

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**

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

# **Multiplayer Game Programming**

# The Addison-Wesley Game Design and Development Series



Visit [informit.com/series/gamedesign](http://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](http://informit.com/socialconnect)

**informit.com**  
the trusted technology learning source

 Addison-Wesley

**Safari**  
Books Online

# 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



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the U.S., please contact [international@pearsoned.com](mailto:international@pearsoned.com).

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

Library of Congress Control Number: 2015950053

Copyright © 2016 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, 200 Old Tappan Road, Old Tappan, New Jersey 07675, or you may fax your request to (201) 236-3290.

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

**Composer**

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

<b>1</b>	Overview of Networked Games . . . . .	1
	A Brief History of Multiplayer Games . . . . .	2
	Starsiege: Tribes . . . . .	5
	Age of Empires . . . . .	10
	Summary . . . . .	13
	Review Questions . . . . .	14
	Additional Readings . . . . .	14
<b>2</b>	The Internet . . . . .	15
	Origins: Packet Switching . . . . .	16
	The TCP/IP Layer Cake . . . . .	17
	The Physical Layer . . . . .	19
	The Link Layer . . . . .	19
	The Network Layer . . . . .	23
	The Transport Layer . . . . .	39
	The Application Layer . . . . .	52
	NAT . . . . .	53
	Summary . . . . .	60
	Review Questions . . . . .	61
	Additional Readings . . . . .	62
<b>3</b>	Berkeley Sockets . . . . .	65
	Creating Sockets . . . . .	66
	API Operating System Differences . . . . .	68
	Socket Address . . . . .	71
	UDP Sockets . . . . .	79
	TCP Sockets . . . . .	83
	Blocking and Non-Blocking I/O . . . . .	88
	Additional Socket Options . . . . .	96

Summary . . . . .	98
Review Questions . . . . .	98
Additional Readings . . . . .	99
<b>4 Object Serialization . . . . .</b>	<b>101</b>
The Need for Serialization . . . . .	102
Streams . . . . .	105
Referenced Data . . . . .	119
Compression . . . . .	124
Maintainability . . . . .	130
Summary . . . . .	136
Review Questions . . . . .	136
Additional Readings . . . . .	137
<b>5 Object Replication . . . . .</b>	<b>139</b>
The State of the World . . . . .	140
Replicating an Object . . . . .	140
Naïve World State Replication . . . . .	148
Changes in World State . . . . .	152
RPCs as Serialized Objects . . . . .	159
Custom Solutions . . . . .	162
Summary . . . . .	163
Review Questions. . . . .	163
Additional Readings . . . . .	164
<b>6 Network Topologies and Sample Games . . . . .</b>	<b>165</b>
Network Topologies . . . . .	166
Implementing Client-Server . . . . .	170
Implementing Peer-to-Peer . . . . .	182
Summary . . . . .	196
Review Questions. . . . .	197
Additional Reading. . . . .	197

<b>7</b>	Latency, Jitter, and Reliability . . . . .	199
	Latency . . . . .	200
	Jitter . . . . .	204
	Packet Loss . . . . .	206
	Reliability: TCP or UDP? . . . . .	207
	Packet Delivery Notification . . . . .	209
	Object Replication Reliability . . . . .	221
	Simulating Real-World Conditions . . . . .	228
	Summary . . . . .	230
	Review Questions . . . . .	231
	Additional Readings . . . . .	232
<b>8</b>	Improved Latency Handling . . . . .	233
	The Dumb Terminal Client . . . . .	234
	Client Side Interpolation . . . . .	236
	Client Side Prediction . . . . .	238
	Server Side Rewind. . . . .	248
	Summary . . . . .	249
	Review Questions . . . . .	250
	Additional Readings . . . . .	251
<b>9</b>	Scalability. . . . .	253
	Object Scope and Relevancy . . . . .	254
	Server Partitioning . . . . .	260
	Instancing . . . . .	262
	Prioritization and Frequency . . . . .	263
	Summary . . . . .	263
	Review Questions . . . . .	264
	Additional Readings . . . . .	264

<b>10</b>	Security . . . . .	265
	Packet Sniffing . . . . .	266
	Input Validation . . . . .	270
	Software Cheat Detection . . . . .	271
	Securing the Server . . . . .	274
	Summary . . . . .	277
	Review Questions . . . . .	278
	Additional Readings . . . . .	278
<b>11</b>	Real-World Engines . . . . .	279
	Unreal Engine 4 . . . . .	280
	Unity . . . . .	284
	Summary . . . . .	287
	Review Questions . . . . .	288
	Additional Readings . . . . .	288
<b>12</b>	Gamer Services . . . . .	289
	Choosing a Gamer Service . . . . .	290
	Basic Setup . . . . .	290
	Lobbies and Matchmaking . . . . .	294
	Networking . . . . .	298
	Player Statistics . . . . .	300
	Player Achievements . . . . .	305
	Leaderboards . . . . .	307
	Other Services . . . . .	308
	Summary . . . . .	309
	Review Questions . . . . .	310
	Additional Readings . . . . .	310

<b>13</b>	Cloud Hosting Dedicated Servers . . . . .	311
	To Host or Not To Host . . . . .	312
	Tools of the Trade . . . . .	313
	Overview and Terminology . . . . .	315
	Local Server Process Manager. . . . .	318
	Virtual Machine Manager. . . . .	324
	Summary . . . . .	333
	Review Questions . . . . .	334
	Additional Readings . . . . .	334
<b>Appendix A</b>	A Modern C++ Primer . . . . .	337
	C++11 . . . . .	338
	References . . . . .	339
	Templates . . . . .	341
	Smart Pointers . . . . .	343
	STL Containers . . . . .	347
	Iterators . . . . .	350
	Additional Readings . . . . .	351
	Index . . . . .	353



*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 Cengage® 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.

