a decision on whether or not the server's information is correct, it may be possible that the packet from Client B has yet to arrive. This means that either Client A has to wait for Client B's packet, which will delay the updating of the game, or return to square one and accept the server at its word.

Software Cheat Detection

The approaches used to combat both man-in-the-middle attacks and invalid input are both relatively defensive in nature. In the case of man-in-the-middle attacks, the data is encrypted so that it cannot be read. In the case of invalid input, validation code is added to disallow bad

commands. However, there is a much more aggressive approach to combatting players who attempt to cheat.

In **software cheat detection**, software that runs either as part of or external to the game process actively monitors the integrity of the game. Most methods of cheating involve running cheat software on the same machine as the game. Some cheats hook into the game process, other cheats overwrite memory in the game process, still other cheats are third-party applications used for automation, and some cheats even modify data files used by the game. All these different types of cheats can be detected using software cheat detection, which makes it a very powerful method to combat cheaters.

Furthermore, software cheat detection can detect cheats that would otherwise be undetectable. Take the example of a real-time strategy game that's using lockstep peer-to-peer. Most real-time strategy games implement fog of war, which allows each player to only see areas of the map that are near that player's units. However, recall from Chapter 6 that in the lockstep peer-to-peer model, each peer is simulating the entire game state. Thus each peer has, in memory, a complete picture of where all of the units in the game are located. This means that the fog of war is implemented entirely in the local executable, and so the fog of war can be removed by writing a cheat program. This type of cheat is commonly referred to as **map hacking**, and while it is popular in real-time strategy games, any game that uses fog of war can be susceptible to map hacks. What makes this difficult to detect is there likely is very little other peers can do to detect the map hack—the other peers would just see data being transmitted as normal. However, software cheat detection can successfully detect if a map hack is being used.

Another popular cheat is a **bot** that either plays the game in lieu of a player, or assists the player in some way. For example, bots have been used for years in MMOs by players wanting to level up or gain money even while they are sleeping or otherwise away from their computer. In FPS games, some players use aim bots in order to help give them perfect accuracy with every shot. Both of these types of bots can compromise the integrity of the game in major ways, and both can only be detected by software cheat detection.

Ultimately, any multiplayer game that wants to foster a strong community will need to consider using software cheat detection. There are several different software cheat solutions in use today. Some are proprietary and only used by specific game companies, while others are available for use either for free or with a license. The remainder of this section discusses two software cheat detection solutions: Valve Anti-Cheat and Warden. For obvious reasons, the amount of public information available for software cheat detection platforms is fairly limited, so it will be presented in broad strokes. In the event you decide to implement your own software cheat detection, be forewarned that it requires a great deal of understanding of low-level software and reverse engineering. It's also worth noting that even the best software cheat detection platforms can be circumvented. So it is imperative to continuously update the cheat detection in order to stay ahead of those writing cheat programs.

Valve Anti-Cheat

Valve Anti-Cheat (VAC) is a software cheat detection platform that is available to games that utilize the Steamworks SDK. Chapter 12, "Gamer Services" contains an in-depth discussion of the Steamworks SDK as a whole. For now, we will focus the discussion on VAC. At a high level, Valve Anti-Cheat maintains a list of banned users for each game. When a banned user tries to connect to a server that uses VAC, the user is denied access to join the server. Some games will even ban across multiple games—for example, if a player is banned from one game using Valve's Source engine, they are likely banned from all games using the Source engine. This provides an extra amount of deterrence to the system.

At a high level, VAC detects cheaters at runtime by scanning for known cheat programs. There are likely several methods used by VAC to detect a cheat program, but at least one of these methods is to scan the memory of the game process. If a user is detected using a cheat, they typically are not banned immediately. The reason for this is an immediate ban would make it apparent that the cheat is no longer safe to use. Instead, VAC simply creates a list of users to ban at some point in the future. This allows the system to catch as many players as possible who are using the cheat and then ban all of them at once. Players use the term **ban wave** for this practice of delayed bans, and it is commonly used by many software cheat detection platforms.

There is also a related functionality, called **pure servers**, implemented in Valve's Source engine (and thus, can only be used in Source engine games). A pure server validates the content of users upon connection. The server has expected checksums of all of the files that should exist on the client. Upon joining the game, the client must send its file checksums to the server, and if there is a mismatch, the client is booted. This process also happens when a map transition occurs in which the level changes. To account for the fact that some games allow customization to, for example, change the looks of characters, it is also possible to whitelist some files and paths so they are not checked for consistency. Although this system is specifically in Source engine, it would be possible to implement a similar system in your own game.

Warden

Warden is the software cheat detection program created and used by Blizzard Entertainment for all of their games. The functionality of Warden is a bit less transparent than VAC. However, much like VAC, while the game is running Warden scans the computer's memory (among other locations) for known cheat programs. If a cheat is detected, that information is sent back to the Warden server, and the user will be banned at some point in a future ban wave.

One especially powerful aspect of Warden is that updates to its functionality can occur while the game is running. This provides an important tactical advantage—typically cheat users are knowledgeable enough to not use a cheat immediately after a new game patch is released. This is both because the cheat may not even work anymore, and even if it does, it will almost

certainly be detected. However, when Warden updates dynamically, it is possible to catch users that did not realize that Warden has been updated. That being said, some cheat program authors claim that their software is able to detect when Warden is updating, and in this event actually unload the cheat program before Warden finishes its update.

Securing the Server

Another important aspect of security for networked games is protecting the server against attack. This is particularly important for shared world games with central servers, but any game server can be susceptible to attack. So you should plan for certain types of attacks, and make sure you have contingencies in place in the event these attacks occur.

Distributed Denial-of-Service Attack

The goal of a **distributed denial-of-service attack** (DDoS) is to overwhelm the server with requests that it cannot successfully fulfill, ultimately causing the server to be unreachable or otherwise unusable for legitimate users. The reason this works is because too much incoming data will either saturate the server's network connection, or use up so much processing power that the server cannot keep up with actual requests. Pretty much every major networked game or online gamer service has been affected by a DDoS at one time or another.

If you are using your own hardware for game servers, it can be difficult and stressful to mitigate against DDoS attacks. It involves working closely with your ISP, as well as potentially upgrading the hardware and distributing the traffic across different servers. On the other hand, if you use a cloud hosting solution for your servers, as covered in Chapter 13, some of the work to prevent the DDoS attacks is done by the cloud provider. The major cloud hosting platforms all have some level of DDoS prevention built in, and there also are specialized cloud-based DDoS mitigation services that can be purchased. That being said, you should never assume that the cloud hosting provider will completely prevent the potential for DDoS—it is prudent to still invest time in planning for and testing different mitigation strategies.

Bad Data

You should also consider that a malicious user may attempt to send malformed or improper packets to the server. This can be done for a number of reasons, but the simplest reason is the user is attempting to crash the server. However, a more insidious user may be trying to cause the server to execute malicious code through a packet buffer overflow or similar attack.

One of the best ways to secure your game against bad data is to utilize a type of automated testing called **fuzz testing**. In general, fuzz testing is used to discover errors in code that normal unit or quality assurance testing is not likely to discover. For a networked game, you would use fuzz testing to send large amounts of unstructured data to the server. The goal is to

see whether or not sending this data to the server will crash it, and fix any bugs discovered by the process.

In order to find the most bugs, it's recommended to use both fully randomized data as well as more structured data—such as packets that contain expected signatures even if the rest of the payload is random and unstructured. With many iterations of fuzz testing and fixing the bugs caught by fuzz testing, you can try to minimize the possibility of your game being vulnerable to bad data.

Timing Attacks

Any code that compares an expected byte signature or hash versus the received signature is potentially susceptible to a **timing attack**. In this type of attack, information can be learned about the implementation of a particular hashing algorithm or cryptography system based on the amount of time data takes invalid data to be rejected.

Suppose you are comparing two arrays of eight 32-bit integers to determine whether or not they are equal to each other. One array, `a`, represents the expected certificate. The other array, `b`, represents the user's provided certificate. Your first thought might be to write a function as follows:

```
bool Compare(int a[8], int b[8])
{
   for (int i = 0; i < 8; ++i)
   {
      if (a[i] != b[i])
      {
         return false;
      }
   }
   return true;
}
```

The `return false` statement seems like an innocuous performance optimization—if a particular index is a mismatch, there's no reason to continue to compare the remainder of the arrays. However, this code is vulnerable to a timing attack *because* of this early return. For incorrect values of `b[0]`, the `Compare` function will return faster than for correct values of `b[0]`. So if a user tried every possible value of `b[0]`, they could actually determine which value is correct by testing which value causes `Compare` to take longer to return. This process could be repeated for every index, and eventually the user would be able to determine the entire certificate.

The solution to this is to rewrite `Compare` such that it always takes the exact same amount of time to execute, regardless of whether `b[0]` or `b[7]` is a mismatch. One can take advantage of the fact that a bitwise exclusive-or (XOR) yields zero if two values are equivalent. Thus, you can

perform a bitwise XOR between every index of `a` and `b`, and bitwise OR those results together, as in the following rewritten `Compare` function:

```
bool Compare(int a[8], int b[8])
{
   int retVal = 0;
   for (int i = 0; i < 8; ++i)
   {
      retVal |= a[i] ^ b[i];
   }
   return retVal == 0;
}
```

Intrusions

One big concern for server security is a malicious user attempting to break into the server, particularly for shared world games. The goal might be to steal user data, credit card numbers, and passwords. Or even worse, the attacker might try to wipe the entire database for the game, effectively erasing the game from existence. Out of all the server security concerns, intrusions have the most possible nightmare outcomes, and should be taken very seriously.

There are several preventative measures that can be taken to limit potential for intrusion. The biggest step you can take is to make sure all of the software on your servers is kept up-to-date. This includes everything from the operating system, the databases, any software used for automation, web apps, and so on. The reason for this is that old versions may contain critical vulnerabilities that are fixed in newer versions. Staying on top of updates will help limit the options that an attacker will have to infiltrate your game's server. Similarly, limiting the number of services the server machine will reduce the number of potential infiltration points.

The same goes for machines used by developers on your project. A common route for many intrusions is to first break into a personal machine that has access to the central server, and then use this personal machine to springboard into the server system. This is referred to as a **spear phishing attack**. So at a minimum, the operating systems as well as any software that accesses the Internet or network, such as web browsers, should always be updated on all of your developer's machines. Another route to combating the springboard to the server is by greatly limiting how accessible your critical server and data machines are to personal machines. It may be worthwhile to enforce two-factor authentication on your servers, so that simply knowing someone's password is not enough to gain access.

But despite your best efforts to prevent intrusions, you should still make the assumption that your server is vulnerable to a skilled hacker. Thus you want to ensure that any sensitive data you store on your server is as secure as possible. This way, in the event of a breach you can still limit the amount of damage done to your game and the players of your game. For example, user passwords should never be stored as plain text, because someone with access to the database

would then instantly have access to all your user's passwords, which can be particularly bad given how often users reuse passwords across several accounts. The passwords should instead be hashed using an appropriate password-hashing algorithm, such as the Blowfish-derived algorithm bcrypt. Do not use a simpler hashing algorithm such as SHA-256, MD5, or DES to secure your passwords, because these older systems can all be easily broken on modern machines. Similar to encrypting passwords, you should also ensure billing information such as credit cards is stored in a cryptographically secure manner consistent with industry best practices.

As evidenced by the widely publicized intelligence leaks in recent years, often the biggest threat to your server's security may not be an external user. Instead, the greatest threat to your security systems might be a rogue or disgruntled employee. Such an employee may attempt to access or disseminate data that they should not. To combat this, having a comprehensive logging and auditing system is important. This can act both a deterrent, and in the event that something does happen, can provide evidence of criminal wrongdoing.

Finally, you should make certain that any important data is backed up regularly to off-site and physical backups. This way, even in the worst case where your entire database is wiped by a malicious attacker or some other calamity, you still have recourse and can restore to a recent version of the data. Having to restore from a backup is never ideal, but it is still much better than the alternative of permanently losing all of your game's data.

Summary

Most multiplayer game engineers need to be concerned with security on some level. The first thing to consider is the security of data transmissions. Since packets can be intercepted by a man-in-the-middle attack, it is important that sensitive information such as passwords and billing information is encrypted. The recommended approach is to use some form of public key cryptography such as RSA to encrypt the data. For data only relevant to the game state, it is useful to minimize the amount of data that is sent. This is particularly helpful to reduce cheating in client-server games, because it gives clients less information to work with.

Input validation is also important to ensure that no user is performing an action when it is not allowed. Bad input may not always be tied to cheating—it is possible in a client-server game that a client simply has not received the latest updates when it sends out the command. That being said, it is important that all commands that are sent over the network are verified. This can work both for the server validating a client's input and for a peer validating another peer's input. For the case of validating data from the server, the foolproof choice is to disallow players from hosting their own servers.

Although it is a more aggressive approach, software cheat detection can be the best tool to eliminate cheating in a game. A typical cheat detection software will actively scan the memory

of a computer running the game in order to determine if any known cheat programs are also running. If a cheat program is detected, the user in question is banned from the game, usually during a future ban wave.

Finally, it is important to protect your servers from a variety of attacks. Distributed denial-of-service attacks seek to overwhelm servers, and can be combatted in part by using cloud hosted servers. Bulletproofing your server code against bad packets can be accomplished by utilizing fuzz testing. Finally, it is important to take measures such as keeping your server software up-to-date and encrypting sensitive data stored on the server in order to mitigate the risk and damage from a server intrusion.

Review Questions

1. Describe two different ways a man-in-the-middle attack might be executed.
2. What is public key cryptography? How can this be useful to minimize the risk of a man-in-the-middle attack?
3. Give an example of when input validation may result in a false positive, meaning that the input validation thinks the user was trying to cheat even if they were not.
4. How might a game that allows players to host their own servers validate data sent from the server?
5. Why is map hacking in a lockstep peer-to-peer game undetectable without usage of software cheat detection?
6. Briefly describe how the Valve Anti-Cheat system works to combat players who are cheating.
7. Describe two different ways to secure a server from potential intrusions.

Additional Readings

Brumley, David, and Dan Boneh. "Remote timing attacks are practical." *Computer Networks* 48, no. 5 (2005): 701-716.

Rivest, Ronald L., Adi Shamir, and Len Adleman. "A method for obtaining digital signatures and public-key cryptosystems." *Communications of the ACM* 21, no. 2 (1978): 120-126.

Valve Software. "Valve Anti-Cheat System." *Steam Support*. https://support.steampowered.com/kb_article.php?ref=7849-Radz-6869. Accessed September 14, 2015.

CHAPTER 11

REAL-WORLD ENGINES

While larger game studios still largely develop their own internal game engines, for smaller studios it is increasingly common to utilize an off-the-shelf engine. For most genres of networked games, it can be much more time- and cost-effective for a smaller studio to utilize an existing engine. In this case, the code the network engineer will write is at a much higher-level than the majority of this book.

This chapter takes a look at two very popular engines used in many games today, *Unreal 4* and *Unity*, and investigates how networked multiplayer functionality can be implemented in both of these engines.

Unreal Engine 4

The Unreal Engine has existed in one form or another since the 1998 release of the video game *Unreal*. However, over the years the engine has changed in many different ways. This section specifically discusses Unreal Engine 4, which was released in 2014. For the remainder of this chapter, "Unreal" will be used in reference to the engine, not the video game that shares its name. A developer using Unreal generally does not have to worry about lower-level networking details. Instead, the developer is concerned with higher-level gameplay code and making sure that it works correctly in a networked environment. This is analogous to the game simulation layer of the *Tribes* networking model.

Because of this, the majority of this section looks at the higher-level aspects of adding networking to an Unreal Engine game. However, in the interest of completeness, it's worthwhile to look at the lower-level details and how they correspond to many of the topics covered in Chapters 1 to 10. A reader more interested in the lower-level aspects of networking in Unreal Engine can also create a developer account for free at www.unrealengine.com in order to gain full access to the source code.

Sockets and Basic Networking

In order to provide support for a large number of platforms, it is necessary for Unreal to abstract the implementation details of the underlying socket implementation. An interface class called `ISocketSubsystem` has implementations for the different platforms that Unreal supports. This is in some ways analogous to the Berkeley Sockets code presented in Chapter 3. Recall that there are slight differences between the socket API on Windows versus Mac or Linux, so the socket subsystem in Unreal needs to take this into account.

The socket subsystem is responsible for creating sockets as well as addresses. The `Create` function of the socket subsystem returns a pointer to the created `FSocket` class, which can then have data sent and received using standard functions with names such as `Send`, `Recv`, and so on. Unlike the code implemented in Chapter 3, TCP and UDP socket functionality is not provided in separate classes.

Similarly, there is a `UNetDriver` class that is responsible for receiving, filtering, processing, and sending packets. This can be thought as similar to the `NetworkManager` class implemented in Chapter 6, though it is a bit lower level. As is the case with the socket subsystem, there are different implementations based on the underlying transport whether it is IP or a gamer service transport such as that used by Steam which is covered in Chapter 12, "Gamer Services."

There is quite a bit of other lower-level code related to transmitting messages. There is a large set of classes related to transport-agnostic messaging. The details of this are fairly complex, so if you are interested, you should consult the Unreal documentation on this particular feature at https://docs.unrealengine.com/latest/INT/API/Runtime/Messaging/index.html.

Game Objects and Topology

Unreal uses some fairly specific terms to reference the key gameplay classes in the engine, so before diving in deeper it's worthwhile to discuss this terminology. An `Actor` is more or less the base game object class. Every object that exists in the game world, whether static or dynamic, visible or not, is a subclass of `Actor`. One important subclass of `Actor` is `Pawn`, which is an `Actor` that can be *controlled* by something. Specifically, this means that `Pawn` has a member pointing to an instance of a `Controller` class. `Controller` also is a subclass of `Actor`, which is due to the fact that a `Controller` is still a game object that needs to be updated. A `Controller` could be a `PlayerController` or an `AIController`, among other things, depending on what is controlling the `Pawn` in question. A very small subset of the Unreal class hierarchy is illustrated in Figure 11.1.

