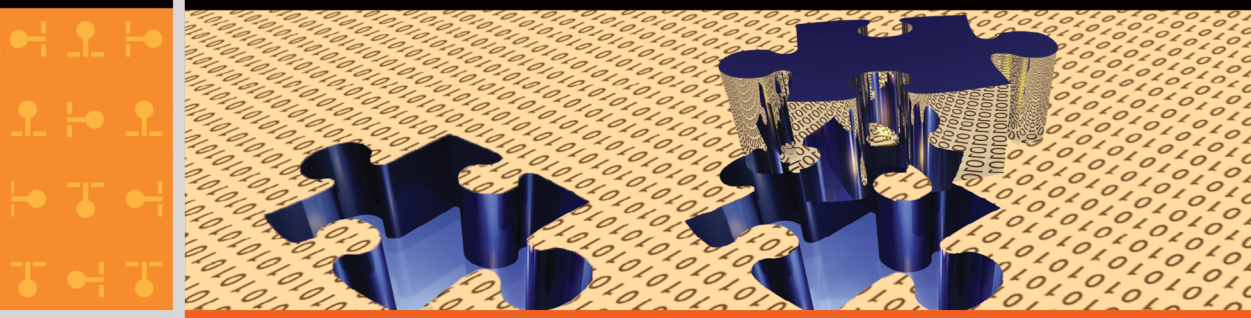
БИБЛИОТЕКА ПРОГРАММИСТА



Джошуа Глейзер, Санджай Мадхав

Многопользовательские ИГРЫ

Разработка сетевых приложений

ПИТЕР®

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

БИБЛИОТЕКА ПРОГРАММИСТА

Джошуа Глейзер, Санджай Мадхав

Многопользовательские ИГРЫ

Разработка сетевых приложений

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2017

ББК 32.988.02-018
УДК 004.738.5
Г53

Глейзер Дж., Мадхав С.

Г53 Многопользовательские игры. Разработка сетевых приложений. — СПб.: Питер, 2017. — 368 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-02290-3

Сетевые многопользовательские игры — это многомиллиардный бизнес, привлекающий десятки миллионов игроков. Эта книга на реальных примерах рассказывает об особенностях разработки таких игр и основах построения надежной многопользовательской архитектуры.

Вы узнаете об основах сетевого программирования с точки зрения разработчиков игр, управлении игрой через передачу данных, сетевых обновлениях, обеспечении надежной работы и научитесь создавать безопасный и масштабируемый код. Не останутся без внимания игровые сервисы и облачные технологии.

Эта книга пригодится всем, кто хочет узнать, как создаются сетевые игры.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-0134034300 англ.
ISBN 978-5-496-02290-3

Оглавление

Посвящаю книгу GrilledCilantro и Jellybean.
Вы знаете, о чем я.
Джошуа Глейзер

Моим родным за поддержку и всем моим коллегам,
работавшим со мной долгие годы.
Санджай Мадхав

Предисловие

Сетевые многопользовательские игры занимают значительную нишу в современной игровой индустрии. Число игроков и суммы денег, вращающиеся в этой области, поражают воображение. В 2014 году в «League of Legends» ежемесячно заходили 67 миллионов игроков. В 2015-м на чемпионате мира по игре в «DoTA 2» призовой фонд составил более 16 миллионов долларов. Продажи серии «Call of Duty», популярность которой отчасти обусловлена поддержкой многопользовательского режима, регулярно приносят более миллиарда долларов уже в первые несколько дней после выхода очередного выпуска. Даже игры, ранее бывшие только однопользовательскими, например серия «Grand Theft Auto», теперь включают сетевые компоненты поддержки многопользовательского режима.

В этой книге подробно рассматриваются все основные понятия, необходимые для создания сетевых многопользовательских игр. Вначале мы познакомимся с основами сетевых взаимодействий: как работает Интернет и как выполняется пересылка данных на другие компьютеры. Далее, после закладки фундамента, будут рассмотрены основы передачи данных в играх, как подготавливать игровые данные к передаче по сети, как изменять содержимое игровых объектов по сети и как организовать взаимодействие компьютеров, вовлеченных в игру. Затем мы поговорим о том, как компенсировать ненадежность соединения и задержки при передаче данных по сети и как обеспечить масштабируемость и безопасность игрового кода. В главах 12 и 13 рассматриваются службы объединения игроков для выделенных серверов и использование облачного хостинга — две темы, чрезвычайно важные для современных сетевых игр.