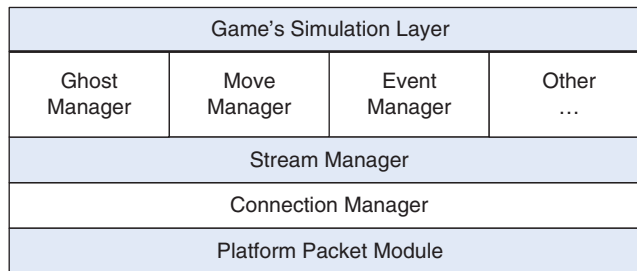
However, using an unreliable protocol can be problematic when a game needs to send information that is important to all the players in the game. Thus, the engineers needed to consider the different types data they wanted to send out. The developers of *Tribes* ultimately separated their data requirements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

Frohnmayr, 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.

*This page intentionally left blank*

# INDEX

Page numbers followed by "f" and "t" indicate figures and tables, respectively.

## A

AchieveData, 305  
Achieve.def, 305  
ACK flag, 46  
acknowledgment. *See also* packet delivery  
    notification  
    delivery status and, 216–221  
    pending, 213–216  
    processing, 216–218  
acknowledgment number (32-bits), 43  
ACK packet, 51, 52  
AckRange, 213–215, 216–218  
Actor class, 281  
actor replication  
    defined, 282  
    Unreal Engine 4, 282–283  
AddPendingAck(), 213  
address, bind function, 78  
address\_len, bind function, 79  
address resolution protocol (ARP), 26–28  
    hardware address length (8 bits), 28  
    hardware type (16 bits), 27  
    operation (16 bits), 28  
    packet structure, 27–28, 27f  
    protocol address length (8 bits), 28  
    protocol type (16 bits), 27  
    sender hardware address (variable length), 28  
    sender protocol address (variable length), 28  
    table, 27t  
    target hardware address (variable length), 28  
    target protocol address (variable length), 28  
AddToStat, 304–305  
AF\_INET, 66, 66t  
AF\_INET6, 66, 66t  
AF\_IPX, 66t  
af parameter, 66

AF\_UNSPEC, 66t  
*Age of Empires*, 10–13  
    deterministic lockstep model, 10  
    synchronization, 12–13  
    turn timer, 11–12  
AIController class, 281  
API, socket creation, 66  
app ID, 290  
application layer, 52–53  
    DHCP, 52  
    DNS, 52–53  
ARPANET, 16  
*Asheron's Call*, 256  
askCloudProviderForVM function, 329  
assertions  
    runtime, 342  
    static, 342–343  
asynchronous, 5  
async.series function, 329  
authoritative server, 167  
authority, 282–283  
autonomous proxy, 282

## B

backend server development, 313  
bad data, 274–275  
ban wave, 273  
*Battlefield*, 167  
BBN Report 1822, 17  
BBS. *See* bulletin board system (BBS)  
Berkeley Sockets API. *See* socket  
Bettner, Paul, 12  
bind function, 78–79  
binding address to socket, 78–7912  
bit streams, 114–119  
    input memory, 119

- bit streams (*continued*)
    - memory, 114
    - output memory, 114–119
    - serialization of field's value, 149–150
  - Bizzard Entertainment, 273
  - Blueprint, 283
  - Bluetooth, 21
  - bot, 272
  - `bReplicateMovement` flag, 283
  - broadcast address
    - MAC address, 30
    - subnet mask, 30
  - `buf`, receiving data
    - TCP socket, 86
    - UDP socket, 80
  - `buf`, sending data
    - TCP socket, 85
    - UDP socket, 79–80
  - bulletin board system (BBS), 3
  - BYTE Magazine*, 2
  - bytes, 43
  - `ByteSwap` function, 113
  - `ByteSwapper`, 113
  - byte swapping functions, 111–113
- C**
- C#, built-in reflection systems, 133
  - C++, 337
    - `offsetof` macro, 135
    - reflection systems, 133
  - C++11, 338–339
    - auto, 338
    - `nullptr`, 339
  - `callback` function, 329
  - cheat prevention, cloud hosting server, 313
  - `checkHeartbeat` function, 323, 332
  - checksum (16 bits), 194
    - IPv4 packet header, 25
    - UDP header, 42
  - `CheckSync` function, 195
  - CIDR. *See* classless inter-domain routing (CIDR)
  - circuit switching, 16, 16f
  - class identifier
    - object creation registry, 144–148
    - replication, 142–144
  - classless inter-domain routing (CIDR), 31
  - client code for move lists, 179–180
  - client function, 283–284
  - client proxy, 177
  - `ClientProxy` class, 180–181
  - client RPC function, 286
  - client-server topology, 7, 166–168, 166f
    - authoritative server, 167
    - dedicated server, 167
    - host migration, 168
    - implementing, 170–182
    - listen server, 168
    - Unity, 285
    - Unreal, 281–282
  - client side interpolation, 236–237
    - interpolation period, 237
    - packet period, 237
    - timing, 236f
  - client side prediction, 238–248
    - dead reckoning, 240–242, 241f
    - optimistic algorithm, 241
  - `closesocket` function, 67
  - cloud hosting dedicated server
    - benefits, 313
    - drawbacks, 312
    - game server machine, 317
    - game server process, 316–317
    - hardware, 317
    - JSON, 314
    - LSPM, 318–324
    - NodeJS, 314–315
    - overview, 311
    - REST, 313–314
    - server game instance, 316
    - terminology, 315–317
    - tools, 313–315
    - VMM, 324–333
  - `cloudProviderId`, 327
  - command, in Unity, 286
  - `Command` class, 186–187
  - `CommandList`, 188
  - communications protocol, 6
  - complexity, cloud hosting server, 312
  - `Component` classes, 285
  - compression, 123–130
    - entropy encoding, 125–127
    - fixed point, 127–129
    - geometry, 129–130
    - sparse array, 124–125
  - `ComputeGlobalCRC`, 194–195