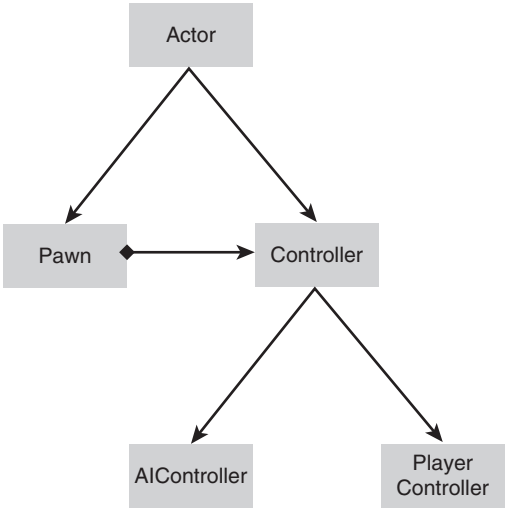

Figure 11.1 Highlights of the Unreal class hierarchy

To solidify how all these classes work together, consider a simple example of a single-player dodgeball game. Suppose a player presses the spacebar to throw a dodgeball. The spacebar input event might be passed to a `PlayerController`. The `PlayerController` will then notify the `PlayerPawn` that it should throw a dodgeball. This will cause the `PlayerPawn` to spawn a `DodgeBall`, which is a subclass of `Actor`. Although there is more happening behind the scenes in the engine, this should provide a basic understanding of how these key classes interact with each other.

For networked games, Unreal only supports the client-server model. There are two different modes the server can run in: dedicated server and listen server. In a **dedicated server**, the

server runs as a process separate from any and all clients. Usually, a dedicated server is run on a separate machine entirely, though that is not a requirement. In the **listen server** mode, one of the game instances is both the server *and* one of the clients. There are some subtle differences between games running in dedicated server mode as opposed to listen server, but that is beyond the scope of this section.

Actor Replication

Given that Unreal uses a client-server model, it follows that there needs to be a way for the server to transmit updates on actors to all of the clients. This is appropriately referred to as **actor replication**. Unreal does a few different things to try to reduce the number of actors that need to be replicated at any one time. As with the *Tribes* model, Unreal tries to determine the set of actors that are relevant to any one client. Furthermore, if there is an actor that will only ever be relevant to one particular client, it is possible to spawn the actor on that client, rather than on the server. An example where this second approach might be utilized is for an actor that is a wrapper for a temporary particle effect. It is also possible to further tweak the relevancy of an actor with a few different flags. For example, `bAlwaysRelevant` will greatly increase the likelihood an actor will be relevant (though contrary to name of the variable, it does not actually *guarantee* the actor will always be relevant).

Relevancy leads to the next important concept of **roles**. In a networked multiplayer game, there will be several separate instances of the game running at once. Each of these instances can query the role for each actor in order to determine who has the authority over the actor. It's important to understand that the role for a particular actor *can be different* depending on the game instance which is querying the role. If we return to the dodgeball example, in a networked multiplayer version of dodgeball, the ball would be spawned on the server. Thus, if the server asked about the role of the dodgeball, it would see that it has role "authority" meaning the server is the final authority for the dodgeball actor. However, every other client would see a role of "simulated proxy," meaning that they are simply simulating the ball and are not the authority of the ball's behavior. The three roles are as follows:

- **Authority.** The game instance is the authority for the actor.
- **Simulated proxy.** When on a client, this means that the server is the authority for the actor. A simulated proxy means that the client may simulate some aspects of the actor, such as movement.
- **Autonomous proxy.** An autonomous proxy is very similar to a simulated proxy, though it implies that it is a proxy that is receiving input events directly from the current game instance, so the player's input should be taken into account when the proxy is simulated.

This does not mean that in a multiplayer game the server is always the authority for every actor. In the case of the local particle effect actor, it may make sense for the client to spawn the actor, in which case the client would see role "authority" and the server would not even know the particle effect actor existed.

However, every actor that the server has role "authority" on will be replicated to all clients, when relevant. Inside of these actors, it is possible to specify which properties should or should not replicate. In this way, bandwidth can be conserved by only replicating properties that are critical to properly simulating the actor. Actor replication in Unreal is *only* ever from the server to the client—there is no way for the client to create an actor and then replicate it to the server (or other clients).

It is also possible for more advanced replication configuration beyond just copying properties. For example, it is possible to only replicate a property based on particular conditions. It is also possible to have a custom function be executed on the client whenever a particular property is replicated from the server. As gameplay code in Unreal Engine 4 is written in C++, the engine uses a complex set of macros to track all of the different replication properties. So when adding a variable in a class' header file, you can also tag the variable with appropriate replication information via the macros. Unreal also has a fairly powerful flowchart-based scripting system called **Blueprint—**surprisingly, much of the multiplayer functionality is also accessible via this scripting system.

Conveniently, Unreal already implements client prediction for actor movement. Specifically, if the `bReplicateMovement` flag is set on an actor, it will replicate and predict movement of simulated proxies based on replicated velocity information. If necessary, it is also possible to override the method by which client prediction is implemented for character movement. However, the default implementation is a good starting point for most games.

Remote Procedure Calls

As in discussed in Chapter 5, "Object Replication," remote procedure calls are instrumental in making replication work. So it should not be a surprise that Unreal has a fairly powerful system for remote procedure calls. There are three types of RPCs in Unreal: server, client, and multicast.

A **server function** is a function that is called on a client, and executed on the server, with one big caveat: The server does not let any client call a server RPC on any actor in the world. This would too easily lead to potential cheating, among other issues. Instead, only the client that is the *owner* of the actor can successfully execute a server RPC on the actor. Note that the owner is *not* the same thing as the game instance that is role authority. Rather, the owner is the `PlayerController` that is associated with the actor in question. For example, if `PlayerController` A controls `PlayerPawn` A, then the client that is driving `PlayerController` A is considered the owner of `PlayerPawn` A. If we return to the dodgeball game example, this means that only Client A can call the `ThrowDodgeBall` server RPC on `PlayerPawn` A—any calls to `ThrowDodgeBall` that Client A might try to invoke on any other `PlayerPawn` would be ignored.

A **client function** is the inverse of a server function. When the server calls a client function, the procedure call is sent to the client who is the owner of the actor in question. For example, when the server determines in the dodgeball game that player C is eliminated, it might invoke a client

function on player C so that the owning client of player C can display the "Eliminated!" message on screen.

As the name implies, a **multicast function** will be sent to multiple game instances. In particular, a multicast function is a function that is called on the server, but executed on the server *and* all of the clients. Multicast functions are used to notify every client about a particular event—for example, a multicast function might be used when the server wants every client to locally spawn a particle effect actor.

Combined, these three different types of RPCs allow for a great deal of flexibility. It's also notable that Unreal provides a choice on whether or not an RPC is reliable. This means that low-priority events could have their RPCs marked as unreliable, which could improve performance when packet loss occurs.

Unity

The Unity game engine was first released in 2005. In the last few years, it has become a very popular game engine used by many developers. As with Unreal, the engine provides some synchronization and RPC functionality built-in, though there are some distinct differences from the approach used by Unreal. Unity 5.1 introduced a new networking library called UNET, and as such this section focuses on this newer library. In UNET, there are two different APIs: a higher-level API that can handle most networked game usage cases, as well as a lower-level transport API that can be used for custom communication over the Internet, as required. The majority of this section will focus on the higher-level API.

While the core Unity game engine is largely written in C++, Unity developers are not provided access to this C++ code. Developers using Unity will typically write the bulk of their code in C#, though it is also possible to use a version of JavaScript, as well. Most serious Unity developers will go with the C# option. Programming gameplay logic in C# instead of C++ presents both advantages and disadvantages, though this is irrelevant to the task at hand.

Transport Layer API

The transport layer API provided by UNET is a wrapper for platform-specific sockets. As one might expect, there are functions for creating connections with other hosts, and this can be used to send and receive data. One of the decisions that can be made when creating a connection is the reliability of the connection. Rather than specifically requesting a UDP or TCP connection, you can instead specify how you wish to use the connection. You can create a communication channel and request one of many values from the `QosType` enum. Possible values include:

- `Unreliable`. Send messages without any guarantees.
- `UnreliableSequenced`. Messages are not guaranteed to arrive, but out-of-order messages are dropped. This is useful for voice communication.

- `Reliable`. The message is guaranteed to arrive as long as the connection is not disconnected.
- `ReliableFragmented`. A reliable message that can be fragmented into several packets. This is useful when wanting to transmit large files over the network, as it can be reassembled on the receiving end.

Connections can be established via the `NetworkTransport.Connect` function call. This will return a connection ID, which can then be used as a parameter for other `NetworkTransport` functions such as `Send`, `Receive`, and `Disconnect`. On a `Receive` call, the returned value is a `NetworkEventType`, which can either encapsulate the data or an event such as a disconnection.

Game Objects and Topology

One big difference from Unreal is the way that game objects are set up in Unity. While Unreal has a relatively monolithic hierarchy when it comes to the game objects and actors, Unity takes a more modular approach. The `GameObject` class in Unity is largely a container for `Component` classes. All behaviors are delegated to the components that are contained in the `GameObject` in question. This can allow for a much better delineation between different aspects of a game object's behavior, though it can sometimes make programming systems more difficult when there are dependencies between multiple components. Normally, a `GameObject` has one or more components that inherit from `MonoBehaviour` that drive any custom functionality for that `GameObject`. So for example, rather than having a `PlayerCat` class that directly inherits from `GameObject`, you would have a `PlayerCat` component that inherits from `MonoBehaviour`. Then the `PlayerCat` component could be attached to any game objects that should behave like a `PlayerCat`.

In the higher-level networking API, Unity uses a `NetworkManager` class to encapsulate the state of a networked game. The `NetworkManager` can run in three different modes: as a standalone client, a standalone (dedicated) server, or a combined "host" that is both a client and a server. This means that Unity essentially supports the same dedicated server or listen server modes that are supported by Unreal.

Spawning Objects and Replication

Because Unity uses a client-server topology, it means that spawning objects in a networked Unity game is very different from spawning them in a single-player game. Specifically, when a game object is spawned on the server via the `NetworkServer.Spawn` function, it means that this game object will be tracked by the server with a generated network instance ID. Furthermore, a game object spawned in this manner should be replicated and spawned to all of the clients as well. In order for the correct game object to be spawned on the client, you are required to register the correct **prefab** for the game object. A prefab in Unity can be thought of as a collection of components, data, and scripts that the game object uses—this can include things like the 3D model, sound effects, and behavior scripts used by the game object. By

registering the prefab on the client, it ensures that all of the object's data is ready for use in the event that the server notifies the client to spawn an instance of that game object.

Once an object is spawned on the server, the properties in its behavior can be replicated to the client via a few different methods. In order for this to work, however, the behavior must inherit from `NetworkBehaviour` instead of the usual `MonoBehaviour`. Once this is done, the simplest way to replicate variables is to flag each variable you wish to replicate with the `[SyncVar]` attribute. This will work on built-in types as well as Unity types such as `Vector3`. Any variables that are marked as `SyncVar`s will automatically have value changes replicated to the clients. There is no need for you to mark the value as dirty. However, keep in mind that while `SyncVar` can also be used for a user-defined struct, the entire contents of the struct will be copied as one set of data. So if you have a struct with 10 members, but only one member changes, it would transmit all 10 members over the network, which may waste bandwidth.

In the event you require more fine-grained control over how variables replicate, you can override the `OnSerialize` and `OnDeserialize` member functions to manually read and write the variables you wish to synchronize. This can allow for customized functionality, but it cannot be combined with `SyncVar`—so you have to choose one or the other.

Remote Procedure Calls

Unity also has support for remote procedure calls, though the terminology is slightly different than the terms used in this book. In Unity, a **command** is an action sent from a client to the server, and only works for objects controlled by that player. In contrast, a **client RPC** function is an action sent from the server to a client. As with `SyncVar`, these types of RPC functions are only supported in subclasses of `NetworkBehaviour`.

The system for flagging functions as either type of remote procedure call is fairly similar to synchronizing variables. To flag a function as a command, it should have the `[Command]` attribute and additionally the function should begin with a prefix `Cmd`, such as `CmdFireWeapon`. Similarly, a function can be flagged with the `[ClientRpc]` attribute and should begin with `Rpc` in the event that it's a client RPC. In either case, the function can be called like a standard function call in C# and it will automatically create the network data and execute it remotely.

Matchmaking

The UNET library also provides some matchmaking functionality that is typically associated with a gamer service, a topic covered in much greater detail in Chapter 12, "Gamer Services." This is in contrast to Unreal, which instead provides wrappers for established gamer services based on the platform in question. The matchmaker in Unity can be used to request and list the current game sessions. Once a suitable session is found, it is then possible to join the

game. This functionality can be added to a `MonoBehaviour` subclass via the `NetworkMatch` class. This will then trigger callbacks such as `OnMatchCreate`, `OnMatchList`, and `OnMatchJoined`.

Summary

For smaller game development studios, using an off-the-shelf game engine can be a reasonable decision. In such a case, the responsibility of the network engineer is at a higher level than the majority of this book. Rather than worrying about how to implement sockets or basic data serialization, the engineer must know how to allow for game functionality to run on a networked game in their engine of choice.

The Unreal Engine has existed for nearly 20 years. The fourth version of the engine, released in 2014, provides full source code in C++. Although there are platform-specific wrappers for functionality such as sockets and addresses, the expectation is generally that the developer will not directly utilize these classes.

Unreal's networking model supports a client-server topology, which can either use a dedicated server or a listen server. The Unreal version of a game object, `Actor`, has a hierarchy that includes many different subclasses. An important aspect of this functionality is the idea of a network role. Authority means the game instance is the authority over an object, whereas simulated and autonomous proxies are used when a client simply is mirroring an object from the server. The `Actor` class also has built-in support for replication of objects. Some functionality, such as movement, can be replicated by setting a Boolean, while custom parameters can also be marked to replicate. Furthermore, there is support for a variety of remote procedure calls.

Unity has existed since 2005, and over the last few years has become a popular game engine. Developers using Unity generally will write all of their gameplay code in C#. In Unity 5.1, a new network library called UNET was introduced, which provides a great deal of high-level networking functionality, though there is also a low-level transport layer that is available.

The transport layer abstracts the creation of sockets and instead allows the developer to transmit data in several modes including reliable and unreliable, but most games implemented in Unity will likely not directly access this. Instead, most developers will use the higher-level API which, as with Unreal, supports both dedicated server and a listen server. All behaviors that need networking support should inherit from the `NetworkBehaviour` class. This adds functionality for replication, which can be handled either via the `[SyncVar]` attribute or custom serialization functions. A similar approach is also utilized for remote procedure calls, both from the server to the client, and the client to the server. Finally, Unity provides some built-in matchmaking functionality that can be used as a lighter-weight option to using a full gamer service.

Review Questions

1. Both Unreal and Unity only provide built-in support for a client-server topology, and not a peer-to-peer topology. Why do you think this is the case?
2. In Unreal, what are the different roles that actors can have in a networked game, and what is their importance?
3. Describe the different usage cases for remote procedure calls in Unreal.
4. Describe how the game object and component model function in Unity. What might be the advantages and disadvantages of such a system?
5. How does Unity implement variable synchronization and remote procedure calls?

Additional Readings

Epic Games. "Networking & Multiplayer." *Unreal Engine.* https://docs.unrealengine.com/latest/INT/Gameplay/Networking/. Accessed September 14, 2015.

Unity Technologies. "Multiplayer and Networking." *Unity Manual*. http://docs.unity3d.com/Manual/UNet.html. Accessed September 14, 2015.

CHAPTER 12

GAMER SERVICES

Most players today have profiles on services such as Steam, Xbox Live, or PlayStation Network. These services provide many features, to both the players and the games, including matchmaking, stats, achievements, leaderboards, cloud-based saves, and more. Because the use of these aptly named gamer services has become so prevalent, players expect that every game, even single-player ones, be integrated with one of these services in some meaningful way. This chapter takes a look at how such services can be integrated into your game.

Choosing a Gamer Service

With so many options, it is worthwhile to consider which gamer service you want to integrate into your game. In some cases, the choice is made for you based on the platform the game is released on. For example, all Xbox One games must be integrated with the Xbox Live gamer service—it's simply not possible to integrate an Xbox One game with PlayStation Network. For PC, Mac, and Linux, however, there are several potential options. Without a doubt, the most popular service on these platforms today is Valve Software's service, Steam. In existence for over 10 years, the Steam platform has a large install base with thousands of available games. Given that *RoboCat RTS* is a PC/Mac game, it made sense to integrate Steam into it.

There are a few prerequisites in order to integrate Steam into your game. First, you must agree to the terms of the Steamworks SDK Access Agreement. This agreement is available online at https://partner.steamgames.com/documentation/sdk_access_agreement. Next, you must register as a Steamworks partner, which involves signing further nondisclosure agreements as well as providing relevant information. Finally, you must get an app ID for your game. An app ID is only provided once you sign up to become a Steamworks partner and your game is greenlit to be offered on Steam.

However, when you complete the first step, agreeing to the Steamworks SDK Access Agreement, you are given access to the SDK files, documentation, and a sample game project (called *SpaceWar!*) that has its own app ID. For demonstration purposes, the code samples provided in this chapter utilize the app ID for *SpaceWar*. This is more than sufficient to understand how to integrate Steamworks into your game once you do complete all of the other steps and receive your own unique app ID.

Basic Setup

Before writing any code specific to a gamer service, consider how you want to integrate the code into your game. A quick option would be to directly add calls to the gamer service code wherever it is needed. So in our case, we would directly call the Steamworks SDK functions in all the files that need to use the gamer service. However, this is discouraged for a couple of reasons. First, this means that every developer on your team may need to have some level of familiarity with Steamworks, because the code using it will be spread throughout your codebase. Second, and more importantly, this makes it far more difficult to integrate a different gamer service into your game. This is particularly a concern for cross-platform games, because, as discussed, different platforms have different restrictions on which gamer service can be used. So even if we know that *RoboCat RTS* is only on PC and Mac for now, if we ever wanted to port it to PlayStation 4, we'd want to make the transition from Steamworks to PlayStation Network as seamless as possible. Having Steamworks code everywhere is counter to this goal.