На страницах этой книги мы попытались совместить теорию с практикой: в большинстве глав не только обсуждаются важные понятия, но и приводятся примеры программного кода, необходимого для полноценной работы сетевой игры. На сайте книги вы найдете полные исходные коды двух игр. Одна из них — сюжетная игра (action game), а другая — стратегия в реальном времени (Real-Time Strategy, RTS). Чтобы помочь читателю в освоении рассматриваемых здесь тем, на протяжении всей книги мы будем изучать многочисленные версии этих двух игр.

Большая часть книги основана на программе обучения, разработанной для курса изучения программирования многопользовательских игр в Университете Южной Калифорнии. Она, в свою очередь, опирается на прекрасно зарекомендовавшую себя методику обучения разработке многопользовательских игр. Однако эту книгу не следует рассматривать только как учебное пособие — она пригодится любым разработчикам, желающим узнать, как создавать сетевые игры.

Кому адресована эта книга

Несмотря на то что в приложении рассматриваются некоторые аспекты современного языка C++, используемого в книге, предполагается, что читатель хорошо знаком с этим языком программирования. Кроме того, предполагается, что читатель уже обладает базовыми знаниями о стандартных структурах данных, обычно рассматриваемых в курсе CS2. Для тех, кто не знаком с C++ или желает освежить в памяти сведения о структурах данных, мы рекомендуем отличную книгу Эрика Робертса (Eric Roberts) «Programming Abstractions in C++».

Также предполагается, что читатель имеет представление о технологии программирования однопользовательских игр. В идеале желательно, чтобы он имел представление об игровых циклах, приемах моделирования игровых объектов, векторной математике и основах игровой физики. Если вы не знакомы с этими вещами, вам следует сначала прочитать какую-нибудь вводную книгу о программировании игр, например «Game Programming Algorithms and Techniques» Санджая Мадхава.

Как упоминалось ранее, книга одинаково хорошо подходит и студентам, и игровым программистам, желающим освоить приемы создания сетевых игр. И даже те из вас, кто давно работает в игровой индустрии, но прежде не занимался созданием сетевых игр, найдут в этой книге немало полезного.

Условные обозначения

Программный код повсюду в книге оформлен моноширинным шрифтом. Небольшие фрагменты кода могут находиться непосредственно в основном тексте или в отдельных абзацах:

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

Более длинные фрагменты кода оформлены в виде листингов, как показано в листинге 0.1.

Листинг 0.1. Пример листинга

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

В тексте вам иногда будут встречаться абзацы, оформленные как примечания, советы, врезки и предупреждения, примеры которых приводятся далее.

ПРИМЕЧАНИЕ Примечания помогают отделить полезную информацию от основного текста. Внимательно отнеситесь к тому, о чем в них говорится.

СОВЕТ Советы содержат подсказки, которые помогут при реализации определенных систем в коде игры.

ВНИМАНИЕ Предупреждения содержат очень важную информацию, обычно касающуюся типичных ошибок или проблем, и способы предупреждения или решения этих проблем.

ВРЕЗКА

Врезки содержат обсуждения, обычно отклоняющиеся от основной темы главы. Во врезках могут приводиться любопытные сведения о различных проблемах, но несущественные с точки зрения обучения.

Почему C++?

Подавляющее большинство примеров программного кода в этой книге написано на языке C++, потому что он фактически остается основным языком программирования, используемым в игровой индустрии и программистами, разрабатывающими игровые движки (game engine). Некоторые движки позволяют написать большую часть игры на других языках, например Unity — на C#, тем не менее важно помнить, что львиная доля низкоуровневого кода в таких движках все еще написана на C++. Поскольку главной темой книги является разработка сетевых многопользовательских игр, что называется, «с нуля», определенно имеет смысл рассматривать эту тему с использованием языка, на котором написаны игровые движки. Но даже если весь сетевой код для своих игр вы будете писать на каком-то другом языке, основные принципы останутся практически теми же. И все же весьма желательно, чтобы вы были знакомы с C++, иначе примеры кода будут вам непонятны.

Почему JavaScript?