congestion control, 50–51  
 connection manager, 8  
 conservative algorithm, 235  
 const-correctness, 341  
 const member function, 340–341  
 const reference, 340  
 control bits (9 bits), 43  
 Controller class, 281  
 cost, cloud hosting server, 312  
*Counter-Strike*, 248  
 CRC. *See* cyclic redundancy check (CRC)  
 Create function, 280  
 CreateGameObjectFromStream  
   functions, 144  
 CreateTCPsocket function, 88  
 C++ REST SDK, 324  
 cryptography, 267–269  
 CSteamID class, 293  
 cyclic redundancy check (CRC), 23, 194–195

## D

daisy chain, 2  
 data driven serialization, 133–135  
 data offset (4 bits), 43  
 data transmission, TCP, 46–51  
   congestion control, 50–51  
   delayed acknowledgment, 50  
   flow control, 49–50, 49f  
   with no packet loss, 47f  
   in order, 48  
   packet lost and retransmitted, 47f  
 DataType class, 134  
 data type registry, 148  
 DDoS. *See* distributed denial-of-service attack (DDoS)  
 dead reckoning, 240–242, 241f. *See also* client side prediction  
 dedicated server, 167  
   Unity, 285  
   Unreal, 281–282  
 default address, 34  
 delayed acknowledgment, 50  
 DeliveryNotificationManager,  
   216–221, 227–228  
 delivery status, receiving acknowledgment  
   and, 216–221  
 delivery status notification, 8

destination address (32 bits), 26  
 destination port (16 bits)  
   TCP header, 42  
   UDP header, 41  
 deterministic lockstep model, 10  
 DHCP. *See* Dynamic host configuration protocol (DHCP)  
 DHCPDISCOVER message, 52  
 DHCPOFFER packet, 52  
 digital rights management (DRM), 313  
 display lag, 202  
 distributed denial-of-service attack (DDoS), 274  
 DNS. *See* Domain name system (DNS)  
 Docker Container format, 325  
 DoClientSidePredictionAfter  
   ReplicationForLocalCat, 247  
 DoClientSidePredictionAfter  
   ReplicationForRemoteCat, 247  
 Domain name system (DNS), 52–53  
 do not fragment flag, 36  
 DownloadLeaderboardEntries  
   function, 308  
 DRM. *See* digital rights management (DRM)  
 dumb terminal client, 234–236  
 Dynamic host configuration protocol (DHCP), 52  
 dynamic ports, 40

## E

eMachineState object, 321  
 embedding. *See* inlining/embedding  
*Empire*, 2  
 endianness, 110–113  
   big-endian, 110  
   byte swapping functions, 111–113  
   little-endian, 110  
 Engine::DoFrame, 293  
 Engine::StaticInit, 292  
 EnterLobby function, 295  
 entropy, 125, 192  
 entropy encoding, 125–127. *See also* compression  
 EPrimitiveType, 134  
 errno, 70  
 Ethernet, 21–23  
   FCS, 23  
   hubs, 23  
   MAC address, 21–22  
   NIC, 21, 22

Ethernet (*continued*)

OUI, 21

switches, 23

EtherType, 22, 25

event manager, 8–9

Express JS, 315

ExtendIfShould(), 215

## F

FCS. *See* frame check sequence (FCS)