This leads to a major design decision for the implementation of gamer services in this chapter. The code in the `GamerServices.h` header makes no references to any Steamworks functions or objects, and thus does not need to include the `steam_api.h` header. One of the mechanisms used to accomplish this is the **pointer to implementation** construct, a C++ idiom used to hide the implementation details of a class. When using pointer to implementation, you have a class that contains both a forward declaration of an implementation class *and* a pointer to this implementation class. In this manner, the implementation details of the class are separated from its declaration. The basic components of pointer to implementation that are used in the `GamerServices` class is shown in Listing 12.1. Notice that the class uses a `unique_ptr` rather than a raw pointer, as this is the recommended approach in modern C++.

Listing 12.1 Pointer to Implementation in `GamerServices.h`

```
class GamerServices
{
public:
  //lots of other stuff omitted
  //...

  //forward declaration
  struct Impl;
private:
  //pointer to implementation
  std::unique_ptr<Impl> mImpl;
};
```

It's important to note that the implementation class itself is never fully declared in the header. Instead, the details of the implementation class are declared in the object file—in this case `GamerServicesSteam.cpp`, and this is also where the `mImpl` pointer is initialized. This means that the only place any Steamworks API calls are made is in this single C++ file. In this way, if at any point we wanted to integrate Xbox Live, it would be possible to create another implementation of the `GamerServices` class in `GamerServicesXbox.cpp`. We would then add this new file to our project instead of the Steam implementation, and in theory no other code should have to change.

Although pointer to implementation is a powerful way to abstract away platform-specific details, there is a performance concern that bears mentioning, particularly for games. When using a pointer to implementation, it means that the vast majority of member function calls for the object will require an additional pointer dereference. Pointer dereferences have a cost associated with them. For a class that will have a very high number of member function calls, such as the render device, the performance decrease would be noticeable. However, in the case of the `GamerServices` object, we should not be making a particularly high number of calls per frame. So in this case, trading performance for flexibility is acceptable.

It should also be noted that the available functionality in the `GamerServices` object is a small subset of the overall Steamworks functionality. This is because it only includes wrappers for the functionality that was desired for *RoboCat RTS*—it would certainly be possible to add more to it. However, if you are adding significantly more features, it probably would be a good idea to separate the gamer services code into multiple files. For example, rather than having the handful of peer-to-peer networking functions directly in `GamerServices`, it might make sense to create a `GamerServiceSocket` class that has functionality similar to `TCPSocket` or `UDPSocket`.

Initialization, Running, and Shutdown

Steamworks is initialized by calling `SteamAPI_Init`. This function takes no parameters and returns a Boolean based on the success of the initialization. The code for this is in `GamerServices::StaticInit`. It's noteworthy that the gamer services are initialized in `Engine::StaticInit` *before* the renderer is initialized. This is because one of the features Steam provides is an overlay. The overlay allows for the player to perform actions such as chat with friends or use a web browser without leaving their current game. The way this overlay works is by hooking into OpenGL functionality. This means that in order for the overlay rendering to work correctly, `SteamAPI_Init` must be called before any rendering initialization. If `SteamAPI_Init` succeeds, it will populate a series of global interface pointers. These pointers can then be accessed via global functions such as `SteamUser`, `SteamUtils`, and `SteamFriends`.

Normally, a game on Steam is launched through the Steam client. This is how Steamworks knows the app ID of the game being run. However, during development you won't be launching your game through the Steam client—typically you will be launching through the debugger or as a standalone executable. In order to let Steamworks know the app ID during development, a `steam_appid.txt` file that contains the app ID is placed in the same directory as the executable. However, even though this removes the requirement of launching the game via the Steam client, an instance of the Steam client with a logged-in user must still be running. If you do not have the Steam client, you can get it from the Steam website at http://store.steampowered.com/about/.

Furthermore, in order to test multiple users playing against each other on Steam, you must create multiple test accounts. Testing locally is a bit more complicated than the Chapter 6 version of the game because it is not possible to run multiple instances of Steam on the same computer. So in order to test the multiplayer functionality for this chapter's code, you will need to either use multiple computers or set up a virtual machine.

Since Steamworks often must communicate with a remote server, many of its function calls are asynchronous. In order to notify the application when the asynchronous call has finished, Steamworks utilizes callbacks. In order to ensure that the callbacks are triggered, the game must call `SteamAPI_RunCallbacks` on a regular basis. It is recommended this function is

called once per frame, and so this is what is done in `GamerServices::Update`, which is called once per frame in `Engine::DoFrame`.

Similar to initialization, shutdown of Steamworks is very straightforward via the `SteamAPI_Shutdown` function. This is called in the destructor of `GamerServices`.

For client-server games, it is further necessary to initialize/shutdown game server code via `SteamGameServer_Init` and `SteamGameServer_Shutdown`. This requires also including `steam_gameserver.h`. Dedicated servers can be run in an anonymous mode that does not require a user to be logged in. However, since *RoboCat RTS* only uses peer-to-peer communication, the code for this chapter does not use any of the game server functionality.

User IDs and Names

In the earlier version of *RoboCat RTS* discussed in Chapter 6, player IDs were stored as unsigned 32-bit integers. You may recall that in this older version of the game, the player IDs were assigned by the master peer. When using a gamer service, each player would already have a unique player ID assigned by the service, so it makes little sense to try to assign unique IDs on your own. In the case of Steamworks, unique IDs are encapsulated by the `CSteamID` class. However, it would defeat the purpose of the modularization of the `GamerServices` class if `CSteamID`s were used everywhere. Luckily, `CSteamID`s can be converted to and from unsigned 64-bit integers.

So it follows that changing the player IDs to correspond to the Steam ID first required changing all player ID variables to be of type `uint64_t`. Furthermore, rather than having the player IDs be assigned by the master peer, the `NetworkManager` now initializes each player's ID by querying the `GamerServices` object, specifically by calling the `GetLocalPlayerId` function in Listing 12.2.

Listing 12.2 Basic User ID and Name Functionality

```
uint64_t GamerServices::GetLocalPlayerId()
{
   CSteamID myID = SteamUser()->GetSteamID();
   return myID.ConvertToUint64();
}

string GamerServices::GetLocalPlayerName()
{
   return string(SteamFriends()->GetPersonaName());
}
string GamerServices::GetRemotePlayerName(uint64_t inPlayerId)
{
   return string(SteamFriends()->GetFriendPersonaName(inPlayerId));
}
```

Similar thin wrappers for getting the name of both the local player and another player are also in Listing 12.2. Instead of having the players specify their name, as in the old version of *RoboCat*, it makes more sense to use the name associated with the player on Steam.

It's worth mentioning that although using 64-bit integers for the player ID works for Steamworks, there's no guarantee that it would work for all gamer services. For example, another gamer service might use a 128-bit UUID to identify all the players. In this case, it would be necessary to add a further layer of abstraction. For example, you could create a `GamerServiceID` class that is a wrapper for the underlying representation used for identification by the gamer service.

Lobbies and Matchmaking

The earlier version of *RoboCat RTS* had a nontrivial amount of code associated with all the players meeting up in a pregame lobby. Each new peer had to first say hello to the master peer, then wait to be welcomed, before finally introducing themselves to all the other peers in the game. For this chapter, all the code related to this welcoming process was removed. The reason is that Steam, along with most major gamer services, provides its own lobby feature. Thus, it makes sense to leverage the Steam functionality, especially given that it has far more features than the functionality previously implemented in *RoboCat*.

The basic flow of preparing to play a multiplayer game via Steamworks is roughly as follows:

1. The game searches for a lobby based on application-customizable parameters. These parameters can include game modes or even skill level (if performing skill-based matchmaking).
2. If one or more suitable lobbies are found, the game either selects one automatically or the player is allowed to pick from a list. If no lobby is found, the game can choose to create one for the player. In any event, once a lobby is either found or created, the player joins the lobby.
3. While in the lobby, it's possible to further configure the parameters of the upcoming game such as characters, map, and so on. During this period, other players will hopefully join the same lobby. It's also possible to send chat messages to each other while in the same lobby.
4. Once the game is ready to start, the players join their game and leave the lobby. Normally, this involves connecting to a game server (either a dedicated server or a player-hosted one). In the case of *RoboCat RTS*, there is no server, so the players instead start communicating peer-to-peer with each other before leaving the lobby.

Since *RoboCat* has no menus or mode selection, the game begins a lobby search almost immediately after Steamworks is initialized. The lobby search is encapsulated by the `LobbySearchAsync` function shown in Listing 12.3. The only filter used is for the game name, which ensures that only lobbies for *RoboCat* are found. But any additional filters could be

applied by calling the appropriate filter functions prior to the call to `RequestLobbyList`. Note that the code only asks for one result, because the game will simply auto-join the first lobby it finds.

Listing 12.3 Searching for a Lobby

```
const char* kGameName = "robocatrts";

void GamerServices::LobbySearchAsync()
{
   //make sure it's Robo Cat RTS!
   SteamMatchmaking()->AddRequestLobbyListStringFilter("game",
      kGameName, k_ELobbyComparisonEqual);

   //only need one result
   SteamMatchmaking()->AddRequestLobbyListResultCountFilter(1);

   SteamAPICall_t call = SteamMatchmaking()->RequestLobbyList();
   mImpl->mLobbyMatchListResult.Set(call, mImpl.get(),
      &Impl::OnLobbyMatchListCallback);
}
```

The use of the `SteamAPICall_t` struct in `LobbySearchAsync` requires a bit more explanation. In the Steamworks SDK, all asynchronous calls return a `SteamAPICall_t` struct, which essentially is a handle to the asynchronous call. Once given this handle, you must let Steamworks know what callback function to invoke when the asynchronous call completes. This association between an asynchronous handle and a callback is encapsulated by an instance of `CCallResult`. In this case, the instance is the `mLobbyMatchListResult` member of the implementation class. This member and the `OnLobbyMatchListCallback` functions are defined as follows inside `GamerServices::Impl`:

```
//Call result when we get a list of lobbies
CCallResult<Impl, LobbyMatchList_t> mLobbyMatchListResult;
void OnLobbyMatchListCallback(LobbyMatchList_t* inCallback, bool inIOFailure);
```

In this particular instance, the implementation of `OnLobbyMatchListCallback` has a couple of cases to consider, as shown in Listing 12.4. Note that we check for the `IOfailure` bool. All callbacks have this bool, and it should be assumed that if the value is true, there is an error and the callback should not proceed. However, if a lobby is successfully found, the code requests to enter that lobby. Otherwise, it will create a new lobby. Both of these cases involve an additional asynchronous function call as well, so there are two more callbacks to look at: `OnLobbyEnteredCallback` and `OnLobbyCreateCallback`. To see the implementation of these callbacks, consult the sample code. One important thing to note in these functions is that once the player enters a lobby, the `NetworkManager` is notified via an `EnterLobby` function.

Listing 12.4 Callback When Lobby Search Completes

```
void GamerServices::Impl::OnLobbyMatchListCallback(LobbyMatchList_t* inCallback,
                                                   bool inIOFailure)
{
   if(inIOFailure) {return;}

   //if we find a lobby, enter, otherwise create one
   if(inCallback->m_nLobbiesMatching > 0)
   {
      mLobbyId = SteamMatchmaking()->GetLobbyByIndex(0);
      SteamAPICall_t call = SteamMatchmaking()->JoinLobby(mLobbyId);
      mLobbyEnteredResult.Set(call, this, &Impl::OnLobbyEnteredCallback);
   }
   else
   {
      SteamAPICall_t call = SteamMatchmaking()->CreateLobby( k_ELobbyTypePublic,
                                                             4);
      mLobbyCreateResult.Set(call, this, &Impl::OnLobbyCreateCallback);
  }
}
```

The `NetworkManager::EnterLobby` function ends up not being particularly noteworthy, except that it does call another function in `NetworkManager` called `UpdateLobbyPlayers`. This `UpdateLobbyPlayers` function is called both when the player first enters the lobby, and whenever another player enters or leaves the lobby. This way, the `NetworkManager` can always be sure that it has an up-to-date list of all the players who are currently in the lobby. This is important, because with the removal of the introduction packets, it is the only way that peers can know when the players in the lobby change.

The way to ensure that `UpdateLobbyPlayers` is always called when the players in the lobby change is to use a general callback function. The difference between callbacks and call results is that call results are associated with a specific asynchronous call, whereas general callbacks are not. Thus, general callbacks can be seen as a way to register for notifications regarding a specific event. Conveniently, a callback is posted every time a user leaves or enters a lobby. For these general callbacks, you use a `STEAM_CALLBACK` macro inside the class that will respond to the callback. In this case, it's the implementation class, and the macro looks like this:

```
//Callback when a user leaves/enters lobby
STEAM_CALLBACK(Impl, OnLobbyChatUpdate, LobbyChatUpdate_t,
               mChatDataUpdateCallback);
```

This macro simplifies declaring the name of the callback function and the member variable that encapsulates the callback. This member variable needs to be instantiated in the initializer list of `GameServices::Impl` like so:

```
mChatDataUpdateCallback(this, &Impl::OnLobbyChatUpdate),
```

The implementation for `OnLobbyChatUpdate` then simply calls `UpdateLobbyPlayers` on the `NetworkManager`. Thus, every time a player enters or leaves the lobby, you can guarantee that `UpdateLobbyPlayers` gets called. Since `UpdateLobbyPlayers` also needs some way to grab a map containing the ID and name of every player in the game, the `GamerServices` class provides a `GetLobbyPlayerMap` function, shown in Listing 12.5.

Listing 12.5 Generating a Map of All the Players in a Lobby

```
void GamerServices::GetLobbyPlayerMap(uint64_t inLobbyId,
                                      map< uint64_t, string >& outPlayerMap)
{
   CSteamID myId = GetLocalPlayerId();
   outPlayerMap.clear();
   int count = GetLobbyNumPlayers(inLobbyId);
   for(int i = 0; i < count; ++i)
   {
      CSteamID playerId = SteamMatchmaking()->
                          GetLobbyMemberByIndex(inLobbyId, i);
      if(playerId == myId)
      {
         outPlayerMap.emplace(playerId.ConvertToUint64(),
                              GetLocalPlayerName());
      }
      else
      {
         outPlayerMap.emplace(playerId.ConvertToUint64(),
            GetRemotePlayerName(playerId.ConvertToUint64()));
      }
   }
}
```

If you want to support player chat messages in the lobby, Steamworks provides a `SetLobbyChatMsg` function to transmit messages. Then there is a `LobbyChatMsg_t` callback that can be registered in order to be notified when new messages appear. Since *RoboCat* does not have any interface for chatting, the `GamerServices` class does not provide this functionality. However, it would not be too time consuming to add wrapper functions for chatting if you desire to support it.

Once the game is ready to start, for a client-server game you would use Steamworks function `SetLobbyGameServer` to associate a specific server with the lobby. This server can be associated either via IP address (for dedicated servers) or it can be associated with a Steam ID (for player-hosted servers). This then triggers a `LobbyGameCreated_t` callback to all the players that can be used to let them know it is time to connect to a server.

However, since *RoboCat RTS* is a peer-to-peer game, it does not utilize this server functionality. Instead, once the game is ready to start, there are three steps taken. First, the lobby is set to

be no longer joinable, so no further players can join. Second, the peers begin communication with each other to synchronize the game start. Finally, once the game enters the playing state, everyone leaves. Once all players leave a Steam lobby, the lobby is automatically destroyed. The functions for setting the lobby to be unjoinable and leaving the lobby are declared in `GamerServices` as `SetLobbyReady` and `LeaveLobby`. These functions are very thin wrappers that each calls a single Steamworks function.

Networking

Many gamer services also provide a wrapper for networked communication between two users on the service. In the case of Steamworks, it provides a handful of functions to send packets to other players. The `GamerServices` class wraps some of these functions, as shown in Listing 12.6.

Listing 12.6 Peer-to-Peer Networking via Steamworks

```
bool GamerServices::SendP2PReliable(const OutputMemoryBitStream&
                                    inOutputStream, uint64_t inToPlayer)
{
   return SteamNetworking()->SendP2PPacket(inToPlayer,
                                           inOutputStream.GetBufferPtr(),
                                           inOutputStream.GetByteLength(),
                                           k_EP2PSendReliable);
}

bool GamerServices::IsP2PPacketAvailable(uint32_t& outPacketSize)
{
   return SteamNetworking()->IsP2PPacketAvailable(&outPacketSize);
}

uint32_t GamerServices::ReadP2PPacket(void* inToReceive, uint32_t inMaxLength,
                                      uint64_t& outFromPlayer)
{
   uint32_t packetSize;
   CSteamID fromId;
   SteamNetworking()->ReadP2PPacket(inToReceive, inMaxLength,
                                    &packetSize, &fromId);
   outFromPlayer = fromId.ConvertToUint64();
   return packetSize;
}
```

You may notice that none of these networking functions refers to an IP or socket address. This is intentional, because Steamworks only allows you to send packets to a particular user via their Steam ID, not via IP address. The reason for this is twofold. First, it provides some amount of protection to each user because their IP address is never revealed to any other user on the

service. Second, and perhaps more importantly, this allows Steam to completely handle the network address translation. Recall that in Chapter 6, one of the concerns of directly referencing a socket address was that the address may not be on the same network. However, by using the Steamworks networking calls, this issue is entirely handle by Steam. We request to send a packet to a particular user and Steam will attempt to send the data to this user via NAT punch-through, if possible. In the event that the NAT cannot be traversed, Steam will use a relay server as a fallback. This guarantees that if the destination user is connected to Steam, there will be some route for the packet to reach them.

As an added bonus, Steamworks also provides a couple of different modes of transmission. In the case of *RoboCat RTS*, all the communication for the turn information is critical, so all packets are sent reliably as noted by the `k_EP2PSendReliable` parameter. This mode allows and sends of up to 1 MB at a time, with automatic packet fragmentation and reassembly at the destination. However, it is also possible to request UDP-like communication via `k_EP2PSendUnreliable`. There are also modes to transmit unreliably assuming a connection is already established, and reliably that buffers via the Nagle algorithm.