Появившийся на свет как наспех сделанный язык сценариев для поддержки браузера Netscape, JavaScript эволюционировал до стандартизованного, полноценного и до некоторой степени функционального языка. Его популярность как языка клиентских сценариев помогла ему перебраться на сторону сервера, где его процедуры, имеющие форму объектов, простой синтаксис замыканий и динамическая типизация весьма эффективно ускоряют разработку служб, управляемых событиями. Программный код на этом языке труднее поддается рефакторингу и имеет худшую производительность, чем код на C++, что делает его не лучшим выбором для разработки клиентской части игр. Но эти качества не являются проблемой для серверной части, когда для вертикального масштабирования службы иногда достаточно лишь сдвинуть бегунок вправо. Примеры серверных служб в главе 13 написаны на JavaScript, и для их понимания требуется довольно близкое знакомство с этим языком. На момент написания книги JavaScript был,

по оценкам GitHub, самым активно используемым языком. Его доля составляла 50 % всех проектов. Следование за тенденциями ради тенденций редко приводит к хорошим результатам, но умение программировать на самом популярном в мире языке дает определенные выгоды.

Веб-сайт книги

Для этой книги создан вспомогательный веб-сайт: *https://github.com/MultiplayerBook*. На сайте вы найдете ссылки на примеры кода из книги. Кроме того, там содержатся список ошибок и опечаток, ссылки на слайды PowerPoint и план курса обучения в университете.

Благодарности

Мы хотели бы выразить свою благодарность всем сотрудникам издательства Pearson за помощь в создании книги. В их числе: наш ответственный редактор Лаура Левин (Laura Lewin), убедившая нас собраться вместе и написать эту книгу; помощник редактора Оливия Беседжио (Olivia Basegio), сделавшая все возможное для обеспечения рабочего процесса; редактор-консультант Майкл Терстон (Michael Thurston), помогавший работать над наполнением книги. Мы также хотим сказать спасибо всему техническому персоналу, включая выпускающего редактора Энди Бистера (Andy Beaster), и всем сотрудникам Cenveo® Publisher Services.

Наши технические редакторы Александр Бочар (Alexander Boczar), Джонатан Ракер (Jonathan Rucker) и Джефф Такер (Jeff Tucker) сыграли важную роль в исправлении ошибок в книге. Мы хотели бы поблагодарить их за то, что они нашли время в своем плотном графике для ее рецензирования. Наконец, мы выражаем благодарность Valve Software за предоставленную нам возможность написать о Steamworks SDK, а также за рецензирование главы 12.

Благодарности от Джошуа Глейзера

Спасибо большое Лори (Lori) и Маккинни (McKinney) за полное понимание, поддержку, любовь и хорошее настроение. Вы — самая лучшая семья. Я потратил на эту книгу массу времени, оторвав его от общения с вами, но теперь я свободен! Ура! Спасибо вам, мама и папа, за вашу поддержку и любовь, за вашу уверенность в том, что их сын способен написать книгу почти так же хорошо, как он пишет код. Спасибо тебе, Бет (Beth), за массу удивительного, что ты сделала для мира в целом, и за то, что иногда присматривала за моими кошками. Спасибо всей моей многочисленной семье за поддержку и веру и за восхищение тем, что я пишу учебник. Спасибо Чарльзу (Charles) и всем моим коллегам-профессионалам из Naked Sky (программистам), которые помогали мне сохранять бдительность и указывали на мои ошибки. Спасибо вам, Тянь (Tian) и Сэм (Sam), что затащили меня в эту индустрию развлечений. Спасибо вам, сенсей Коппинг (Sensei Copping), за науку, что человек, который убирает свой дом, но оставляет грязным туалет, разрушает себя. И конечно, спасибо тебе, Санджай, за то, что пригласил меня на борт USC и принял меня в свою команду! Я никогда не смог бы довести это дело до конца без твоей мудрости и трезвости мышления, не говоря уже о том, что половина написанного принадлежит тебе! (Ах, да, спасибо тебе, Лори, еще раз, так, на всякий случай, если на первую благодарность ты не обратила внимания!)

Благодарности от Санджая Мадхава

Между количеством книг, написанных автором, и объемом благодарностей есть определенная зависимость. Так как я написал массу благодарностей в своей последней книге, на этот раз буду краток. Я хочу, конечно же, выразить благодарность моим родителям и сестре, сказать спасибо моим коллегам из программы *Information Technology Program* Университета Южной Калифорнии (USC). Наконец, я хочу сказать спасибо Джошуа за согласие изучить наш курс «Программирование многопользовательских игр», потому что без этого курса не увидела бы свет и эта книга.