file output stream, 105

FIN packet, 51–52

FIN packet, 67–68

fixed point compression, 127–129

flagging functions, in Unity, 286

flags, receiving data

TCP socket, 86

UDP socket, 80

flags, sending data

TCP socket, 85

UDP socket, 80

flow control, TCP, 49–50, 49f

fragmentation, IPv4, 35–38

concept, 35

do not fragment flag, 36

fragment flags (3 bits), 25, 36

fragment identification (16 bits), 25, 35

fragment offset (13 bits), 25, 35

relevant header fields, 36, 36f

fragment flags (3 bits), 25, 36

fragment identification (16 bits), 25, 35

fragment offset (13 bits), 25, 35

frame, 19

delivery of, 20

jumbo, 22

frame check sequence (FCS), 23

frequency, 263

from, receiving data

UDP socket, 81

fuzz testing, 274–275

## G

game instance. *See* server game instance

GameObject class, 285

gamer service

basic setup, 290–294

choosing, 290

leaderboards, 307–308

lobbies and matchmaking, 294–298

networking, 298–300

other options, 308–309

overview, 289

player achievements, 305–306

player statistics, 300–305

GamerServices class, 291, 298

GamerServices.h, 291

GamerServices::Impl, 295, 303

GamerServices object, 291, 303

GamerServiceSocket class, 292

GamerServices::StaticInit, 292

GamerServicesSteam.cpp, 291

GamerServices::Update, 293

game server machine, 317

game server process, 316–317

gAvailableVMs map, 326, 329, 330

geometry compression, 129–130

GetDataType virtual function, 148

GetDesiredHorizontalDelta  
function, 178

GetDesiredVerticalDelta function, 178

getFirstAvailableVM function, 327

GetLobbyPlayerMap function, 297

GetLocalPlayerId function, 293

GetOffsetof method, 135

GetPrimitiveType method, 135

GetSize, 74

GetStatInt, 304–305

GetTimeDispatched(), 219

gHeartbeatCheckPeriod, 323

ghost manager, 9

gMachineState, 321, 332

gMaxProcessCount, 321

gMaxRunningHeartbeat, 323

Google on IPv6, 38

gProcessCount, 321

gProcesses, 321

gProcesses map, 321

guaranteed data, 6

guaranteed quickest data, 6

## H

HandleDeliveryFailure(), 226

hardware address length (8 bits), 28

hardware type (16 bits), 27  
 hashing algorithm for passwords, 277  
 hash maps, VMM, 326  
 header checksum (16 bits), 25  
 header length (4 bits), 25  
 heartbeat monitoring system, 330–332  
*Hearthstone: Heroes of Warcraft*, 5  
 helper functions, 178  
 host migration, 168  
*how*, 67–68  
 HTTP, 313–314  
 hubs, 23

**I**

IANA. *See* Internet Assigned Numbers Authority (IANA)  
 ICANN. *See* Internet Corporation for Assigned Names and Numbers (ICANN)  
 IGDP. *See* Internet gateway device protocol (IGDP)  
 indirect routing, IPv4. *See* subnet mask  
*InFlightPackets*, 217–218, 228  
 in-flight packets, optimization from, 226–228  
 information cheat, 269  
*inGameObject*, 142  
*InitDataType* function, 134  
 inlining/embedding, 120–121  
 input lag, 12  
*InputManager* class, 177, 178  
*InputMemoryStream* class, 131  
 input memory streams, 108–109  
 input sampling latency, 200  
 input sharing model, 169  
*InputState* class, 177–178  
 input stream, 105  
 input validation, 270–271  
 instancing, 262  
 interface identifier, 39. *See also* IPv6  
 Internet. *See* IP address; TCP/IP suit  
 Internet Assigned Numbers Authority (IANA), 40  
 Internet Corporation for Assigned Names and Numbers (ICANN), 40, 53  
 Internet gateway device protocol (IGDP), 60  
 Internet protocol version 4. *See* IPv4  
 Internet Service Provider (ISP), 32–33  
*InterpolateClientSidePrediction()*, 247

interpolation period, 237. *See also* client side interpolation  
 intrusions, 276–277  
 IP address  
   DHCP server, 52  
   DNS and, 52–53  
   ICANN distribution, 53  
   loopback, 35  
   name server and, 53  
   ports and, 41  
   privately routable, 53–54, 54t  
   as publically routable, 53  
   subnet mask and, 30, 30t  
   zero network broadcast address, 35  
*IPPROTO\_TCP* options, 97t  
 IPv4, 24–38  
   ARP, 26–28  
   concept, 24  
   fragmentation, 35–38  
   IP address, 24  
   IPv6 vs., 39  
   packet, 24–26  
   prefix, 39  
   subnet and indirect routing, 29–35. *See also* subnet mask  
 IPv6, 38–39  
   address forms, 39t  
   final 64 bits of, 39  
   first 64 bits of, 39  
   Google on, 38  
   interface identifier, 39  
   IPv4 vs., 39  
*IsInput* method, 131  
*ISocketSubsystem* class, 280  
 iterators, 350–351

## J

Java, 314  
   built-in reflection systems, 133  
 JavaScript object notation (JSON), 314  
 jitter, 204–205  
   defined, 204  
   processing delay, 205  
   propagation delay, 205  
   queuing delay, 205  
   simulating, 228–230  
   transmission delay, 205

JSON. *See* JavaScript object notation (JSON)

jumbo frames, 22

## L

lastHeartbeat, 321, 326, 332

lastSequenceIndex, 326, 332

latency, 200–204

defined, 200

display lag, 202

dumb terminal client, 234–236

input sampling, 200

multithreaded render pipeline, 201

network, 202–204

non-network, 200–202

pixel response time, 202

render pipeline, 200–201

simulating, 228–230

VSync, 201

leaderboards, 307–308. *See also* gamer service

Leaderboards.def, 307

League of Legends, 316

len, sending data

TCP socket, 85

UDP socket, 80

length (16 bits)

IPv4 packet, 25

UDP header, 41

linking, 121–123

LinkingContext class, 122–123

link layer, 19–23

concept, 19

duties of, 19

Ethernet, 21–23

physical medium and, 20, 20*t*

shortcomings, 23–24

listen server, 168

Unity, 285

Unreal, 282

lobbies, gamer service, 294–298

LobbyChatMsg\_t callback, 297

LobbySearchAsync function, 294, 295

local area network (LAN), 3, 54

localhost address. *See* loopback

local multiplayer games, 2

local perception filter, 236

local server process manager (LSPM), 318–324

initialization, 319–321

kill routes, 319–321

launch, 319–321

process monitoring, 322–324

sending heartbeat to, 323–324

loopback, 35

lpWSAData, 70

LSPM. *See* local server process manager (LSPM)