The first time a packet is sent to a particular user via `SendP2PPacket`, it may take several seconds to be received. This is because the Steam service will take some time to negotiate the route between the source and the destination. Furthermore, when the destination receives a packet from a new user, the destination must accept the session request from the source. This is to disallow unwanted packets from a particular user. In order to accept a session request, a callback is fired every time a session request is received. Similarly, there's another callback that's fired when a session connection fails. The code *RoboCat* uses to handle both of these callbacks is shown in Listing 12.7.

Listing 12.7 Peer-to-Peer Session Callbacks

```
void GamerServices::Impl::OnP2PSessionRequest(P2PSessionRequest_t* inCallback)
{
   CSteamID playerId = inCallback->m_steamIDRemote;
   if(NetworkManager::sInstance->IsPlayerInGame(playerId.ConvertToUint64()))
   {
      SteamNetworking()->AcceptP2PSessionWithUser(playerId);
   }
}

void GamerServices::Impl::OnP2PSessionFail(P2PSessionConnectFail_t* inCallback)
{
   //we've lost this player, so let the network manager know
   NetworkManager::sInstance->HandleConnectionReset(
      inCallback->m_steamIDRemote.ConvertToUint64());
}
```

To account for the fact that the first packet sent to a peer takes some amount of time, the startup procedure for *RoboCat* was adjusted slightly. When the lobby owner/master peer is ready to start the game, they press the return key as before. However, rather than immediately starting the game countdown, the `NetworkManager` enters a new "ready" state. This ready state transmits a packet to all the other peers in the game. In turn, when a peer receives a ready packet, it transmits its own ready packet to all the other peers. This allows all the peers to establish sessions with each other before the game starts.

Once the master peer receives a ready packet from every peer in the game, it then enters the "starting" state and issues a start packet to all the peers, as before. The key observation is that without a ready state, there would not be any sessions established between the peers before the game starts. This would mean that the turn 0 packets would take several seconds to arrive, meaning that every player would end up in a delay state at the start of the game.

As for where this new networking code is used, the packet handling code in the `NetworkManager` was rewritten for this version of *RoboCat*. Rather than using the `UDPSocket` class as before, all packet handling is now done via the functions provided by the `GamerServices` class.

Player Statistics

A popular feature of gamer services is the ability to track various statistics. This way, it is possible to browse your or your friend's profile to see what they have accomplished in various games. To support statistics like this, there typically is some way to query the server for the player's statistics as well as a way to update and write new values to the server. Although it is conceivably possible to always read and write directly from the server, generally it is a good idea to cache the values locally in memory. This is the approach taken by the stats functionality implemented in the `GamerServices` class.

For a Steamworks game, the name and type of stats are defined for a particular app ID on the Steamworks partner site. Since the code for this chapter is using the *SpaceWar* app ID, this means that it is limited to using the stats that were defined for *SpaceWar*. However, the functionality provided would still work for any game's set of stats, you would just have to change the stat definitions to match.

Steam supports three different types of stats. Integer and float stats are, unsurprisingly, integer and floating point values. The third type of stat is called an "average rate" stat. The way this stat works is it provides a sliding window average, with a configurable window size. When you retrieve an average rate stat from the server, you still only receive a single floating point value. However, when you update an average rate stat, you provide a value as well as a duration during which the value was achieved. Steam will then automatically compute for you a new sliding average. This way, it is possible for a stat such a "gold per hour" to still change noticeably as a player's performance improves in the game, even when the player has logged many hours.

When defining the stats for a game on the Steamworks site, one of the properties assigned is the "API Name," which is a string value. Then, all the SDK functions associated with getting and setting a particular stat require you to pass in the string corresponding to the stat. A simple approach would be to have the `GamerServices` functions related to stats simply taking in a string as a parameter. However, the problem with this is that it requires you to remember the exact API names for each stat, and there is always the potential for a typo. Furthermore, since there is a local cache of the stats, each query into the local cache would likely require some sort of hashing. Both these issues can be solved by instead using an enum to define all the possible stats.

One approach would be to define this enum and then separately define an array that contains the API names for each corresponding value in the enum. But the problem with this approach is that if the stats change, it means you now need to remember to update both the enum and the array of strings. There might even be a third place to edit if your game also uses a scripting language, because somewhere in the scripts there would be a redefinition of the same enums. Remembering to keep all three of these in sync is both error-prone and annoying.

Luckily, there is an interesting technique that can be employed, thanks to the C++ preprocessor. This technique, called an **X macro**, allows the stats to be defined in a single location. These definitions are then automatically reused wherever needed, which guarantees synchronization. This completely eliminates any potential for error when changing the stats supported by the game.

The first step to implementing an X macro is to create a definition file that defines each element as well as any additional properties of the element that are important. In this case, the definitions are placed in a separate `Stats.def` file. There are two pieces of data we care about for each stat: its name and the type associated with the stat. Thus, the definitions of the stats look something like this:

```
STAT(NumGames,INT)
STAT(FeetTraveled,FLOAT)
STAT(AverageSpeed,AVGRATE)
```

Next, in `GamerServices.h`, there are two definitions of enums related to stats. One of the enums, `StatType`, is nothing special. It just defines the three `INT`, `FLOAT`, and `AVGRATE` types of stats that are supported. The other enum, `Stat`, is much more complex because it uses the X macro technique. Thus, it is shown in Listing 12.8.

Listing 12.8 Declaring the `Stat` Enum via X Macro

```
enum Stat
{
   #define STAT(a,b) Stat_##a,
   #include "Stats.def"
   #undef STAT
   MAX_STAT
};
```

The code first defines a macro called `STAT` that takes two parameters. Notice that this corresponds to the number of parameters each entry in `Stats.def` contains. In this case, the macro completely ignores the second parameter. This is because the type of the stat does not matter for this particular enum. It then uses the `##` operator to concatenate the characters of the first parameter with the prefix of `Stat_`. Next, we include `Stats.def` which will, in essence, copy and paste the contents of `Stats.def` into the enum's declaration. Since `STAT` is now defined as a macro, it will be replaced by the evaluation of the macro. So for example, the first element of the enum will be defined as `Stat_NumGames`, because that is what the macro `STAT(NumGames,INT)` evaluates to.

Finally, the `STAT` macro is undefined, and the last element of the enum is defined as `MAX_STAT`. So the X macro trick not only defines every member of the enum to correspond to a stat definition in `Stats.def`, it also yields the total number of stats that have been defined.

What makes the X macro so powerful is that the same idiom can be reused anywhere the list of stats is needed. This way, whenever `Stats.def` is modified, a simple recompile of the code will perform macro magic and update all the code that depends on it. Furthermore, because `Stats.def` is a fairly simple file, it could also easily be parsed by a scripting language, should your game use one.

An X macro is used once more when it is time to declare the array of the stats in the implementation file. First, there is a `StatData` structure that represents the locally cached values associated with each stat. To simplify things, each `StatData` has elements for an integer, float, or average rate statistic. This is shown in Listing 12.9.

Listing 12.9 `StatData` Structure

```
struct StatData
{
   const char* Name;
   GamerServices::StatType Type;

   int IntStat = 0;
   float FloatStat = 0.0f;
   struct
   {
      float SessionValue = 0.0f;
      float SessionLength = 0.0f;
   } AvgRateStat;

   StatData(const char* inName, GamerServices::StatType inType):
      Name(inName),
      Type(inType)
   { }
};
```

Next, the `GamerServices::Impl` class has a member array declared as follows:

```
std::array<StatData, MAX_STAT> mStatArray;
```

Notice how the definition of the array takes in `MAX_STAT`, an automatically updated value, as the number of elements it should contain.

Finally, the X macro comes into play during the initializer list of `GamerServices::Impl`. It is used to construct each `StatData` element of `mStatArray`, as shown in Listing 12.10.

Listing 12.10 Initializing `mStatArray` via X Macro

```
mStatArray({
   #define STAT(a,b) StatData(#a, StatType::##b),
   #include "Stats.def"
   #undef STAT
} ),
```

For this second X macro, both elements of the `STAT` macro are used. The first element is converted into a string literal via the # operator, and the second element corresponds to an element of the `StatType` enum. So for example, the `STAT(NumGames,INT)` definition would conveniently evaluate to the following `StatData` instantiation:

```
StatData("NumGames", StatType::INT),
```

The X macro technique is also used for the definitions of the achievements and the leaderboards, since both of those are also instances where multiple values need to stay synchronized in multiple places. That being said, even though this is a powerful technique, it should not be overused as it does not result in particularly readable code. However, it is certainly a useful tool to have in your tool belt for situations like this where it is helpful.

With the X macro implemented, the rest of the stats code falls into place relatively easily. `GamerServices` has a protected function called `RetrieveStatsAsync` that is called when the `GamerServices` object initializes. When the stats are received, `Steamworks` issues a callback. Both of these are in Listing 12.11. Notice how the code for `OnStatsReceived` does not hardcode the stats in anyway—it uses the information stored in the `mStatsArray`, which was auto-generated by the X macro. Also, for debugging purposes, the code logs out the values of the stats when they are first loaded.

Listing 12.11 Retrieving Stats from the Steam Server

```
void GamerServices::RetrieveStatsAsync()
{
   SteamUserStats()->RequestCurrentStats();
}
```

```
void GamerServices::Impl::OnStatsReceived(UserStatsReceived_t* inCallback)
{
   LOG("Stats loaded from server...");
   mAreStatsReady = true;
   if(inCallback->m_nGameID == mGameId && inCallback->m_eResult == k_EResultOK)
   {
      //load stats
      for(int i = 0; i < MAX_STAT; ++i)
      {
         StatData& stat = mStatArray[i];
         if(stat.Type == StatType::INT)
         {
            SteamUserStats()->GetStat(stat.Name, &stat.IntStat);
            LOG("Stat %s = %d", stat.Name, stat.IntStat);
         }
         else
         {
            //when we get average rate, we still only get one float
            SteamUserStats()->GetStat(stat.Name, &stat.FloatStat);
            LOG("Stat %s = %f", stat.Name, stat.FloatStat );
         }
      }

      //load achievements
      //...
   }
}
```

The `GamerServices` class also provides functions to get and update stat values. When a get function is called, the value is immediately returned from the locally cached copy. When a function to update the stat value is called, it will update the locally cached copy and also issue an update request to the server. This ensures that the server and the local cache stay synchronized. The code for `GetStatInt` and `AddToStat` for integers is shown in Listing 12.12. The code for float and average rate stats is rather similar, though as previously mentioned, the average rate stat updates with two values.

Listing 12.12 `GetStatInt` and `AddToStat` Functions

```
int GamerServices::GetStatInt(Stat inStat)
{
   if(!mImpl->mAreStatsReady)
   {
      LOG("Stats ERROR: Stats not ready yet");
      return -1;
   }

   StatData& stat = mImpl->mStatArray[inStat];
```

```
    if(stat.Type != StatType::INT)
    {
        LOG("Stats ERROR: %s is not an integer stat", stat.Name);
        return -1;
    }
    return stat.IntStat;
}

void GamerServices::AddToStat(Stat inStat, int inInc)
{
    //Check if stats are ready
    //...
    StatData& stat = mImpl->mStatArray[inStat ];
    //Check if is integer stat
    //...
    stat.IntStat += inInc;
    SteamUserStats()->SetStat(stat.Name, stat.IntStat );
}
```

RoboCat RTS currently uses the stats to track the number of enemy cats destroyed, as well as the number of friendly cats lost. The code that updates the stats is in `RoboCat.cpp`. This sort of approach where the stat updating code is called wherever necessary is fairly common in games that track stats.

Player Achievements

Another popular feature of gamer services is achievements. These are awarded to players after accomplishing certain feats during the course of a game. Some examples of achievements include one-time events such as defeating a particular boss or winning the game on a certain difficulty. Other achievements are given as a stat accrues over time—for example, an achievement for winning 100 matches. Some dedicated players enjoy achievements so much that they try to unlock everyone.

In Steam, achievements are treated in a similar manner as stats. The set of achievements for a particular game is defined on the Steamworks site, and so as with the stats, *RoboCat* is limited to the set of achievements associated with *SpaceWar*. As with stats, the code for achievements uses X macros. The achievements are defined in `Achieve.def`, and a corresponding `Achievement` enum is derived from this. There also is an `AchieveData` struct and an array of said structs called `mAchieveArray`.

The `RequestCurrentStats` function also grabs the current achievement information from Steam. This means that when the `OnStatsReceived` callback is triggered, the achievement data can also be locally cached. These achievements are copied with a small loop that calls `GetAchievement` to get the Boolean value signifying whether or not the achievement is unlocked:

```
for(int i = 0; i < MAX_ACHIEVEMENT; ++i)
{
   AchieveData& ach = mAchieveArray[i];
   SteamUserStats()->GetAchievement(ach.Name, &ach.Unlocked);
   LOG("Achievement %s = %d", ach.Name, ach.Unlocked);
}
```

Next, there are some fairly simple wrappers for determining whether an achievement is unlocked and actually unlocking an achievement. As was the case with the stats, checking for an unlocked achievement uses the local cache, whereas the function that unlocks the achievement both updates the cache and immediately writes it to the server. This code is shown in Listing 12.13.

Listing 12.13 Checking for and Unlocking Achievements

```
bool GamerServices::IsAchievementUnlocked(Achievement inAch)
{
   //Check if stats are ready
   //...
   return mImpl->mAchieveArray[inAch].Unlocked;
}

void GamerServices::UnlockAchievement(Achievement inAch)
{
   //Check if stats are ready
   //...
   AchieveData& ach = mImpl->mAchieveArray[inAch];
   //ignore if already unlocked
   if(ach.Unlocked) {return;}

   SteamUserStats()->SetAchievement(ach.Name);
   ach.Unlocked = true;
   LOG("Unlocking achievement %s", ach.Name);
}
```

As for when achievements should be unlocked, generally it's a good idea to unlock the achievement soon after it is earned. Otherwise, a player may get confused when they meet the conditions to unlock an achievement, but it doesn't unlock. That being said, for a multiplayer game it may be a good idea to queue up the achievements to be unlocked at the end of the match. This way, the player doesn't potentially get distracted by a UI notification for the achievement.

Since the tracked achievements in *RoboCat RTS* are based on achieving a certain number of kills in game, code to track achievement progress was added in the `TryAdvanceTurn` function in `NetworkManager`. This way, at the end of each turn the game will check whether or not the player has unlocked an achievement.

Leaderboards

Leaderboards are a way to provide rankings for certain aspects of a game, for example, a score or time to complete a particular level. Generally, leaderboard rankings can be browsed both in terms of a global rank as well as ranks relative to your friends on the gamer service. For leaderboards on Steam, they can either be created via the Steamworks website, or they can be created programmatically via an SDK call.

As with stats and achievements, the `GamerServices` implementation uses an X macro to define the enum of leaderboards. In this case, the leaderboards are defined in `Leaderboards.def`. Each entry in this file contains the name of the leaderboard, how the leaderboard should be sorted, and how the leaderboard values should be displayed when viewed on Steam.

The code for retrieving the leaderboards is a bit different than the code for stats or achievements. First, it is only possible to find one leaderboard at a time. When the leaderboard is found, it triggers a call result. So if you want your game to find all the leaderboards in sequence, the call result's code should request a find for the next leaderboard, and repeat this process until all leaderboards are found. This is shown in Listing 12.14.

Listing 12.14 Finding All the Leaderboards

```
void GamerServices::RetrieveLeaderboardsAsync()
{
   FindLeaderboardAsync(static_cast<Leaderboard>(0));
}

void GamerServices::FindLeaderboardAsync(Leaderboard inLead)
{
   mImpl->mCurrentLeaderFind = inLead;
   LeaderboardData& lead = mImpl->mLeaderArray[inLead];
   SteamAPICall_t call = SteamUserStats()->FindOrCreateLeaderboard(lead.Name,
      lead.SortMethod, lead.DisplayType);
   mImpl->mLeaderFindResult.Set(call, mImpl.get(),
      &Impl::OnLeaderFindCallback);
}

void GamerServices::Impl::OnLeaderFindCallback(
   LeaderboardFindResult_t* inCallback, bool inIOFailure)
{
   if(!inIOFailure && inCallback->m_bLeaderboardFound)
   {
      mLeaderArray[mCurrentLeaderFind].Handle =
         inCallback->m_hSteamLeaderboard;
      //load the next one
      mCurrentLeaderFind++;
```

```
        if(mCurrentLeaderFind != MAX_LEADERBOARD)
        {
            GamerServices::sInstance->FindLeaderboardAsync(
                static_cast<Leaderboard>(mCurrentLeaderFind));
        }
        else
        {
            mAreLeadersReady = true;
        }
    }
}
```

The other thing that's different is that finding the leaderboard doesn't download any of the entries from the leaderboard. Instead, it simply gives you a handle to the leaderboard. If you want to download the entries from a leaderboard for display, you provide the handle and the parameters of your download (global, friends only, etc.) to the `DownloadLeaderboardEntries` function in the Steamworks SDK. This will then trigger a call result when the leaderboard entries have downloaded, at which point you can display the leaderboards. A similar process is used for uploading leaderboard scores, via the `UploadLeaderboardScore` function. Code using these two functions can be found in `GamerServicesSteam.cpp`.

Since *RoboCat* doesn't contain a user interface to display the leaderboard, to verify the leaderboard functionality, there are a couple of debug commands. Pressing F10 will upload your current kill count to the leaderboard, and pressing F11 will download the global kill count leaderboard, centered on your current global rank. On a related note, pressing F9 will also reset all the achievements and stats associated with the app ID (in this case, *SpaceWar*).

One cool aspect of leaderboards on Steam is that it is possible to upload user-generated content associated with a leaderboard entry. For example, a quick run through a level could have an associated screenshot or video showing the run. Alternatively, a racing game could have a ghost that players could download to race against. This allows for ways to make the leaderboards more interactive than simply listing top scores.

Other Services

Although this chapter has covered many different aspects of the Steamworks SDK, there still is much more available. There's cloud storage that allows users to synchronize their saves across multiple computers. There's support for a text entry UI for playing in "Big Picture Mode," that's designed for users with only a controller. There's also support for microtransactions and downloadable content (DLC).