Об авторах

Джошуа Глейзер (Joshua Glazer) — сооснователь и технический директор *Naked Sky Entertainment*, независимой студии, занимающейся разработкой игр для приставок и персональных компьютеров, таких как «RoboBlitz», «MicroBot», «Twister Mania», а кроме того, модных хитов для мобильных устройств: «Max Axe» и «Scrap Force». Как руководитель *Naked Sky* он консультировал несколько внешних проектов, включая «Unreal Engine» в Epic Games, «League of Legends» в Riot Games, «Destroy All Humans» в THQ и множество других проектов для *Electronic Arts*, *Midway*, *Microsoft* и P*aramount Pictures*.

По совместительству Джошуа читает лекции в Университете Южной Калифорнии, где и заинтересовался курсами по программированию многопользовательских игр и разработке игровых движков.

Санджай Мадхав (Sanjay Madhav) — старший преподаватель Университета Южной Калифорнии, ведет несколько курсов по программированию и созданию видеоигр. Основной из них — программирование игр для студентов старших курсов, который он читает с 2008 года. Также он ведет несколько других курсов, по таким темам, как игровые движки, структуры данных и разработка компиляторов. Является автором книги «Game Programming Algorithms and Techniques».

До поступления на работу в Университет Южной Калифорнии Санджай работал программистом в нескольких компаниях, занимавшихся разработкой видеоигр, включая *Electronic Arts*, *Neversoft* и *Pandemic Studios*. Участвовал в создании множества игр, включая «Medal of Honor: Pacific Assault», «Tony Hawk's Project 8», «Lord of the Rings: Conquest» и «The Saboteur», — большинство из которых в той или иной степени можно назвать многопользовательскими сетевыми играми.

1 Обзор сетевых игр

Идея сетевых многопользовательских игр не была популярна у основных производителей игр до 1990-х, впрочем, нет правил без исключений. В этой главе мы сначала познакомимся с краткой историей эволюции многопользовательских игр, от первых сетевых игр 1970-х до обширной современной индустрии. А затем будет представлен обзор архитектуры двух популярных сетевых игр из 1990-х — «Starsiege: Tribes» и «Age of Empires». Многие приемы, применявшиеся в этих играх, используются и в наши дни, поэтому данный обзор поможет лучше понять сложности создания сетевой многопользовательской игры.

Краткая история многопользовательских игр

Первые прародители современных сетевых многопользовательских игр начали появляться еще в 1970-х и работали на университетских мейнфреймах. Однако игры этого рода не получили широкого распространения, пока Интернет не превратился из информационного чуда в нечто обыденное, — это произошло во второй половине 1990-х. В этом разделе мы поговорим о том, как появились первые сетевые игры и в каких направлениях шло их развитие почти полвека, начиная с появления первых игровых «динозавров».

Локальные многопользовательские игры

Некоторые из ранних видеоигр были *локальными многопользовательскими*, в том смысле, что допускали возможность участия двух и более игроков в игре на одном компьютере. К числу таких игр относятся самые первые игры, включая «Tennis for Two» (1958) и «Spacewar!» (1962). Локальные многопользовательские игры программируются практически так же, как однопользовательские. Единственное отличие заключалось в наличии нескольких изображений игровой сцены на экране и/или поддержке нескольких устройств ввода. Так как программирование

локальных многопользовательских игр очень схоже с программированием однопользовательских игр, в этой книге к нему мы больше не вернемся.

Ранние сетевые многопользовательские игры

Первые *сетевые многопользовательские игры* работали в небольших сетях, состоящих из больших ЭВМ. Главное отличие сетевой многопользовательской игры от локальной многопользовательской состоит в том, что в активный сеанс игры включены два или более компьютера, соединенных друг с другом. Одной из таких сетей больших ЭВМ была система PLATO, разработанная в Иллинойском университете. Именно в системе PLATO была создана одна из первых сетевых игр — пошаговая стратегия «Empire» (1973). Примерно в то же время увидела свет и сетевая игра «от первого лица» «Maze War», и до сих пор нет единого мнения о том, какая из двух может претендовать на первородство.

С началом распространения персональных компьютеров в конце 1970-х разработчики искали способы организации взаимодействия двух компьютеров через *последовательные порты*. Последовательный порт позволяет передавать данные по одному биту, и основным его назначением было взаимодействие с внешними устройствами, такими как принтеры или модемы. Однако с его помощью также можно было связать два компьютера и организовать обмен данными между ними. Это позволяло создать игровой сеанс, объединяющий несколько персональных компьютеров, благодаря чему появились первые сетевые игры для PC. В декабрьском номере журнала «BYTE» за 1980 год была опубликована статья Вассермана и Страйкера о том, как программировать так называемые многомашинные игры на Бейсике.