## M

MAC (media access control) address, 21–22

mAchieveArray, 305

machine images, 317

machineState, 326

man-in-the-middle attack, 266–269, 266*f*

concept, 266

public key cryptography, 267–268, 268*f*

map hacking, 272

map hacks, 13

Massively Multiplayer Online Game (MMO), 4–5

master peer, 169, 183

matchmaking, 169

gamer service, 294–298

Unity, 285–286

maximum segment size (MSS), 48

maximum transmission unit (MTU), 22

Maze War, 2

media access control (MAC). *See* MAC (media access control) address

members, VM object, 326–327

MemberVariable class, 134, 135

MemoryStream, 131–132, 141

memory streams, 106–110

mMemberVariables, 134

MMOFPS, 4

MMORPG, 4–5

mNetworkReplicationCommand, 222

mobile networked games, 5

MonoBehaviour, 285, 287

more fragments flag, 36

most recent state data, 6

MouseStatus function, 134

Move class, 178–179

MoveList class, 179, 188

move manager, 10

MSS. *See* maximum segment size (MSS)

MTU. *See* maximum transmission unit (MTU)

MUD. *See* multi-user dungeon (MUD)

multicast function, 284  
 multiplayer games  
     brief history of, 2–5  
     early networked, 2–3  
     local area network, 3  
     local multiplayer, 2  
     MMO, 4–5  
     mobile networked games, 5  
     multi-user dungeon, 3  
     online games, 4  
 multithreaded render pipeline latency, 201  
 multi-user dungeon (MUD), 3

**N**

Nagle's algorithm, 51  
 name server, 53  
 NAT. *See* network address translation (NAT)  
 NAT table, 56  
     original destination IP address and port to, 57  
     STUN, 58*f*  
 NDP. *See* neighbor discovery protocol (NDP)  
 neighbor discovery protocol (NDP), 39  
 network address, 30  
 network address translation (NAT), 53–60  
     concept, 53  
     functioning, 54–56  
     privately routable IP address, 53–54, 54*t*  
     STUN, 57–59  
     traversal, 57–59  
 NetworkBehaviour, 286  
 networked multiplayer games. *See* multiplayer games  
 NetworkEventType, 285  
 network interface controller (NIC), 21, 22  
 network latency  
     processing delay, 202–203  
     propagation delay, 203, 204  
     queuing delay, 203  
     transmission delay, 203  
 network layer, 23–39  
     duty, 24  
     IPv4, 24–38  
     IPv6, 38–39  
 NetworkManager class, 185  
     Unity, 285  
     Unreal, 280  
 NetworkManagerClient, 179

NetworkManager::EnterLobby function, 296  
 NetworkMatch class, 287  
 NetworkServer, 285  
 network stream, 105  
 network topologies  
     client-server, 166–168, 166*f*  
     concept, 166  
     peer-to-peer, 168–169, 168*f*  
     Unity game engine, 285  
     Unreal Engine 4, 281–282  
 NetworkTransport.Connect function, 285  
 NIC. *See* network interface controller (NIC)  
 NodeJS, 314–315  
 Node package manager (npm), 314  
 nodes, 17  
 non-guaranteed data, 6  
 non-network latency, 200–202  
 nullptr, 339

## O

object  
     identifying serialized object, 141–142  
     multiple, per packet, 148  
     replication. *See* replication  
     serialization. *See* serialization  
 object creation registry, 144–148  
 ObjectCreationRegistry, 163  
 object relevancy, 254  
 ObjectReplicationHeader, 162  
 object state delta, 152–153  
 octets, 43  
 offsetof macro, 135  
 OnDeserialize, 286  
 online game, 4  
 OnLobbyChatUpdate, 297  
 OnLobbyCreateCallback, 295  
 OnLobbyEnteredCallback, 295  
 OnLobbyMatchListCallback functions, 295  
 OnSerialize, 286  
 OnStatsReceived, 303  
 Open Systems Interconnection (OSI) model, 18  
 operating system differences, for sockets, 68–71  
 operation (16 bits), 28  
 ## operator, 302  
 optimistic algorithm, 241  
 optimization from in-flight packets, 226–228  
 organizationally unique identifier (OUI), 21