There also are many other gamer service options in use today. PlayStation Network works on the PlayStation family of devices such as PlayStation consoles, PlayStation Vita, and PlayStation

mobile phones. Xbox Live has historically been designed for the Xbox consoles, but with Windows 10, it is also available on PC. Other services include Apple's Game Center for Mac/iOS games and the Googles Play Games Services, which work on both Android and iOS devices.

Gamer services sometimes have features specific to them. For example, Xbox Live supports the idea of parties persisting between different games, and the idea that an entire party can start a new game together. Also, on the consoles it's very common to have standardized user interfaces provided via the gamer service. So for example, choosing a save location on the Xbox must always use a specific UI that's provided via a gamer service call.

The concept of what a gamer service should provide is constantly evolving over time. Players will expect these features to be integrated with the latest and greatest games, so whichever gamer service you choose, it is important to spend some time thinking about how best you can leverage the service to improve the experience for your players.

Summary

Gamer services provide a wide range of features for players. Some gamer services are tied to a specific platform, but on a platform such as PC, there are many possible choices. Arguably the most popular gamer service for PC, Mac, and Linux is Steam, and this was the service that was integrated throughout this chapter.

One important decision when adding gamer service code is to devise a method to modularize the code specific to a particular gamer service. This is important because a future port on a different platform may not support the first gamer service you add to your codebase. One way to accomplish this is via the pointer to implementation idiom.

Matchmaking is an important feature provided by most gamer services. This allows users to meet up with each other in order to play a game. In the case of Steamworks, the players first search for and join a lobby. Once the game is ready to start, the players connect to a server (if client-server), or begin communicating with each other (if peer-to-peer) prior to leaving the lobby.

Gamer services also commonly provide a mechanism to send packets of data to other users. This is both to protect users from having their IP addresses revealed, and to allow for the gamer service to perform any necessary NAT punch-through or relaying. In the case of *RoboCat RTS*, the networking code was changed to solely use the Steamworks SDK for sending data. As a bonus, the SDK provides a reliable method of communication. Because the first packet sent to a user has some amount of delay for the session to be established, the startup procedure for *RoboCat* was modified so that peers begin communicating with each other in a "ready" state prior to beginning the game countdown.

Other common features in a gamer service include statistics tracking, achievements, and leaderboards. The `GamerServices` class' implementation of statistics involved declaring

all the possible statistics in an external `Stats.def` file. This information then was used in multiple spots via an X macro, in order to ensure that an enum and an array containing the stats information remained synchronized. A similar approach was used for the implementation of both achievements and leaderboards.

Review Questions

1. Describe the pointer to implementation idiom. What advantages does it provide? What are its disadvantages?
2. What purpose does a callback serve in Steamworks?
3. Roughly describe the lobby and matchmaking procedure used by Steamworks.
4. What are the advantages of networking provided by the gamer service?
5. Describe how the X macro technique works. What benefits and drawbacks does it have?
6. Implement a `GamerServiceID` class, and use this as a wrapper for a Steam ID. Change every reference to a `uint64_t` player ID value to use this new class.
7. Implement a `GamerServicesSocket` class, in the vein of the `UDPSocket` class, which internally uses the Steamworks SDK to send data. Be sure to provide the ability to specify the reliability of communication. Change the `NetworkManager` to use this new class.
8. Implement a menu that displays the stats for the current user. Now implement a leaderboard browser.

Additional Readings

Apple, Inc. "Game Center for Developers." *Apple Developer*. https://developer.apple.com/game-center/. Accessed September 14, 2015.

Google. "Play Games Services." *Google Developers*. https://developers.google.com/games/services/. Accessed September 14, 2015.

Microsoft Corporation. "Developing Games – Xbox One and Windows 10." *Microsoft Xbox*. http://www.xbox.com/en-us/Developers/. Accessed September 14, 2015.

Sony Computer Entertainment America. "Develop." *PlayStation Developer*. https://www.playstation.com/en-us/develop/. Accessed September 14, 2015.

Valve Software. "Steamworks." *Steamworks*. https://partner.steamgames.com/. Accessed September 14, 2015.

CHAPTER 13

CLOUD HOSTING DEDICATED SERVERS

The changing cloudscape means even small studios can afford to host their own dedicated servers. No longer must the fate of a game rely on players with fast net connections hosting fairly administered servers. This chapter explores the pros, cons, and methods necessary to get your game's servers running in the cloud.

To Host or Not To Host

In the early days of online gaming, hosting your own dedicated servers required the Herculean task of acquiring and maintaining large amounts of computer hardware, networking infrastructure, and IT staff. Any hardware ramp-up was a gamble at that. If you overestimated the number of players at launch, you'd end up with racks and racks of machines lying fallow. Worse, if you underestimated, your paying players would be unable to connect due to processing and bandwidth constraints. While you struggled to obtain more last minute equipment, your players would give up, write bad reviews, and warn their friends not to play your game.

Those days of terror are over. Thanks to the abundance of on-demand processing power available from giant cloud host providers like Amazon, Microsoft, and Google, gaming companies are able to spin up and down servers on a whim. Third-party services like Heroku and MongoLabs make deployment even easier by providing server and database management services as needed.

With the huge barrier to entry gone, the proposition of hosting dedicated servers is one that every developer should consider, no matter how small the studio. Despite the lack of upfront server cost, there are still some potential drawbacks to consider:

- **Complexity.** Running a dedicated fleet of servers is more complex than allowing players to host their own. Even though cloud hosts provide the infrastructure and some of the management software, you still need to write custom process and virtual machine management code, as described later in this chapter. Also you have to interface with one or more cloud host providers, which means adapting to changing APIs.
- **Cost.** Even though the cloud decreases upfront and long-term cost significantly, it's still not free. Increased player interest may cover the increased cost, but that's not always the case.
- **Reliance on a third party.** Hosting your game on Amazon or Microsoft's servers means the entire fate of your game rests on Amazon or Microsoft's shoulders. Although hosting companies offer **service-level agreements** that guarantee minimum uptime, these do little to console paying players when every server suddenly goes down at once.
- **Unexpected hardware changes.** Hosting providers usually guarantee to provide hardware that meets certain minimum specifications. This does not prevent them from changing hardware without warning, as long as it is above the minimum specification. If they suddenly introduce a bizarre hardware configuration which you have not tested, it may cause issues.
- **Loss of player ownership.** In the early days of multiplayer gaming, administering your own game server was a matter of pride. It was a way for players to be an important part of the game community, and it created alpha players that spread the gospel of whatever game they were hosting. Even today the culture still lives on in the myriad custom *Minecraft* servers hosted across the land. The intangible benefits of player ownership are lost when the responsibility of running servers moves to the cloud.

Although these downsides can be significant, the benefits often outweigh them:

- **Reliable, scalable, high-bandwidth servers.** Upstream bandwidth comes at a premium, and there's no guarantee that the right players will be hosting the right servers when your other players want to play. With cloud hosting and a good server management program, you can spin up whatever server is necessary, wherever and whenever you need it.
- **Cheat prevention.** If you run all the servers, you can make sure they're running unmodified, legitimate versions of the games. This means all players get a uniform experience not subject to the whims of player administrators. This enables not only reliable rankings and leaderboards, but also persistent player progress based on gameplay, as found in *Call of Duty*, for example.
- **Reasonable copy protection.** Players have a lot of hate for intrusive copy protection and **digital rights management** (**DRM**). However, DRM can be a necessity for some types of games, especially those that rely on microtransactions for revenue, like *League of Legends*. Restricting your game to run on company hosted, dedicated servers provides a de facto, nonintrusive form of DRM. You never have to release server executables to players, which makes it much harder for them to run cracked servers that illegally unlock content. It also allows you to check login credentials for every player, ensuring that they really should be playing your game.

As a multiplayer engineer, the choice of whether to host dedicated servers may be above your pay grade. However, given the value of full stack engineers in the work force, it is important to understand all the implications of the decision so you can weigh in with an informed opinion based on the specifics of the game your team is making.

Tools of the Trade

When working in a new environment, it is most efficient to work with tools tailored for that environment. Backend server development is a rapidly evolving field, with a rapidly evolving set of tools. There are many languages, platforms, and protocols designed to make life easier for the backend developer. At the time of this writing, there is a definite trend for services to use REST APIs, JSON data, and Node.JS. These are flexible and widely accepted tools for server development, and the examples in this chapter make use of them. You can choose different tools for your cloud server hosting development and the basic concepts will remain the same.

REST

REST stands for **representational state transfer**. A REST interface is one that supports the idea that all requests to a server should be self-contained and not rely on previous or future requests for interpretation. HTTP, the protocol that drives the web, is a perfect example of this, and thus typical REST APIs are built heavily around the use of HTTP requests to store, fetch, and modify server-side data. Requests are sent using the common HTTP methods GET and POST, and also

the less common PUT, DELETE, and PATCH. Although various authors have proposed standards on exactly how these HTTP requests need to be structured to qualify as a REST interface, many engineers end up creating interfaces that are REST-flavored to best suit the needs of the users, but do not adhere strictly to any set of REST requirements. Generally, REST interfaces should use the HTTP methods in a fairly consistent manor: GET requests fetch data, POST requests create new pieces of data, PUT requests store data in a specific place, DELETE requests remove data, and PATCH requests edit data directly.

One major advantage of REST interfaces is that they are mostly plain text. Thus, they are human readable, discoverable, and debuggable. In addition, they employ HTTP, which itself uses TCP for transport and thus they are reliable. The self-contained nature of the REST request expands request debuggability, cementing REST as the chosen API style for the backbone of today's cloud services. More details on REST style interfaces and proposed REST standards can be found in the resources listed in this chapter's "Additional Readings" section.

JSON

In the late 1990s and early 2000s, XML was heralded as the universal data exchange format that would change the world. It started to change the world, but it had way too many angle brackets, equal signs, and closing element tags to last forever. These days **JSON** is the new darling for universal data exchange. Standing for **JavaScript object notation**, JSON is actually a subset of the JavaScript language. An object serialized to JSON is exactly the JavaScript that would be needed to recreate that object. It is text based, maintaining all the human readability of XML, but with fewer formatting and tag closing requirements. This makes it even more pleasant to read and debug. Additionally, because it is valid JavaScript, you can paste it directly into a JavaScript program to debug it.

JavaScript works well as a data format for REST queries. By specifying a `Content-Type` of `application/json` in the HTTP header, you can pass data to a POST, PATCH, or PUT request in JSON format, or return data from a GET request. It supports all the basic JavaScript datatypes, such as bools, strings, numbers, arrays, and objects.

Node.JS

Built on Google's V8 JavaScript engine, **Node JS** is an open-source engine for building backend services in JavaScript. The idea behind the language choice was that it would facilitate development of AJAX style websites that also used JavaScript on the frontend. By using the same language on both client and server, developers can write functions and easily switch or share them between layers as necessary. The idea caught on and a very rich community has grown up around Node. Part of its success is due to the vast number of open-source packages available for Node, easily installable through the **Node package manager** (**npm**). Almost all popular services with REST APIs have node package wrappers, making it trivial to interface with the vast array of cloud service providers.

Node itself provides a single-threaded, event-driven JavaScript environment. An event loop runs on the main thread, much like in a video game, dispatching event handlers for any incoming events. These event handlers can in turn make long running requests to the file system, or to external services like databases or REST servers, that execute as asynchronous jobs on non-JavaScript threads. While the jobs execute, the main thread returns to the processing of incoming events. When an asynchronous job completes, it sends an event to the main thread, so the event loop can call an appropriate callback and execute an appropriate JavaScript handler. In this way, Node provides an environment that prevents the pain of race conditions while still allowing for non-blocking asynchronous behavior. As such it is a prime candidate for building services to handle incoming REST requests.

Node ships with a simple built-in HTTP server, but the task of decoding incoming HTTP requests, headers and parameters, and routing them to the appropriate JavaScript functions is usually handled by one of several open-source Node packages dedicated to the purpose. **Express JS** is one such very popular package and the one used by the examples in this chapter. More information on Express JS and Node JS can be found in the resources listed in the "Additional Readings" section.

Overview and Terminology

From the player's perspective, the cloud server spin-up process should be transparent. When a player wants to join a game, the player's client requests info on a match from the matchmaking service endpoint. The endpoint looks for one, and if it can't find one, it should somehow trigger a new server to spin up. It then returns the IP address and port of the new server instance to the client. The client connects there automatically and the player joins the game.

Note that it can be tempting to combine the processes of matchmaking and dedicated server deployment into one giant blob of functionality. It saves on some redundant code and data, and can even aid in performance a little. However, it can be more useful to keep them separate for the single fact that you may want to plug one or more third-party matchmaking solutions into your dedicated server system. Just because your studio hosts its own dedicated servers, does not mean it can't take advantage of third-party matchmaking solutions like Steam, Xbox Live, or PlayStation Network. In fact, depending on the platform for which you're developing, it may be required. For this reason, it is sensible to keep the server deployment module clearly isolated from your matchmaking module.

When your deployment system finishes spinning up a new server, it should simply register itself with the matchmaking system just as a player hosted game server would. After that, the matchmaking system can take over matching players to server instances and your cloud deployment system can focus on what it does best—spinning up and down game instances as necessary.

Server Game Instance

Before going on, it is worthwhile to disambiguate some of the overloaded meanings of the word "server" when used in various contexts. Sometimes "server" refers to an instance of the class in code that simulates the one true version of the game world and replicates it to clients. Other times, it refers to the process listening for incoming connections, hosting that class instance. Still other times, it refers to the physical piece of hardware running that process, as in "check out all the servers I can fit on this rack."

To avoid confusion, this chapter uses the term **server game instance** or just **game instance** to represent the entity that simulates the game world and replicates information to clients. The concept is an abstraction that represents a single reality shared by a group of players playing together. If your game supports 16-player battles, then a server game instance is a running 16-player battle. In *League of Legends* it is typically a 5 versus 5 game in the "Summoner's Rift" level. In matchmaking terms, it is a single match.

Game Server Process

A game instance does not exist in a void. It lives inside a **game server process**, which updates it, manages its clients, interacts with the operating system, and does everything else a process typically does. It is the embodiment of your game, as far as the operating system is concerned. In all previous chapters, the concepts of game server process and game instance were not separated because there was a one-to-one mapping between them. Each game server process was responsible for maintaining only one game instance. However, in the world of dedicated server hosting, that can change.

In properly abstracted code, a single process can manage multiple game instances. As long as the process updates each instance, binds a unique port for each instance, and does not share mutable data between the instances, multiple game worlds can coexist peacefully in the same process.

Multiple instances per process can be an efficient way to host multiple games, because it allows sharing of large immutable resources like collision geometry, navigation meshes, and animation data. When multiple game instances run in their own processes, they each need copies of this data, which can cause unnecessary memory pressure. Games employing multiple instances per process also benefit from finer control of scheduling: By iterating through each instance each update, they can assure a roughly regular update pattern across instances. With multiple processes on the same host, this is not necessarily the case, as the operating system scheduler decides which process is updated when. This is not always a problem, but finer-grained control can be useful at times.

The significant advantages of the multi-instance approach may seem compelling, but the disadvantages of the tactic are just as significant. If a single instance crashes it can bring down the entire process, with all of its contained game instances. This can be particularly nasty if an

individual instance corrupts a shared, supposedly immutable resource. Alternatively, when each game instance runs in a dedicated process, a corrupted or crashing game instance can only bring down itself. In addition, single game instance processes are easier to maintain and test. Engineers developing server code commonly only need a single game instance at a time to test and debug code. If the process supports multiple instances and engineers aren't running them, it leaves a large code path without regular development coverage. A good QA team with a solid test plan can partially accommodate for this, but there is no substitution for engineers having full coverage of production code paths during development. For these reasons, it is most common for game server processes to contain a single game instance.

Game Server Machine

Just as a game instance needs to live in a game server process, a game server process needs to live on a **game server machine**, and just as a single process can host multiple instances, a single machine can host multiple processes. The choice of how many processes to run per machine should depend on the performance requirements of your specific game. For maximum performance, you can run a single process per machine. This ensures the machine's full resources, including CPU, GPU, and RAM, are dedicated to your game process. However, it can be quite wasteful. Each machine needs an operating system and a typical OS is an immense consumer of resources.

Running a unique OS just for a single game process, especially one that contains only a single game instance, can be too expensive an endeavor. Luckily operating systems are designed to support multiple processes with features like protected memory to keep them from interfering with each other's immutable assets. On a modern operating system, it is extremely unlikely that a crashing process can bring down another process on the same game server machine. Therefore, to be cost-efficient, it is typical to run multiple game server processes per server machine—often as many as the performance requirements will allow. Tweaking and tuning server performance and RAM use can pay off many times over if it allows more game processes to be hosted on the same server machine.

Hardware

In the cloud, a game server machine does not necessarily equate to a physical piece of hardware. Instead, **machine images** represent **virtual machines** (**VM**s) which are spun up and down at will, sometimes residing alone on a physical machine, or other times sharing resources with multiple other virtual machines on a physical machine of 16 cores or more. Depending on your cloud hosting provider, and your budget, you may not get to choose how your virtual machines are hosted. At lower price points, they must often share hardware, and are put to sleep when not used for a set amount of time. This can result in erratic performance. At higher price points, you can often specify the exact physical hardware configurations you desire.

WHY VIRTUAL MACHINES?

It may seem odd to have to pack your operating system of choice and game process into a virtual machine just to get hosted in the cloud. However, virtual machines provide an excellent way for cloud service providers to distribute the use of their hardware across their customer base. At Amazon, a single 16-core computer might be running four *Call of Duty* VMs, each requiring 4 cores. As demand for *Call of Duty* wanes at a certain time of day, Amazon might spin down two of those VMs, leaving an underutilized piece of hardware. When a request comes in from EA to spin up an 8-core *Sim City* machine, it can run that VM on the same hardware running the two *Call of Duty* VMs and make the most of its resources.

Virtual machines are also useful when dealing with hardware failure. Because virtual machine images contain the OS and application all as a single package, providers can recover from hardware failure very rapidly by just moving virtual machines from one physical piece of hardware to another.