Но большой недостаток использования последовательных портов состоял в том, что компьютеры обычно имели не более двух таких портов (если, конечно, не использовалась карта расширения). Это означало, что для соединения более чем двух компьютеров через последовательные порты необходимо было использовать *гирляндную* схему соединения, в которой множество компьютеров объединялось в кольцо. Такая организация компьютеров является одним из вариантов топологии сети, о которых подробнее рассказывается в главе 6 «Топологии сетей и примеры игр».

Так, несмотря на наличие технологии, доступной уже в начале 1980-х, большинство игр, выпущенных в течение десятилетия, не использовало имевшиеся сетевые возможности. Только в 1990-е идея объединения нескольких компьютеров в локальную сеть для игры получила дальнейшее развитие, о чем рассказывается ниже в этой главе.

Многопользовательский мир

Многопользовательский мир, или МПМ (Multi-User Dungeon, MUD), — это обычная текстовая многопользовательская игра, в которой несколько игроков подключаются к общему виртуальному миру. Игры этого вида впервые появились на больших ЭВМ в крупных университетах, а сама аббревиатура MUD происходит от названия игры «MUD» (1978), созданной Робом Трушоу из Эссекского универси-

тета. Многопользовательский мир можно считать ранней компьютерной версией ролевой игры «Dungeons and Dragons», хотя не все многопользовательские миры были ролевыми играми.

С ростом мощности персональных компьютеров производители аппаратного обеспечения начали выпускать модемы, позволяющие двум компьютерам обмениваться данными по стандартным телефонным линиям. Несмотря на то что скорости обмена были чрезвычайно низкими по современным меркам, модемы давали возможность играть в многопользовательских мирах уже вне стен университетов. Некоторые запускали игры МПМ в *электронных досках объявлений* (Bulletin Board System, BBS), позволявших нескольким пользователям подключаться посредством модемов и выполнять множество действий, в том числе и играть в игры.

Игры в локальных сетях

Термин *локальная сеть* (Local Area Network, LAN) используется для описания соединения нескольких компьютеров в относительно небольшую сеть. Для создания локальных соединений могли использоваться самые разные механизмы — одним из примеров служит объединение в сеть через последовательные порты, как описывалось выше в этой главе. Однако звездный час локальных сетей пробил с появлением Ethernet (протокола, который подробнее рассматривается в главе 2 «Интернет»).

Игру «Doom» (1993) во многом можно считать прародителем современных сетевых игр, хотя она и не была первой игрой, обеспечивающей многопользовательский режим в локальной сети. Первая версия *шутера от первого лица*, выпущенная компанией *id Software*, поддерживала до четырех игроков в одном сеансе игры с возможностью играть вместе против компьютера или каждый сам за себя. Так как «Doom» был активной игрой с быстро изменяющейся игровой ситуацией, он потребовал реализации нескольких важнейших идей, описываемых в этой книге. Конечно, все эти приемы претерпели существенные изменения после 1993 года, но первенство «Doom» в данной области никем не оспаривается. Подробнее история создания «Doom» описывается в книге «Masters of Doom» (см. ссылку в конце главы).

Многие игры, предполагавшие использование многопользовательского режима, поддерживали взаимодействия игроков не только в локальных сетях, но также через модемные или телекоммуникационные соединения. Через несколько лет уже подавляющее большинство сетевых игр обеспечило поддержку локальных сетей. Это привело к появлению клубов, оснащенных соединенными в сеть компьютерами, в которых собирались любители сетевых игр. Несмотря на то что некоторые сетевые многопользовательские игры все еще выпускаются с поддержкой локальных сетей, в последние годы на рынке доминируют многопользовательские *онлайн-игры*.

Онлайн-игры

В онлайн-игре игроки связываются друг с другом посредством некоторой глобальной сети. В настоящее время онлайн-игры часто называют интернет-играми, но термин «онлайн» намного шире и включает также некоторые ранние сети, такие как CompuServe, которые раньше не были соединены с Интернетом.

С началом бурного развития Интернета в конце 1990-х начался расцвет онлайн-игр. В числе игр, завоевавших популярность в этот период, можно назвать «Quake» (1996) компании *id Software* и «*Unreal*» (1998) компании *Epic Game*.

Хотя может показаться, что онлайн-игры реализуются так же, как игры для локальных сетей, главной проблемой онлайн-игр являются *задержки*, неизбежно возникающие при передаче данных по сети. Первоначальная версия «Quake», например, вообще не предполагала взаимодействия через интернет-соединения, и только с выходом исправлений QuakeWorld появилась возможность игры через Интернет. Методы компенсации сетевых задержек более подробно рассматриваются в главе 7 «Задержки, флуктуации и надежность» и в главе 8 «Улучшенная обработка задержек».