- OUI. *See* organizationally unique identifier (OUI)
  - OutputMemoryBiyStream class
    - declaration, 114
    - WriteBits methods, 114–116
  - OutputMemoryStream class, 131
  - output memory streams, 106–108
  - output stream, 105
- P**
- packet, IPv4, 24–26
  - packet delivery notification, 209–221
    - acknowledgments and delivery status, 216–221
    - pending acknowledgment, 213–216
    - processing incoming sequence number, 211–213
    - tagging outgoing packets, 210–211
  - packet length (16 bits), IPv4 header, 25
  - packet loss, 206–207
    - simulating, 228–230
  - packet period, 237
  - packets, 7, 17
  - packet sniffing
    - concept, 266
    - host machine, 269–270
    - man-in-the-middle attack, 266–269
  - packet switching, 16–17, 17*f*
  - PacketType enum, 140–141, 162
  - partial object state replication, 156–159
  - passwords, 276–277
  - peer-to-peer topology, 7, 11, 168–169, 168*f*
    - connecting new players in, 169
    - implementing, 182–196
    - input sharing model, 169
    - synchronization, 191–196
  - peer-to-peer validation system, 270, 271
  - pending acknowledgment
    - adding, 213–215
    - writing, 215–216
  - perfect forwarding, 348
  - physical layer, 19
  - pixel response time, 202
  - platform packet module, 7
  - PLATO system, 2
  - player, gamer service and
    - achievements, 305–306
    - IDs and name, 293–294
    - statistics, 300–305
  - PlayerCat component, 285
  - PlayerController, 281, 283
  - player IDs and name, 293–294
  - pointers, 343–345
    - shared, 345–346
    - unique, 346
    - weak, 346–347
  - pointer to implementation, 291
  - port(s)
    - bind, 40
    - concept, 40
    - dynamic (49152 to 65535), 40
    - IP addresses and, 41
    - system (0 to 1023), 40
    - user (1024–49151), 40
  - port assignment prediction, 60
  - port number registry, 40
  - POSIX-compatible operating systems, sockets
    - on, 68–69, 70–71
  - potentially visible set (PVS), 258–259
  - preamble, 22
  - prefab in Unity, 285–286
  - prefix, IPv6, 39
  - prioritization, 263
  - privately routable IP address, 53–54, 54*t*
  - PRNG. *See* pseudo-random number generator (PRNG)
  - ProcessAcks(), 216–217
  - ProcessCommand, 188
  - ProcessCommand, 187, 188
  - processing delay
    - jitter, 205
    - network latency, 202–203
  - ProcessReplicationAction, 160
  - ProcessSequenceNumber(), 211–213
  - ProcessTimedOutPackets(), 218
  - propagation delay
    - jitter, 205
    - network latency, 203, 204
  - protocol (8 bits), 25
  - protocol address length (8 bits), 28
  - protocol type (16 bits), 27
  - pseudo-random number generator (PRNG), 13
    - synchronizing, 191–194
  - PT\_ReplicationData, 141, 148
  - publically routable IP address, 53
  - public key cryptography, 267–269, 268*f*
  - pure servers, 273
  - PVS. *See* potentially visible set (PVS)

## Q

QoSType enum, 284  
*Quake*, 234–235  
 quaternion, 129  
 queuing delay  
   jitter, 205  
   network latency, 203

## R

random\_device class, 193  
 range-based for loop, 351  
 Read function, 187  
 ReadLastMoveProcessedOnServer  
   Timestamp, 245  
 real-time strategy game. *See Age of Empires*  
 reasonable copy protection, 313  
 receive window (16 bits), 43  
 receiving data  
   TCP socket, 86  
   UDP socket, 80–81  
 recentLaunchUnknown, 330  
 recv, TCP socket, 86  
 recvfrom function, UDP socket, 80–81  
 redis, 333  
 reference, 339–341  
   const, 340  
   const member function, 340–341  
 referenced data  
   inlining/embedding, 120–121  
   linking, 121–123  
 reflection systems, 133–134  
 registered ports. *See* user ports  
 reliability  
   object replication, 221–228  
   TCP, 207–208, 209t  
   UDP, 208–209, 209t  
 Reliable, 285  
 reliable data transfer, 43–44, 44f  
 ReliableFragmented, 285  
 reliance on third party, 312  
 Remote Method Invocation (RMI), 162  
 remote players, dead reckoning for, 242  
 remote procedure calls (RPC)  
   as serialized objects, 159–162  
   Unity game engine, 286  
   Unreal Engine 4, 283–284  
 RemovedProcessedMoves, 245

render pipeline latency, 200–201  
 rendezvous server, 184, 184f  
 replication  
   customization, 162  
   defined, 140  
   identifying class, 142–144  
   marking packet, 140–141  
   object creation registry, 144–148  
   preparatory steps, 140  
   reliability, 221–228  
   RPC as serialized object, 159–162  
   serialized object identifier, 141–142  
   Unity game engine, 286  
   world state. *See* world state  
 ReplicationCommand, 223  
 replication commands, 177  
 replication header, 153–154  
 ReplicationHeader serialization  
   code, 160  
 ReplicationManager, 161, 162, 221–227  
 ReplicationTransmissionData, 227  
 ReplicationTransmissions, 228  
 replication update packets, 177  
 representational state transfer (REST), 313–314  
 RequestCurrentStats function, 305  
 request library for REST, 329  
 reserved ports. *See* system ports  
 REST. *See* representational state transfer (REST)  
 RetrieveStatsAsync, 303  
 RFC 1122, 18  
 RMI. *See* Remote Method Invocation (RMI)  
*Robo Cat Action*, 167  
   client-server model, 170–182  
   controls for, 170  
*Robo Cat RTS*  
   hello packet, 183  
   introduction packet, 183–184  
   launching, 183  
   master peer, 183  
   peer-to-peer model, 182–196  
 roles, 282  
   authority, 282, 283  
   autonomous proxy, 282  
   simulated proxy, 282  
 round trip time (RTT), 167, 204, 234  
 routing table, 31, 31t  
 RPC. *See* remote procedure calls (RPC)  
 RPCManager, 160–161, 162  
 RSA system, 268–269