Local Server Process Manager

A cloud server provisioning system needs a way to start up and monitor game server processes on game server machines. Server machines cannot simply launch the maximum number of game server processes at boot with the expectation that they will run for the uptime of the machine. One process could crash at any time, at which point the virtual machine would be underutilizing its resources. Also, even the most carefully engineered games can end up shipping with memory leaks. Sometimes ship dates are immovable and it is necessary to deploy servers that leak a few megabytes here or there. To keep small memory leaks from accumulating, and also to avoid the problem of resetting game state improperly, it is a good practice to shut down and restart server processes at the end of each match when possible.

If server processes can terminate, the virtual machine needs a way to start them back up. It also needs a way to configure them based on what kind of game players want to start. For all these reasons, a robust provisioning system needs a mechanism through which it can ask a given server machine to start up a server process configured in a specific way. To build such a system, you could hunt and peck around in the details of your operating system to see if there is a built-in way to remotely start and monitor processes. A more cross-platform and less fragile approach, however, is to build a **local server process manager** (**LSPM**).

The LSPM is itself a process that assumes the responsibility of listening for remote commands, spawning server processes as requested, and monitoring those processes to determine which processes the given machine is currently running. Listing 8.1 demonstrates initialization, launch, and kill routes for a simple node.js/express application to manage local server processes.

Listing 13.1 Initialization, Launch, and Kill

```
var gProcesses = {};
var gProcessCount = 0;
var gProcessPath = process.env.GAME_SERVER_PROCESS_PATH;
var gMaxProcessCount = process.env.MAX_PROCESS_COUNT;
var gSequenceIndex = 0;

var eMachineState =
{
    empty: "empty",
    partial: "partial",
    full: "full",
    shuttingDown: "shuttingDown",
};
var gMachineState = eMachineState.empty;
var gSequenceIndex = 0;

router.post('/processes/', function(req, res)
{
    if(gMachineState === eMachineState.full)
    {
        res.send(
        {
            msg: 'Already Full',
            machineState: gMachineState,
            sequenceIndex: ++gSequenceIndex
        });
    }
    else if(gMachineState === eMachineState.shuttingDown)
    {
        res.send(
        {
            msg: 'Already Shutting Down',
            machineState: gMachineState,
            sequenceIndex: ++gSequenceIndex
        });
    }
    else
    {
        var processUUID = uuid.v1();
        var params = req.body.params;
        var child = childProcess.spawn(gProcessPath,
        [
            '--processUUID', processUUID,
            '--lspmURL', "http://127.0.0.1:" + gListenPort,
            '--json', JSON.stringify(params)
        ] );
        gProcesses[processUUID] =
        {
```

```
            child: child,
            params: params,
            state: 'starting',
            lastHeartbeat: getUTCSecondsSince1970()
        };
        ++gProcessCount;
        gMachineState = gProcessCount === gMaxProcessCount?
             eMachineState.full: eMachineState.partial;
        child.stdout.on('data', function (data) {
          console.log('stdout: ' + data);
        });
        child.stderr.on('data', function (data) {
          console.log('stderr: ' + data);
        });
        child.on('close', function (code, signal)
        {
            console.log('child terminated by signal '+ signal);
            //were you at max process count?
            var oldMachineState = gMachineState;
            --gProcessCount;
            gMachineState = gProcessCount > 0 ?
                eMachineState.partial: eMachineState.empty;
            if(oldMachineState !== gMachineState)
            {
                console.log("Machine state changed to " + gMachineState);
            }
            delete gProcesses[processUUID];
        });
        res.send(
        {
            msg: 'OK',
            processUUID: processUUID,
            machineState: gMachineState,
            sequenceIndex: ++gSequenceIndex
        });
    }
});

router.post('/process/:processUUID/kill', function(req, res)
{
    var processUUID = req.params.processUUID;
    console.log("attempting to kill process: " + processUUID);
    var process = gProcesses[processUUID];
    if(process)
    {
        //killing triggers the close event and removes from the process list
        process.child.kill();
        res.sendStatus(200);
    }
```

```
    else
    {
        res.sendStatus(404);
    }
});
```

The LSPM starts by initializing some global variables. `gProcesses` holds a map of all the processes currently being managed, while `gProcessCount` tracks the count. `gProcessPath` and `gMaxProcessCount` are read in from environment variables so they can be easily configured on a machine-by-machine basis. `gMachineState` caches the state of the entire machine, regarding whether it has room for more processes, is full, or is shutting down. The variable holds values from the `eMachineState` object.

The LSPM supports creation of new processes through a POST request to the `/api/processes/` endpoint. Specifically, if the LSPM is running locally and listening on port 3000, you can use the curl web request program to launch a new process configured to host four players with the command line:

```
curl -H "Content-Type: application/json" -X POST -d '{"params":{"maxPlayers":4}}'
http://127.0.0.1:3000/api/processes
```

When the LSPM receives this request, it first checks that it is neither shutting down nor running the maximum number of processes allowed. If that is the case, it creates a new universally unique identifier for the pending process, and uses the Node JS `child_process` module to spawn a game server process. Through command line arguments, it passes the process both the unique ID and any configuration parameters posted by the requester.

Next, the LSPM stores a record of the spawned child process in its `gProcesses` map. The `state` variable is used to track whether the process is currently starting up, or is known to be running. The `lastHeartbeat` variable tracks the last time the LSPM heard from this process, and will come into play in the next section.

After recording the existence of the process, the LSPM sets up some event handlers to receive and log any output from the process. It also sets up a very important listener for the `"close"` event, which removes the process from the `gProcesses` map and reports on any change in `gMachineState`.

Finally, the LSPM responds to the request with the unique process ID and information regarding how many processes are currently running. Remember that the Node event model is single threaded, so there is no worry of a race condition changing the `gProcessCount` or the `gProcesses` hash map during the execution of the function.

With a copy of the unique process ID, the requester can then query information about the process by sending a GET request to the `/processes/:processUUID` endpoint (code not

shown) or shutdown a process by sending a POST to the `/processes/:processUUID/kill` endpoint.

> **warning**
>
> When in production, you want to restrict who can launch and kill servers through your LSPM. One way to accomplish this is by whitelisting all IP addresses that are allowed to send requests directly to the LSPM, and then discarding any incoming requests not from those IP addresses. This will prevent mischievous players from sending process launch commands directly to your LSPM. Alternatively, you can add a security token in the request header and verify its presence before granting any request. Either way, you need to implement some level of security or run the risk of your provisioning system being disrupted.

Process Monitoring

Once the LSPM can launch a process, it needs a way to monitor them. It accomplishes this by listening for heartbeats from the processes. These are periodic packets from the processes indicating that they are still alive. If a set amount of time passes without the LSPM hearing from a particular process, the LSPM assumes that the process has halted, hung, slowed down, or broken in some unacceptable fashion, and it terminates the process. Listing 13.2 demonstrates.

Listing 13.2 Process Monitoring

```
var gMaxStartingHeartbeatAge = 20;
var gMaxRunningHeartbeatAge = 10;
var gHeartbeatCheckPeriod = 5000;

router.post('/processes/:processUUID/heartbeat', function(req, res)
{
    var processUUID = req.params.processUUID;
    console.log("heartbeat received for: " + processUUID);
    var process = gProcesses[processUUID];
    if(process)
    {
        process.lastHeartbeat = getUTCSecondsSince1970();
        process.state = 'running';
        res.sendStatus(200);
    }
    else
    {
        res.sendStatus(404);
    }
});
```

```
function checkHeartbeats()
{
    console.log("Checking for heartbeats...");
    var processesToKill = [], processUUID;
    var process, heartbeatAge;
    var time = getUTCSecondsSince1970();
    for(processUUID in gProcesses)
    {
        process = gProcesses[processUUID];
        heartbeatAge = time - process.lastHeartbeat;
        if(heartbeatAge > gMaxStartingHeartbeatAge ||
            (heartbeatAge > gMaxRunningHeartbeatAge
                && process.state !== 'starting'))
        {
            console.log("Process " + processUUID + " timeout!");
            processesToKill.push(process.child);
        }
    }
    processesToKill.forEach(function(toKill)
    {
        toKill.kill();
    });
}

setInterval(checkHeartbeats, gHeartbeatCheckPeriod);
```

Sending a POST to the `/processes/:processUUID/heartbeat` endpoint registers a heartbeat for the given process ID. When a heartbeat comes in, the LSPM checks the current timestamp and updates the last received heartbeat time of the appropriate process. Once a process sends its first heartbeat, the LSPM changes its state from `starting` to `running` to mark that it has proof that the game process has started.

The `checkHeartbeat` function loops through all processes owned by the LSPM and checks to make sure it has received a recent enough heartbeat. If a process is still in the `starting` state, it may have a slow initialization process to complete, so the function allows it a little extra time to register its first heartbeat. After that, if the latest heartbeat for a process is not within `gMaxRunningHeartbeat` seconds of the current time, it means something terrible happened to the server process. To deal with this, the LSPM attempts to manually kill the child process, in case it is not dead yet. When the process dies, the close event registered earlier removes it from the list of processes. The LSPM calls the `checkHeartbeat` function every `gHeartbeatCheckPeriod` ms by means of the `setInterval` call at the bottom of the script.

To send a heartbeat to the LSPM, each process needs to make a POST request to its LSPM heartbeat endpoint at least once every `gHeartbeatCheckPeriod` seconds. To send a REST request from a C++ program, you can build the http request as a string and then send it to the appropriate LSPM's port using the `TCPSocket` class described in Chapter 3. For example,

if the LSPM, listening on port 3000, launched a process with the `-processUUID` command line parameter `49b74f902d9711e5-8de0f3f32180aa49`, then the process can register heartbeats by sending the following string via TCP to port 3000:

```
POST /api/processes/49b74f902d9711e5-8de0f3f32180aa49/heartbeat HTTP/1.1\r\n\r\n
```

Notice the two end line sequences in a row used to denote the end of the http request. For more on the textual format of HTTP requests, see the "Additional Readings" section. Alternatively, for a more turn-key solution, you can integrate a third-party C++ REST library like Microsoft's open-source, cross-platform C++ REST SDK library. Listing 13.3 demonstrates how to send a heartbeat using the C++ REST SDK.

Listing 13.3 Sending a Heartbeat with the C++ REST SDK

```
void sendHeartbeat(const std::string& inURL,const std::string& inProcessUUID)
{
    http_client client(U(inURL.c_str()));
    uri_builder builder(U("/api/processes/" + inProcessUUID + "/heartbeat"));
    client.request(methods::POST, builder.to_string());
}
```

To check on the results of the heartbeat, you can append continuation tasks to the task returned by the request invocation. The C++ REST SDK offers a rich library that provides not only asynchronous, task-based HTTP request functionality, but also server functionality, JSON parsing, WebSocket support, and more. For more on the C++ REST SDK and what it can do, refer to the resources listed in the "Additional Readings" section.

> **note**
>
> REST requests are not the only way to send heartbeats to an LSPM. If you prefer, the LSPM can open a TCP or even UDP port directly in Node, and the server process can send very small heartbeat packets without the overhead of HTTP. Or, the game can just write heartbeat data to its log file and the LSPM can monitor that. However, given that your game will probably end up needing a REST API to talk to one or more other services, and the ease of debugging REST data, and the fact that the LSPM is already listening for incoming REST requests, it reduces complexity to just send heartbeats via REST.

Virtual Machine Manager

By facilitating remote startup and monitoring of an arbitrary number of processes on a virtual machine, the LSPM solves a significant portion of the cloud hosting problem. However, it does nothing to actually provision the machines themselves. To do this, you need a **virtual machine**

manager (**VMM**). The VMM is responsible for tracking all the LSPMs, requesting LSPMs to spawn game processes when necessary, and spinning up and down entire virtual machines, with their associated LSPMs.

To provision a new virtual machine with a cloud provider, the VMM must identify what software to run on the machine. It does this by specifying a **virtual machine image** (**VMI**). The VMI represents the contents of the disk drive that the VM should boot. It contains the OS, the process executables, and any initialization scripts to run at boot. Each cloud host provider has a slightly different VMI format they prefer, and usually custom tools to create the VMs. To prepare for VM provisioning, you must create a VMI with your chosen OS, your compiled game server executable and data, your LSPM, and any necessary assets.

> **note**
>
> Although each cloud provider has their own VMI format, many may soon be standardizing on the Docker Container format. For more on the Docker standard, see the "Additional Readings" section.

Asking a cloud hosting provider to spin up a VM from a VMI comes down to the details of the provider. Providers typically have a REST API for this purpose, with wrappers in common backend languages like JavaScript and Java. Because you may need to switch cloud host providers, or use multiple ones in multiple regions, it is a good idea to cleanly abstract the details of the provider API from your VMM code.

In addition to simply spinning up VMs when necessary, a VMM must be able to request new processes from the LSPM on each VM. It must also ask the cloud provider to shut down and deprovision any VMs no longer in use. Finally, it must monitor the health of all the VMs it manages to make sure none leak in case of error. Although Node is single threaded, the asynchronous interactions between requester, VMM, and LSPM present ample opportunity for a variety of race conditions. In addition, even though TCP is reliable, each REST request is on its own connection, which means communications can arrive out of order. Listing 13.4 shows the initialization and data structure of the VMM.

Listing 13.4 Initialization and Data Structures

```
var eMachineState =
{
    empty: "empty",
    partial: "partial",
    full: "full",
    pending: "pending",
    shuttingDown: "shuttingDown",
    recentLaunchUnknown: "recentLaunchUnknown"
};
```

```
var gVMs = {};
var gAvailableVMs = {};

function getFirstAvailableVM()
{
    for( var vmuuid in gAvailableVMs)
    {
        return gAvailableVMs[vmuuid];
    }
    return null;
}

function updateVMState(vm, newState)
{
    if(vm.machineState !== newState)
    {
        if(vm.machineState === eMachineState.partial)
        {
            delete gAvailableVMs[vm.uuid];
        }
        vm.machineState = newState;
        if(newState === eMachineState.partial)
        {
            gAvailableVMs[vm.uuid] = vm;
        }
    }
}
```

The core data of the VMM lives in two hash maps. The `gVMs` hash map contains all currently active VMs managed by the VMM. The `gAvailableVMs` map is the subset of VMs which are available for spawning a new process. That is, they are not shutting down, starting up, currently spawning a process, or already at max process count. Each VM object needs the following members:

- **`machineState`.** Representing the current state of the VM, this holds one of the members of the `eMachineStates` object. These states are a superset of the `eMachineStates` the LSPM uses, containing a few more states that are only relevant to the VMM.
- **`uuid`.** This is the VMM-assigned unique identifier for the VM. When spawning the VM, the VMM passes the uuid to the LSPM so that the LSPM can tag any updates it sends the VMM.
- **`url`.** The url stores the IP address and port of the LSPM on the VM. The IP and possibly the port are assigned by the cloud service provider whenever a VM is provisioned. The VMM must store it so it can communicate with the LSPM on the VM.
- **`lastHeartbeat`.** Similar to how the LSPM listens for process heartbeats, the VMM listens for LSPM heartbeats. This stores the time the last heartbeat was received.
- **`lastSequenceIndex`.** Because each REST request can come in on its own TCP connection, it's possible for them to arrive out of their original order. To make sure the VMM ignores

any stale updates from an LSPM, the LSPM tags each piece of communication with an increasing sequence index, and the VMM ignores any incoming data with a sequence index less than the `lastSequenceIndex`.

- **`cloudProviderId`.** This stores the VMs identity as far as the cloud service provider is concerned. The VMM uses this when asking the provider to deprovision the VM.

When it's time to spawn a new VM, the `getFirstAvailableVM` function finds the first VM in the `gAvailableVMs` map and returns it. The `updateVMState` function is responsible for transitioning VMs into and out of the `gAvailableVMs` map as their state changes. For consistency, the VMM should only change the `state` of a VM via the `updateVMState` function. With the necessary data structures in place, Listing 13.5 shows the REST endpoint handler that actually spawns a process. It provisions a VM first if necessary.

Listing 13.5 Spawning a Process and Provisioning a VM

```
router.post('/processes/', function(req, res)
{
    var params = req.body.params;
    var vm = getFirstAvailableVM();
    async.series(
    [
        function(callback)
        {
            if(!vm ) //spin up if necessary
            {
                var vmUUID = uuid.v1();
                askCloudProviderForVM(vmUUID,
                    function(err, cloudProviderResponse)
                {
                    if(err) {callback(err);}
                    else
                    {
                        vm =
                        {
                            lastSequenceIndex: 0,
                            machineState: eMachineState.pending,
                            uuid: vmUUID,
                            url: cloudProviderResponse.url,
                            cloudProviderId: cloudProviderResponse.id,
                            lastHeartbeat: getUTCSecondsSince1970()
                        };
                        gVMs[vm.uuid] = vm;
                        callback(null);
                    }
                });
            }
```

```
            else
            {
                updateVMState(vm, eMachineState.pending);
                callback(null);
            }
        },
        //vm is valid and in the pending state so no other can touch it
        function(callback)
        {
            var options =
            {
                url: vm.url + "/api/processes/",
                method: 'POST',
                json: {params: params}
            };

            request(options, function(error, response, body)
            {
                 if(!error && response.statusCode === 200)
                 {
                     if(body.sequenceIndex > vm.lastSequenceIndex)
                     {
                         vm.lastSequenceIndex = body.sequenceIndex;
                         if(body.msg === 'OK')
                         {
                             updateVMState(vm, body.machineState);
                             callback(null);
                         }
                         else
                         {
                             callback(body.msg); //failure- probably full
                         }
                     }
                     else
                     {
                         callback("seq# out of order: can't trust state");
                     }
                 }
                 else
                 {
                    callback("error from lspm: " + error);
                 }
            });
        }
    ],
    function(err)
    {
        if(err)
        {
```

```
            //if vm is set, make sure it's not stuck in the pending state
            if(vm)
            {
                updateVMState(vm, eMachineState.recentLaunchUnknown);
            }
            res.send({msg: "Error starting server process: " + err});
        }
        else
        {
            res.send({msg: 'OK'});
        }
    });
});
```

note

This endpoint handler makes use of the `async.series` function, which is a utility in the popular **async** JavaScript library. It takes an array of functions, and a final completion function as parameters. It calls each of the functions in the array in order, waiting until they call their respective `callback` function to proceed. When the series is done, `async.series` calls the completion function. If any one of the functions in the array passes an error to its callback function, `series` immediately passes the error to the completion function and aborts the calling of any more functions in the array. async contains many other useful high-order asynchronous constructs and is one of the most depended upon packages in the Node community.

The handler also makes use of the **request** library for making REST requests to the LSPM. request is a full featured HTTP client library, similar in power and functionality to the curl command line utility. Like async, it is also a top library in the Node community and one worth learning. More information on both the async and request libraries can be found in the "Additional Readings" section.

Posting game parameters to the `/processes/` endpoint of the VMM triggers the launch of a game process with those parameters. The handler has two main sections: the VM procurement and then the process spawn. First, the handler checks the `gAvailableVMs` map to see if there is a VM available to spawn a process. If there is not, it creates a unique ID for a new VM and asks the cloud provider to provision it. The function `askCloudProviderForVM` is highly dependent on the specific cloud provider used, and so is not listed here. It should call the cloud provider's API for provisioning a VM, use the image that contains the game and the LSPM, and then start the LSPM, passing the VM identifier as a parameter.

Whether the VM is started up fresh, or already available, the handler sets its state to `pending`. This makes sure that the VMM will not try to start up another process on it while there is one

currently starting up. The single-threaded nature of Node prevents traditional race conditions, but because the endpoint handler uses asynchronous callbacks, it is possible another process-launch request might arrive before the current one is fulfilled. In that case, it is necessary for the request to be handled by a different VM to avoid overlapping state updates. To facilitate this, the change to `pending` state removes the VM from the `gAvailableVMs` map.

With the VM in `pending` state, the handler sends a REST request to the VM's LSPM to launch a game process. If the launch succeeds, the handler sets the VM state to the new state returned by the LSPM—it should be either `partial` or `full`, depending on how many game processes the VM is currently hosting. If there is a bad or missing response from the LSPM, the VMM cannot know the resultant state of the VM. It is possible that the process did not launch before the error was returned, or that the process did launch and the response was lost somewhere in the network. Even though TCP is reliable, HTTP clients and servers have timeouts. Loose network cables, persistent traffic spikes, or bad Wi-Fi signals can cause communication to time out. In the case of indeterminate error, the handler sets the VM's state to `recentLaunchUnknown`. This removes the server from the `pending` state so that the heartbeat monitoring system, explained later, can either restore the VM to a known state or kill it. It also keeps the VM out of the `gAvailableVMs` map, because its availability is unknown.

If all goes well, the handler finally responds to the original request with the message "OK," meaning the new game process on a remote VM has launched.

Virtual Machine Monitoring

Because an LSPM can hang or crash at any time, the VMM needs to monitor each LSPM for heartbeats. To ensure that the VMM's perception of the LSPM state remains accurate, the LSPM can send state updates with each heartbeat, tagged with an increasing `sequenceIndex` to help the VMM ignore out-of-order heartbeats. When a heartbeat indicates that an LSPM is running no processes, the VMM initiates a shutdown handshake with the LSPM. The handshake prevents race conditions that might cause the LSPM to launch a process while the VMM is trying to shut it down. Due to both the shutdown handshake and the state included in the heartbeat, the system is somewhat more complicated than the one the LSPM uses to monitor processes. Listing 13.6 demonstrates the VMM heartbeat monitoring system.

Listing 13.6 VMM Heartbeat Monitoring

```
router.post('/vms/:vmUUID/heartbeat', function(req, res)
{
    var vmUUID = req.params.vmUUID;
    var sequenceIndex = req.body.sequenceIndex;
    var newState = req.body.machineState;
    var vm = gVMs[vmUUID];
    if(vm)
    {
```

```
        var oldState = vm.machineState;
        res.sendStatus(200); //send status now so lspm can close connection
        if(oldState !== eMachineState.pending &&
            oldState !== eMachineState.shuttingDown &&
            sequenceIndex > vm.lastSequenceIndex)
        {
            vm.lastHeartbeat = getUTCSecondsSince1970();
            vm.lastSequenceIndex = sequenceIndex;
            if(newState === eMachineState.empty)
            {
                var options = {url: vm.url + "/api/shutdown", method: 'POST'};
                request(options, function( error, response, body)
                {
                    body = JSON.parse( body );
                    if(!error && response.statusCode === 200)
                    {
                        updateVMState(vm, body.machineState);
                        //does lspm still think it's okay to shut down?
                        if(body.machineState === eMachineState.shuttingDown)
                        {
                            shutdownVM(vm);
                        }
                    }
                } );
            }
            else
            {
                updateVMState(vm, newState);
            }
        }
    }
    else
    {
        res.sendStatus(404);
    }
} );

function shutdownVM(vm)
{
    updateVMState(vm, eMachineState.shuttingDown);
    askCloudProviderToKillVM(vm.cloudProviderId, function(err)
    {
        if(err)
        {
            console.log("Error closing vm " + vm.uuid);
            //we'll try again when heartbeat is missed
        }
        else
        {
```

```
                delete gVMs[vm.uuid]; //success...delete from everywhere
                delete gAvailableVMs[vm.uuid];
            }
        } );
}
function checkHeartbeats()
{
    var vmsToKill = [], vmUUID, vm, heartbeatAge;
    var time = getUTCSecondsSince1970();
    for(vmUUID in gVMs)
    {
        vm = gVMs[vmUUID];
        heartbeatAge = time - vm.lastHeartbeat;
        if(heartbeatAge > gMaxRunningHeartbeatAge &&
            vm.machineState !== eMachineState.pending)
        {
            vmsToKill.push(vm);
        }
    }
    vmsToKill.forEach(shutdownVM);
}
setInterval(checkHeartbeats, gHeartbeatCheckPeriodMS);
```

The heartbeat endpoint handler ignores heartbeats for VMs that are in the `pending` or `shuttingDown` states. Pending VMs change state as soon as their launch request is answered, so any other state change during that time needs to be handled after the launch completes. VMs in the `shuttingDown` state are shutting down already so do not require monitoring updates. The handler also ignores heartbeats with out-of-order sequence indices. If a heartbeat is worth considering, the handler updates the `lastSequenceIndex` and `lastHeartbeat` properties of the VM. Then, if the state is `empty`, indicating there are no game processes running on the VM, the handler begins the shutdown process by sending a shutdown request to the LSPM. The LSPM's shutdown handler checks its own `gMachineState` to make sure that it hasn't changed since the `empty` heartbeat went out. If it did not, it changes its own state to `shuttingDown` and responds to the VMM that it has accepted the request to shut down. The VMM then marks the VM as `shuttingDown` and asks the cloud provider to completely deprovision the VM.

The VMM `checkHeartbeats` function works like the LSPM function, but it ignores any timeouts for servers in the `pending` state. If a VM does time out, it means there is something wrong with the LSPM, so the VMM does not bother with the shutdown handshake. It instead immediately requests deprovisioning from the cloud service provider.

When the LSPM experiences a change in state due to a process shutting down, it does not need to wait for the predetermined heartbeat interval to notify the VMM. Instead, it can just send an extra heartbeat right away in response to the change. This is a simple way to give immediate feedback to the VMM and requires no extra functionality on the VMM's part.

This VMM implementation is functionally correct, prevents errors from race conditions, and is reasonably efficient. If many requests come in at once during the time it takes to provision a VM, though, it will end up provisioning one VM for each request. If the traffic is consistent this won't be a problem, but in the case of an anomalous spike, this may end up spawning a wasteful number of VMs. A better implementation could detect this situation and throttle the VM provisioning requests. Similarly, the VMM is possibly inefficiently aggressive in its shutting down of empty VMs. Depending on the rate at which games are requested and exited, it might be beneficial to keep empty VMs alive for a certain duration before deprovisioning them. A more robust VMM would have a tweakable threshold for this. Improvement of the VMM is left as an exercise.

> **tip**
>
> If a VMM needs to handle hundreds of requests per second, it may need a dynamic load balancer in front of it, and several Node instances to bear the brunt of the requests. In this case, the statuses of the VMs in the `gVMs` array need to be shared between instances, so instead of living in a single process' local memory, they should live in a rapid access shared data store such as **redis**. For more on redis, see the "Additional Readings" section. Alternatively, if requests are this frequent, it may be better to shard players geographically, with a statically dedicated VMM for each region.

Summary

With the increased prevalence of cloud service providers, every studio building a multiplayer game should consider hosting dedicated servers in the cloud. Even though it is easier than ever before, hosting dedicated servers still costs more than having the players host the servers, and increases complexity as well. It also introduces a dependency on third-party cloud service providers and removes feelings of ownership from your players. The advantages of hosting dedicated servers often outweigh the drawbacks though. Hosted servers provide reliability, availability, high bandwidth, cheat prevention, and unobtrusive copy protection.

Hosting dedicated servers requires building a few backend utilities. The tools of backend development differ significantly from those of client-side game development. REST APIs provide a text-based, discoverable, and easily debuggable interface between services. JSON provides a clean and compact format for data exchange. Node JS provides an optimized, event-loop driven, JavaScript engine for rapid development.

There are several moving parts in a dedicated server infrastructure. The server game instance represents an instance of the game shared between players. There may be one or more game instances in a game server process, which represents the game to the OS. One or more

game server processes may run on a game server machine. Typically game server machines are actually virtual machines, running with zero or more other virtual machines on the same physical machine.

To manage all of these parts, there is a local server process manager and a virtual machine manager. There is one LSPM per virtual machine, and it is responsible for spawning and monitoring processes on that machine, as well as reporting on its own health to the VMM. The VMM itself is the main entry point for process launch. When a matchmaking service decides that it needs a new game server launched, it sends a REST request to a VMM endpoint. The handler for that endpoint then either finds an underutilized VM or requests the cloud service provider provision a new one. With a VM identified, it requests the VM's LSPM launch the new game server process.

All these pieces work in concert to provide a robust, dedicated server environment, capable of supporting a vast and scalable number of players with no upfront hardware cost.

Review Questions

1. What are the advantages and disadvantages of hosting dedicated servers? Why was hosting dedicated servers much harder in the past?
2. What are the pros and cons of supporting multiple game instances per game server process?
3. What is a virtual machine? Why does cloud hosting typically involve virtual machines?
4. What main functions does a local server process manager provide?
5. List multiple ways a server game process can provide feedback to a local server process manager.
6. What is a virtual machine manager and what purpose does it serve?
7. Explain how the VMM might sometimes provision more VMs than it needs. Implement an improvement.
8. Explain how the VMM might sometimes deprovision VMs sooner than it should. Implement an improvement.

Additional Readings

C++ REST SDK—Home. Retrieved from https://casablanca.codeplex.com. Accessed September 12, 2015.

Caolan/async. Retrieved from https://github.com/caolan/async. Accessed September 12, 2015.

Docker—Build, Ship, and Run Any App, Anywhere. Retrieved from https://www.docker.com. Accessed September 12, 2015.

Express—Node.js web application framework. Retrieved from http://expressjs.com. Accessed September 12, 2015.

Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. (1999, June). *Hypertext Transfer Protocol—HTTP/1.1*. Retrieved from http://www.w3.org/Protocols/rfc2616/rfc2616.html. Accessed September 12, 2015.

Introducing JSON. Retrieved from http://json.org. Accessed September 12, 2015.

Node.js. Retrieved from https://nodejs.org. Accessed September 12, 2015.

Redis. Retrieved from http://redis.io/documentation. Accessed September 12, 2015.

Request/request. Retrieved from https://github.com/request/request. Accessed September 12, 2015.

Rest. Retrieved from http://www.w3.org/2001/sw/wiki/REST. Accessed September 12, 2015.

This page intentionally left blank

APPENDIX A

A MODERN C++ PRIMER

C++ is the video game industry standard programming language. While many game companies might use higher-level languages for gameplay logic, lower-level code such as networking logic is almost exclusively written in C++. The code throughout this book uses concepts relatively new to the C++ language, and this appendix covers these concepts.

C++11

Ratified in 2011, C++11 introduced many changes to C++ standard. Several major features were added in C++11, including both fundamental language constructs (such as lambda expressions) and new libraries (such as one for threading). Although a large number of concepts were added to C++11 this book only uses a handful of them. That being said, it is a worthwhile exercise to peruse additional references to get a sense of all of the additions that were made to the language. This section covers some general C++11 concepts that did not really fit in the other sections of this appendix.

One caveat is that since the C++11 standard is still relatively new, not all compilers are fully C++11-compliant. However, all the C++11 concepts used in this book work in three of the most popular compilers in use today: Microsoft Visual Studio, Clang, and GCC.

It should also be noted that there is a newer version of the C++ standard called C++14. However, C++14 is more of an incremental update, so there are not nearly as many language additions as in C++11. The next major revision to the standard is slated for release in 2017.

auto

While the `auto` keyword existed in previous versions of C++, in C++11 it takes on a new meaning. Specifically, this keyword is used in place of a type, and instructs the compiler to deduce the type at compile time. Since the type is deduced at compile time, this means that there is no runtime cost for using `auto`—but it certainly allows for more succinct code to be written.

For example, one headache in old C++ is declaring an iterator (if you are fuzzy on the concept iterators, you can read about them later in this appendix):

```
//Declare a vector of ints
std::vector<int> myVect;
//Declare an iterator referring to begin
std::vector<int>::iterator iter = myVect.begin();
```

However, in C++11 you can replace the complicated type for the iterator with `auto`:

```
//Declare a vector of ints
std::vector<int> myVect;
//Declare an iterator referring to begin (using auto)
auto iter = myVect.begin();
```

Since the return type of `myVect.begin()` is known at compile time, it is possible for the compiler to deduce the appropriate type for `iter`. The `auto` keyword can even be used for basic types such as integers or floats, but the value in these cases is rather questionable. One caveat to keep in mind is that `auto` does not default to a reference nor is it `const`—if these properties are desired, `auto&`, `const auto`, or even `const auto&` can be specified.

nullptr

Prior to C++11, the way a pointer was set to null was either with the number `0` or the macro `NULL` (which is just a `#define` for the number `0`). However, one major issue with this approach is that `0` is first and foremost treated as an integer. This can be a problem in the case of function overloading. For example, suppose the following two functions were defined:

```
void myFunc(int* ptr)
{
   //Do stuff
   //...
}
void myFunc(int a)
{
   //Do stuff
   //...
}
```

An issue comes up if `myFunc` is called with `NULL` passed as the parameter. Although one might expect that the first version would be called, this is not the case. That's because `NULL` is 0, and 0 is treated as an integer. If, on the other hand, `nullptr` is passed as the parameter, it will call the first version, as `nullptr` is treated as a pointer.

Although this example is a bit contrived, the point holds that `nullptr` is strongly typed as a pointer, whereas `NULL` or `0` is not. There's a further benefit that `nullptr` can be easily searched for in a file without any false positives, whereas 0 may appear in many cases where there is not a pointer in use.

References

A **reference** is a variable type that refers to another variable. This means that modifying the reference will modify the original variable. The most basic usage case of references is when writing a function that modifies function parameters. For example, the following function would swap the two parameters `a` and `b`:

```
void swap(int& a, int& b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

Thus if the `swap` function is called on two integer variables, upon completion of the function, these two variables would have their values swapped. This is because `a` and `b` are references to the original variables. Were the `swap` function written in C, we would have to use pointers instead of references. Internally, a reference is in fact implemented as a pointer—however, the semantics

of using a reference are simpler because dereferences are implicit. References are also generally safer to use as function parameters, because it can be assumed that a reference will never be null (though it is technically possible to write malformed code where a reference is null).

Const References

Modifying parameters is only the tip of the iceberg when it comes to references. For nonbasic types (such as classes and structs), passing by reference is almost always going to be more efficient than passing by value. This is because passing by value necessitates creating a copy of the variable—in the case of nonbasic type such as a vector or a string, creating the copy requires a dynamic allocation which adds a huge amount of overhead.

Of course, if the vector or string were just passed into a function by reference, this would mean that the function would be free to modify the original variable. What about the cases where this should be disallowed, such as when the variable is data encapsulated in a class? The solution to this is what's called a **const reference**. A const reference is still passed by reference, but it can only be accessed—no modification is allowed. This is the best of both worlds—a copy is avoided and the function can't modify the data. The following `print` function is one example of passing a parameter by const reference:

```
void print(const std::string& toPrint)
{
   std::cout << toPrint << std::endl;
}
```

In general, for nonbasic types it is a good idea to pass them into functions by const reference, unless the function intends to modify the original variable, in which case a normal reference should be used. However, for basic types (such as integers and floats) it generally is slower to use references as opposed to making a copy. Thus, for basic types it's preferred to pass by value, unless the function intends to modify the original variable, in which case a non-const reference should be used.

Const Member Functions

Member functions and parameters should follow the same rules as standalone functions. So nonbasic types should generally be passed by const reference and basic types should generally be passed by value. It gets a little bit trickier for the return type of so-called getter functions—functions that return encapsulated data. Generally, such functions should return const references to the member data—this is to prevent the caller from violating encapsulation and modifying the data.

However, once const references are being used with classes, it is very important that any member functions that do not modify member data are designated as const member functions. A **const member function** guarantees that the member function in question does not modify internal class data (and it is strictly enforced). This is important because given a const reference

to an object, only const member functions can be called on said object. If you attempt to call a non-const function on a const reference, it causes a compile error.

To designate a member function as const, the `const` keyword appears in the declaration, after the closing parenthesis for the function's parameters. The following `Student` class demonstrates proper usage of references as well as const member functions. Using const appropriately in this manner is often referred to as **const-correctness**.