RTT. *See* round trip time (RTT)

runtime assertions, 342

## S

scalability

frequency, 263

instancings, 262

overview, 253

prioritization, 263

server partitioning/sharding, 260–262, 261*f*

visibility culling, 255–260

SD\_BOTH, 67

SD\_RECEIVE, 67

SD\_SEND, 67–68

security

input validation, 270–271

packet sniffing, 266–270

server, 274–276

software cheat detection, 271–274

seeds, 191

segment, TCP, 42–43

ACK flag, 46

SYN flag, 46

semiprime, 268

sender hardware address (variable length), 28

sender protocol address (variable length), 28

sending data

TCP socket, 85–86

UDP socket, 79–80

SendInputPacket, 180, 245

SendP2PPacket, 299

sendto function

TCP socket, 85–86

UDP socket, 79–80

sequence number (32-bits), 42–43

serialization

abstracting direction, 131–132

compression, 123–130

data driven, 133–135

defined, 102

of field's value, bits for, 149–150

maintainability, 130–135

need for, 102–105

referenced data, 119–123

streams, 105–119

Serialize function, 135

Serialize method, 131–132

serial port, 2

server function, 283

server game instance, 316

server partitioning/sharding, 260–262, 261*f*

server security

bad data, 274–275

DDoS, 274

fuzz testing, 274–275

intrusions, 276–277

timing attacks, 275–276

server side rewind, 248–249

service-level agreements, 312

setInterval call, 323

SetLobbyChatMsg function, 297

SetLobbyGameServer function, 297

setsockopt, 96

SFD. *See* start frame delimiter (SFD)

shutdown function, 67

shuttingDown state, 332

simple traversal of UDP through NAT. *See* STUN

simulated proxy, 282

simulating

jitter, 228–230

latency, 228–230

packet loss, 228–230

sin\_addr, 71–72

sin\_family, 71

sin\_port, 71

sin\_zero, 72

smart pointers. *See* pointers

sock, bind function, 78

sock, receiving data

TCP socket, 86

UDP socket, 80

sock, sending data

TCP socket, 85

UDP socket, 79

sockaddr

data type, 71

from string, 75–78

sockaddr\_in, 71

SOCK\_DGRAM, 66*t*

socket

additional options, 96, 97*t*, 98*t*

closing, 67

creating, 66–68

operating system differences, 68–71

POSIX-based platforms, 68–69

TCP, 83–88

UDP, 79–83



- Unreal Engine 4, 280
- socket address, 71–79
  - binding, 78–79
  - sockaddr from string, 75–78
- SocketAddress class, 74
- SocketAddressFactory, 77–78
- socket function, 66
  - af parameter, 66
  - protocol parameter, 67
  - type parameter, 66–67
- SOCK\_RAW, 66t
- SOCK\_SEQPACKET, 66t
- SOCK\_STREAM, 66t, 67
- software cheat detection, 271–274
  - bot, 272
  - concept, 272
  - map hacking, 272
  - VAC, 273
  - Warden, 273–274
- SO\_KEEPALIVE, 97t
- SOL\_SOCKET options, 97t
- SO\_RCVBUF, 97t
- SO\_RECVTIMEO, 97t
- SO\_REUSEADDR, 97t
- SO\_SNDBUF, 97t
- SO\_SNDTIMEO, 97t
- source address (32 bits), 26
- source port (16 bits)
  - TCP header, 42
  - UDP header, 41
- SpaceWar, 290
- sparse array compression, 124–125
- spatial approach, 254
- spawning objects, in Unity game engine, 285–286
- spear phishing attack, 276
- standard template library (STL) containers,
  - 347–350
  - array, 348
  - forward\_list, 349
  - list, 349
  - map, 349
  - unordered\_map, 349–350
  - unordered\_set, 350
  - vector, 348–349
- Starsiege: Tribes, 5–10
- start frame delimiter (SFD), 22
- start packet, 185
- Star Wars: The Old Republic, 262
- STAT, 302
- StatData instantiation, 303
- StatData structure, 302
- static assertion, 342–343
- StaticCreate function, 187–188
- StaticReadAndCreate function, 187
- static zones, 255–256
- Stat\_. Next, 302
- Stat\_NumGames, 302
- STAT(NumGames, INT), 302
- Stats.def, 302
- Steam, 290
  - integrating, 290
- SteamAPICall\_t, 295
- SteamAPI\_Init, 292
- SteamAPI\_RunCallbacks, 292–293
- steam\_appid.txt file, 292
- STEAM\_CALLBACK macro, 296
- SteamFriends function, 292
- SteamGameServer\_Init, 293
- SteamGameServer\_Shutdown, 293
- SteamUser function, 292
- SteamUtils function, 292
- Steamworks partner, 290
- Steamworks SDK Access Agreement, 290
- store and forward process, 17
- stream manager, 8
- streams
  - bit, 114–119
  - defined, 105
  - endian compatibility, 110–113
  - file output, 105
  - input, 105
  - memory, 106–110
  - network, 105
  - output, 105
- STUN, 57–59
  - data flow, 58f
  - defined, 57
  - NAT tables, 58f
  - packets exchanged, 58f
- subnet mask
  - in binary form, 30–31
  - broadcast address, 30
  - CIDR notation, 31
  - default address, 34
  - defined, 30
  - indirect routing and, 29–35
  - IP addresses and, 30, 30t
  - ISP, 32–33

- network address, 30, 31
- routing table, 31
- sample, 30t
- Sweeney, Tim, 234
- switches, 23
- symmetric NAT, 59–60
- SYN-ACK segment, 46
- SyncVars, 286
- SYN flag, 46
- system ports, 40

**T**

- target hardware address (variable length), 28
- target protocol address (variable length), 28
- TCP header, 42–43, 42f
- TCP hole punching, 60
- TCP/IP suite, 17–19
  - layers, 18–19, 18f. *See also specific layer*
- TCP\_NODELAY, 98t
- TCP phantom byte, 46
- tcpSocket, 67
- TCPSocket class, 323–324
- TCPSocket class
  - type-safe, 87–88
- TCPSocketPtr, 88
- TCP sockets, 83–88
  - connection, 83–85
  - creating, 67
  - disposing, 67
  - type-safe, 86–88
- template, 341–343
  - specialization, 342
- template metaprogramming, 150
- Terrano, Mark, 12
- third party, reliance on, 312
- third-party host, STUN and, 57–59
- time dilation, 262
- time to live (8 bits), 25
- timing attack, 275–276
- transmission control protocol (TCP), 42–52
  - concept, 42
  - data transmission, 46–51
  - delayed acknowledgment, 50
  - disconnecting, 51–52
  - Nagle's algorithm, 51
  - reliability, 207–208, 209t
  - reliable data transfer, 43–44, 44f

- segment, 42–43
- state variables, 44, 45t
- three-way handshake, 45–46, 45f
- transmission delay
  - jitter, 205
  - network latency, 203
- transport layer, 39–52
  - bind, 40
  - concept, 39–40
  - ports, 40
  - TCP, 42–52. *See also* transmission control protocol (TCP)
  - UDP, 41–42
- transport layer API
  - Unity game engine, 284–285
- transport layer protocol, 41, 41t
- TryAdvanceTurn function, 189–190
- TTL. *See* time to live (8 bits)
- TurnData class, 188
- TurnData constructor, 189
- turn timer, 11–12
- TypeAliaser, 113
- type of service (8 bits), 25
- type-safe
  - socket address, 73–74
  - TCP sockets, 86–88

**U**

- UDP. *See* user datagram protocol (UDP)
- UDP socket
  - receiving data, 80–81
  - sending data, 79–80
  - type-safe, 81–83
- udpSocket, 67
- UDP sockets
  - creating, 67
- UNetDriver class, 280
- UNET library, 284, 285
- unexpected hardware changes, 312
- uniform\_int\_distribution class, 193
- uniqueness, between networks, 54
- United States Advanced Research Projects
  - Agency, 16
- Unity game engine, 284–287
  - game objects, 285
  - matchmaking, 286–287
  - network topology, 285

- remote procedure calls, 286
- replication, 286
- spawning objects, 285–286
- transport layer API, 284–285
- Universal Plug and Play (UPnP), 60
- Unreal Engine 4
  - actor replication, 282–283
  - game object class, 281
  - networking, 280
  - network topology, 281–282
  - remote procedure calls (RPC), 283–284
  - socket subsystem, 280
- Unreliable, 284
- UnreliableSequenced, 284
- UpdateLobbyPlayers function, 296, 297
- UPnP. *See* Universal Plug and Play (UPnP)
- urgent pointer (16 bits), 43
- url, 326
- user datagram protocol (UDP), 41–42
  - checksum (16 bits), 42
  - destination port (16 bits), 41
  - length (16 bits), 41
  - reliability, 208–209, 209t
  - source port (16 bits), 41
- user passwords, 276–277
- user ports, 40
- uuid, 326

**V**

- VAC. *See* Valve Anti-Cheat (VAC)
- values
  - af parameter, 66t
  - protocol parameter, 67t
  - type parameter, 66t
- Valve Anti-Cheat (VAC), 273
- Valve Software, 290
- version (4 bits), IPv4 packet, 25
- view frustum, 256–257, 258f
- virtual machine image (VMI), 325
- virtual machine manager (VMM), 324–333
  - hash maps, 326
  - initialization and data structure, 325–326
  - members, 326–327
  - monitoring, 330–333
  - spawning and provisioning, 327–329

- virtual machines (VM), 317, 318
- visibility culling
  - defined, 255
  - hierarchical techniques, 259–260
  - PVS, 258–259, 259f
  - relevancy when not visible, 260
  - static zones, 255–256
  - view frustum, 256–257, 258f
- VMI. *See* virtual machine image (VMI)
- VMM. *See* virtual machine manager (VMM)
- VSync, 201

## W

- WAN. *See* wide area network (WAN)
- Warden, 273–274
- wide area network (WAN), 54
- Wi-Fi, 21
- Windows version of socket library, 69–71
- Winsock2-specific functions, 69–71
- Words with Friends*, 5
- World of Warcraft*, 261
- world state, 140
  - changes, 152–159
  - replication, 148–152
- world state delta, 152
- WriteBatchedCommand(), 223–225
- WriteBits methods, 114–116
- WriteForCRC function, 194
- Write function, 187
- Write method, 119
- WritePendingAcks(), 215
- WSACleanup, 70
- WSAGetLastError, 70
- WSAStartup functions, 69–70
- wVersionRequested, 69

## X

- Xbox Live games, 290
- Xbox One games, 290
- X macro, 301–303, 305

## Z

- zero network broadcast address, 35