```
class Student
private:
   std::string mName;
   int mAge;
public:
   Student(const std::string& name, int age)
      : mName(name)
      , mAge(age)
   { }

   const std::string& getName() const {return mName;}
   void setName(const std::string& name) {mName = name;}

   int getAge() const {return mAge;}
   void setAge(int age) {mAge = age;}
};
```

Templates

A **template** is a way to declare a function or class such that it can generically apply to any type. For example, this templated `max` function would support any type that supports the greater than operator:

```
template <typename T>
T max(const T& a, const T& b)
{
   return ((a > b) ? a : b);
}
```

When the compiler sees a call to `max`, it instantiates a version of the template for the type in question. So if there are two calls to `max`—one with integers and one with floats—the compiler would create two corresponding versions of `max`. This means that the executable size and execution performance will be identical to code where two versions of `max` were manually declared.

An approach similar to this can be applied to classes and/or structs, and it is used extensively in STL (covered later in this appendix). However, as with references there are quite a few additional possible uses of templates.

Template Specialization

Suppose there is a templated function called `copyToBuffer` that takes in two parameters: a pointer to the buffer to write to, and the (templated) variable that should be written. One way to write this function might be:

```
template <typename T>
void copyToBuffer(char* buffer, const T& value)
{
   std::memcpy(buffer, &value, sizeof(T));
}
```

However, there is a fundamental problem with this function. While it'll work perfectly fine for basic types, nonbasic types such as string will not function properly. This is because the function will perform a shallow copy as opposed to a deep copy of the underlying data. To solve this issue, a specialized version of `copyToBuffer` can be created that performs the deep copy for strings:

```
template <>
void copyToBuffer<std::string>(char* buffer, const std::string& value)
{
   std::memcpy(buffer, value.c_str(), value.length());
}
```

Then, when `copyToBuffer` is invoked in code, if the type of value is a `string` it will choose the specialized version. This sort of specialization can also be applied to a template that takes in multiple template parameters—in which case it is possible to specialize on any number of the template parameters.

Static Assertions and Type Traits

Runtime assertions are very useful for validation of values. In games, assertions are often preferred over exceptions both because there is less overhead and the assertions can be easily removed for an optimized release build.

A **static assertion** is a type of assertion that is performed at compile time. Since this assertion is during compilation, the Boolean expression to be validated must also be known at compilation. Here's a very simple example of a function which will not compile due to the static assertion:

```
void test()
{
   static_assert(false, "Doesn't compile!");
}
```

Of course, a static assert with a `false` condition doesn't really accomplish much other than halt compilation. An actual usage case is combining static assertions with the C++11 `type_traits`

header in order to disallow templated functions on certain types. Returning to the earlier `copyToBuffer` example, it would be preferable if the generic version of the function only worked on basic types. This could be accomplished with a static assertion, like so:

```
template <typename T>
void copyToBuffer(char* buffer, const T& value)
{
   static_assert(std::is_fundamental<T>::value,
      "copyToBuffer requires specialization for non-basic types.");
   std::memcpy(buffer, &value, sizeof(T));
}
```

The `is_fundamental` value will only be true in the case where `T` is a basic type. This means that any calls to the generic version of `copyToBuffer` will not compile if `T` is nonbasic. Where this gets interesting is when specializations are thrown into the mix—if the type in question has a template specialization associated with it, then the generic version is ignored, thus skipping the static assertion. This means that if the string version of `copyToBuffer` were still written as in that earlier example, calls to the function with a string as the second parameter would work just fine.

Smart Pointers

A **pointer** is a type of variable that stores a memory address, and is a fundamental construct used by C/C++ programmers. However, there are a few common issues that can crop up when using pointers incorrectly. One such issue is a memory leak—when memory is dynamically allocated on the heap, but never deleted. For example, the following class leaks memory:

```
class Texture
{
private:
   struct ImageData
   {
      //...
   };
   ImageData* mData;
public:
   Texture(const char* filename)
   {
      mData = new ImageData;
      //Load ImageData from the file
      //...
   }
};
```

Notice how there is memory dynamically allocated in the constructor of the class, but that memory is not deleted in the destructor. To fix this memory leak, we need to add a destructor that deletes `mData`. The corrected version of `Texture` follows:

```
class Texture
{
private:
   struct ImageData
   {
      //...
   };
   ImageData* mData;
public:
   Texture(const char* fileName)
   {
      mData = new ImageData;
      //Load ImageData from the file
      //...
   }
   ~Texture()
   {
      delete mData; //Fix memory leak
   }
};
```

A second, more insidious, issue can crop up when multiple objects have pointers to the same variable that was dynamically allocated. For example, suppose there is the following `Button` class (that uses the previously declared `Texture` class):

```
class Button
{
private:
   Texture* mTexture;
public:
   Button(Texture* texture)
      : mTexture(texture)
   {}
   ~Button()
   {
      delete mTexture;
   }
};
```

The idea is that each `Button` should display a `Texture`, and the `Texture` must have been dynamically allocated beforehand. However, what happens if two instances of `Button` are created, both pointing to the same `Texture`? As long as both buttons are active, everything will work fine. But once the first `Button` instance is destructed, the `Texture` will no longer be valid. But the second `Button` instance would still have a pointer to that newly deleted `Texture`, which in the best case causes some graphical corruption, and in the worst case causes the program to crash. This issue is not easily solvable with normal pointers.

Smart pointers are a way to solve both of these issues, and as of C++11 they are now part of the standard library (in the `memory` header file).

Shared Pointers

A **shared pointer** is a type of smart pointer that allows for multiple pointers to the same dynamically allocated variable. Behind the scenes, a shared pointer tracks the number of pointers to the underlying variable, which is a process called **reference counting**. The underlying variable is only deleted once the reference count hits zero. In this way, a shared pointer can ensure that a variable that's still being pointed at is not deleted prematurely.

To construct a shared pointer, it is preferred to use the `make_shared` templated function. Here's a simple example of using shared pointers:

```
{
   //Construct a shared pointer to an int
   //Initialize underlying variable to 50
   //Reference count is 1
   std::shared_ptr<int> p1 = std::make_shared<int>(50);
   {
      //Make a new shared pointer that's set to the
      //same underlying variable.
      //Reference count is now 2
      std::shared_ptr<int> p2 = p1;

      //Dereference a shared_ptr just like a regular one
      *p2 = 100;
      std::cout << *p2 << std::endl;
   } //p2 destructed, reference count now 1
} //p1 destructed, reference count 0, so underlying variable is deleted
```

Note that both the `shared_ptr` itself and the `make_shared` function are templated by the type of the underlying dynamically allocated variable. The `make_shared` function automatically performs the actual dynamic allocation—notice how there are no direct calls to either `new` or `delete` in this code. It is possible to directly pass a memory address into the constructor of a `shared_ptr`, but this approach is not recommended unless absolutely necessary, as it is less efficient and more error-prone than using `make_shared`.

If you want to pass a shared pointer as a parameter to a function, it should always be passed by value, as if it were a basic type. This is contrary to the usual rules of passing by reference, but it is the only way to ensure the reference count of the shared pointer is correct.

Putting this all together, it means that the `Button` class from earlier in this section could be rewritten to instead use a `shared_ptr` to a `Texture`, as shown in the following code. In this way, the underlying `Texture` data is guaranteed to never be deleted as long as there are active shared pointers to that `Texture`.

```
class Button
{
private:
   std::shared_ptr<Texture> mTexture;
public:
   Button(std::shared_ptr<Texture> texture)
      : mTexture(texture)
   {}
   //No destructor needed, b/c smart pointer!
};
```

There's another related feature of `shared_ptr` that bears mentioning. If a class needs to get a `shared_ptr` to itself, it should not manually construct a new `shared_ptr` from the `this` pointer, as this would not take into account any existing references. Instead, there is a template class you can inherit from called `enable_shared_from_this`. For example, if `Texture` needs to be able to get a `shared_ptr` to itself, it could inherit from `enable_shared_from_this` as follows:

```
class Texture: public std::enable_shared_from_this<Texture>
{
   //Implementation
   //...
};
```

Then, inside any of `Texture`'s member functions, you can call the `shared_from_this` member function, which will return a `shared_ptr` with the correct reference count.

There also are templated functions that can be used to cast between shared pointers to different classes in a hierarchy: `static_pointer_cast` and `dynamic_pointer_cast`.

Unique Pointer

A **unique pointer** is similar to a shared pointer, except it guarantees that only one pointer can ever point to the underlying variable. If you try to assign one unique pointer to another, it results in an error. This means that unique pointers don't need to track a reference count—they simply automatically delete the underlying variable when the unique pointer is destructed.

For unique pointers, use `unique_ptr` and `make_unique`–beyond the lack of reference counting, the code for using `unique_ptr` is very similar to code for using `shared_ptr`.

Weak Pointer

Behind the scenes, a `shared_ptr` actually has two types of reference counts: a strong reference count and a weak reference count. When the strong reference count hits zero, the underlying object is destroyed. However, the weak reference count has no bearing on whether or not the underlying object is destroyed. This leads to a weak pointer, which holds a weak

reference to the object controlled by a shared pointer. The basic idea of a weak pointer is it allows code that doesn't actually want to own an object to safely check whether or not said object still exists. The class used for this in C++11 is `weak_ptr`.

Suppose `sp` is already declared as a `shared_ptr<int>`. You could then create a `weak_ptr` directly from the `shared_ptr` as follows:

```
std::weak_ptr<int> wp = sp;
```

You can then use the expired function to test whether or not the weak pointer still exists. And if it's not expired, you can use lock to reacquire a `shared_ptr`, which will increase the strong reference count. This would look like:

```
if (!wp.expired())
{
   //This will increase the strong reference count
   std::shared_ptr<int> sp2 = wp.lock();
   //Now use sp2 like a shared_ptr
   //...
}
```

Weak pointers can also be used to avoid a circular reference. Specifically, if object A has a `shared_ptr` to object B, and object B has a `shared_ptr` to object A, there is no way object A or B can ever be deleted. However, if one of them has a `weak_ptr`, then the circular reference is avoided.

Caveats

There are a couple of things to watch out for with regards to smart pointers as implemented in C++11. First of all, they are difficult to use correctly with dynamically allocated arrays. If you want to use a smart pointer to an array, it is generally simpler to use an STL container array. It should also be noted that in comparison to normal pointers, smart pointers do come with a slight added memory overhead and performance cost. So for code that needs to be absolutely as fast as possible, it is not wise to use smart pointers. But for most typical usage cases, it's safer and easier (and thus, likely preferred) to use smart pointers.

STL Containers

The C++ **standard template library** (**STL**) contains a large number of container data structures. This section summarizes the most commonly used containers and their typical usage cases. Each container is declared in a header file corresponding to the container name, so it's not uncommon to need to include several headers to support several containers.

array

The `array` container (added in C++11) is essentially a wrapper for a constant size array. Because it is constant size, there are no `push_back` member functions or the like. Indices into the `array` can be accessed using the standard array subscript operator `[]`. Recall that the main advantage of arrays (in general) is that random access can be performed with an algorithmic complexity of *O*(1).

While C-style arrays serve the same purpose, one advantage of using the `array` container is that it supports iterator semantics. Furthermore, it is possible to employ bounds checking if the `at` member function is used instead of the array subscript operator.

vector

The `vector` container is a dynamically sized array. Elements can be added and removed from the back of a vector using `push_back` and `pop_back`, with *O*(1) algorithmic complexity. It is also possible to use `insert` and `remove` at any arbitrary location in the vector. However, these operations require copying some or all of the data in the array, which can make them computationally expensive. Resizing a vector is expensive for the same reason, in spite of its *O*(*n*) algorithmic complexity. This further means that even though `push_back` is considered *O*(1), calling it on a full vector will incur copying costs. As with `array`, bounds checking is performed if the `at` member function is used.

If you know how many elements you need to place in the vector, you can use the `reserve` member function to allocate space to fit that many elements. This will avoid any cost of growing and copying the vector as you add elements, and can save a tremendous amount of time.

For adding elements to a vector, C++11 provides a new member function `emplace_back`. The difference between `emplace_back` and `push_back` is apparent when you have a vector of a nonbasic type. Suppose you have a vector of a custom class `Student`. Suppose that the constructor of Student takes in the name of the student and their grade. If you were to use `push_back`, you might write code like this:

```
students.push_back(Student("John", 100));
```

This code first constructs a temporary instance of the Student class, and then makes a copy of this temporary instance in order to add it to the vector. However, `emplace_back` can construct the object in place, which avoids creating a temporary. You would call `emplace_back` as follows:

```
students.emplace_back("John", 100);
```

Notice how the call to `emplace_back` does not explicitly mention the `Student` type. This is called **perfect forwarding**, because the parameters are forwarded to the `Student` that is constructed in the vector.

There is no disadvantage of using `emplace_back` in lieu of `push_back`. All the other STL containers (other than array) support emplace functionality, as well, so you should get into the habit of using emplace functions to add elements to containers.

`list`

The `list` container is a doubly linked list. Elements can be added/removed from the front and back with guaranteed *O*(1) algorithmic complexity. Furthermore, given an iterator at an arbitrary location in the list, it is possible to `insert` and `remove` with *O*(1) complexity. Recall that lists do not support random access of specific elements. One advantage of a linked list is that it can never really be "full"—elements are added one at a time, so there is no need to worry about resizing a linked list. However, it should be noted that one disadvantage of a linked list is that because elements are not next to each other in memory, they are not as cache-friendly as an array. It turns out that cache performance is actually a significant bottleneck on modern computers. So as long as the size of each element is relatively small (64 bytes or less), a vector will almost always outperform a list.

`forward_list`

The `forward_list` container (added in C++11) is a singly linked list. This means that `forward_list` only supports *O*(1) addition and removal from the front of the list. The advantage of this is that it uses less memory per node in the list.

`map`

A `map` is an ordered container of {key, value} pairs, that are ordered by the key. Each key in the map must be unique and support **strict weak ordering**, meaning that if key A is less than B, then key B cannot be less than or equal to A. If you wish to use a custom type as a key, typically you override the less than operator. A map is implemented as a type of binary search tree, which means that lookup by key has an average algorithmic complexity of *O*(log(*n*)). Since it is ordered, iterating through the map is guaranteed to be sorted in ascending order.

`set`

A `set` is like map, except there is no pair—the key is simply also the value. All other behavior is identical.

`unordered_map`

The `unordered_map` container (added in C++11) is a hash table of {key, value} pairs. Each key must be unique. Since it is a hash table, lookup can be performed with an algorithmic complexity of *O*(1). However, it's unordered which means iterating through an `unordered_map` will not yield any meaningful order. Similarly, there is a hash set container called

`unordered_set`. For both `unordered_map` and `unordered_set`, hashing functions are provided for built-in types. If you wish to hash a custom type, you must provide your own specialization of the templated `std::hash` function.

Iterators

An **iterator** is a type of object whose intent is to allow for traversal through a container. All STL containers support iterators, and this section covers the common usage cases.

The following code snippet constructs a vector, adds the first five Fibonacci numbers to the vector, and then uses iterators to print out each element in the vector:

```
std::vector<int> myVec;
myVec.emplace_back(1);
myVec.emplace_back(1);
myVec.emplace_back(2);
myVec.emplace_back(3 );
myVec.emplace_back(5);

//Iterate through vector, and output each element
for(auto iter = myVec.begin();
    iter != myVec.end();
    ++iter)
{
   std::cout << *iter << std::endl;
}
```

To grab an iterator to the first element in an STL container, the `begin` member function is used, and likewise the `end` member function grabs an iterator to one past the last element. Notice that the code used `auto` to declare the type of the iterator, in order to avoid needing to spell out the full type (which in this case is `std::vector<int>::iterator`).

Also notice that the iterator is incremented to the next element by using the prefix `++` operator—for performance reasons, the prefix operator should be used in lieu of the postfix operator. Finally, iterators are dereferenced like pointers are dereferenced—this is how the underlying data at the element is accessed. This can be tricky if the underlying element is a pointer, because there are two dereferences: first of the iterator and then of the pointer itself.

All STL containers also support two kinds of iterators: the normal iterator as shown earlier, and a `const_iterator`. The difference is that a `const_iterator` does not allow modification of the data in the container, whereas a normal iterator does. This means that if code has a const reference to an STL container, it is only allowed to use a `const_iterator`.

Range-Based For Loop

In the case where it is simply desired to loop through an entire container, it is simpler to use a new C++11 addition called the **range-based for loop**. The loop just mentioned could instead be rewritten as follows:

```
//Iterate using a range-based for
for (auto i : myVec)
{
   std::cout << i << std::endl;
}
```

A range-based for loop looks much like what a `foreach` might look like in other languages such as Java or C#. This code grabs each element in the container, and saves it into the temporary variable `i`. The loop ends only once all elements have been visited. In a range-based for, it is possible to grab each element by value or by reference. This means that if it is desired to modify elements in the container, references should be used, and furthermore for nonbasic types either references or const references should always be used.

Internally, a range-based for will work on any container that supports STL-style iterator semantics (e.g., there is an `iterator` member, a `begin`, an `end`, the iterator can be incremented, dereferenced, and so on). This means that it is possible to create a custom container that supports the range-based for loop.

Other Uses of Iterators

There are a multitude of functions in the `algorithm` header that use iterators in one way or another. However, one other common use of iterators is the `find` member function that `map`, `set`, and `unordered_map` support. The `find` member function searches through the container for the specified key, and returns an iterator to the corresponding element in the container. If the key is not found, `find` will instead return an iterator equal to the `end` iterator.

Additional Readings

Meyers, Scott. (2014, December). *Effective Modern C++*. O'Reilly Media.

Stroustrup, Bjarne. (2013, May). *The C++ Programming Language, 4th ed*. Addison-Wesley.

This page intentionally left blank

INDEX

Page numbers followed by "*f*" and "*t*" indicate figures and tables, respectively.

A

B

C

D

E

F

G

H

I

J

N

O

P

Q

R

S

T

U

V

W

X

Z

JOIN THE INFORMIT AFFILIATE TEAM!

You love our titles and you love to share them with your colleagues and friends...why not earn some $$ doing it!

If you have a website, blog, or even a Facebook page, you can start earning money by putting InformIT links on your page.

Whenever a visitor clicks on these links and makes a purchase on informit.com, you earn commissions* on all sales!

Every sale you bring to our site will earn you a commission. All you have to do is post the links to the titles you want, as many as you want, and we'll take care of the rest.

APPLY AND GET STARTED!

It's quick and easy to apply.
To learn more go to:
http://www.informit.com/affiliates/

*Valid for all books, eBooks and video sales at www.informit.com

Addison Wesley PRENTICE HALL SAMS

informIT