С появлением в 2000-х таких служб, как Xbox Live и PlayStation Network, являвшихся прямыми потомками служб для персональных компьютеров типа GameSpy и DWANGO, стала возможной поддержка онлайн-игр на игровых консолях. В часы пик этими службами пользуются миллионы активных пользователей, хотя, с распространением потокового видео и других служб для игровых консолей, не все эти активные игроки действительно играют в игры. В главе 12 «Игровые службы» рассказывается, как интегрировать одну из таких служб — Steam — в игру для персонального компьютера.

Массовые многопользовательские онлайн-игры

Даже в наши дни большинство многопользовательских онлайн-игр ограничивается поддержкой небольшого числа игроков в одном игровом сеансе — обычно от 4 до 32. Однако в *массовой многопользовательской онлайн-игре* (Massively Multiplayer Online Game, MMO) в одном игровом сеансе могут принимать участие сотни, если не тысячи игроков. Большинство массовых онлайн-игр — это ролевые игры (Role-Playing Games), и поэтому их обозначают аббревиатурой *MMORPG*. Но существуют также и другие виды массовых онлайн-игр, например *шутеры от первого лица* (First-Person Shooters, MMOFPS).

Во многих случаях игры MMORPG можно считать графическим развитием многопользовательских миров. Некоторые из ранних игр MMORPG фактически появились до широкого распространения Интернета и функционировали в коммутируемых сетях, таких как Quantum Link (более позднее название — America Online) и CompuServe. Одной из первых таких игр стала «Habitat» (1986), в которой были реализованы некоторые элементы новейшей технологии, описанные в книге Морнигстара и Фармера (см. ссылку в конце главы). Однако только с распространением Интернета игры этого жанра стали обретать популярность. Первым настоящим хитом стала «Ultima Online» (1997).

Другие игры MMORPG, такие как «EverQuest» (1999), также имели определенный успех, но настоящий взрыв интереса во всем мире произошел с выходом «World of Warcraft» (2004). В какой-то момент игра от Blizzard собрала более 12 миллионов активных подписчиков по всему миру и стала настолько значительной частью популярной культуры, что в 2006 году легла в основу одного из эпизодов американского мультипликационного сериала «Южный парк» (South Park).

Создание массовых онлайн-игр сопряжено с решением сложных технических задач, часть из которых обсуждается в главе 9 «Масштабируемость». Однако большинство приемов, используемых при этом процессе, далеко выходит за рамки нашей книги. Кроме того, прежде чем даже рассматривать саму возможность создания массовой онлайн-игры, следует ознакомиться с основами создания менее масштабных сетевых игр.

Мобильные сетевые игры

С появлением игр для мобильных устройств не остались в стороне и многопользовательские игры. Многие из них на мобильных платформах являются *асинхронными* — обычно со сменой хода — и не требуют передачи данных в реальном времени. В этой модели игроки извещаются о том, что наступает их право сделать ход, и имеют довольно много времени, чтобы выполнить его. Асинхронная модель существовала с самого момента появления сетевых многопользовательских игр. Некоторые электронные доски объявлений имели единственную телефонную линию для входящих соединений, а это означало, что в каждый момент времени к таким системам мог быть подключен только один пользователь. То есть игрок должен был подключиться, сделать свой ход и отключиться. Затем в следующий момент второй игрок может подключиться, чтобы сделать ответный ход.

В качестве примера мобильной игры, использующей асинхронную многопользовательскую модель, можно привести «Words with Friends» (2009). С технической точки зрения асинхронная сетевая игра проще в реализации, чем игра, действующая в режиме реального времени, особенно для мобильных устройств, потому что прикладные программные интерфейсы мобильных платформ (Application Program Interface, API) уже имеют функции для асинхронных взаимодействий. Использование асинхронной модели для мобильных игр на ранней стадии развития мобильных сетей было практически невозможно из-за низкой надежности этих сетей по сравнению с проводными соединениями. Однако с быстрым ростом числа устройств, поддерживающих Wi-Fi, и улучшением качества мобильных сетей стало появляться все больше сетевых игр реального времени для этих устройств. Примером такой игры, использующей преимущества сетевых взаимодействий в реальном масштабе времени, является «Hearthstone: Heroes of Warcraft» (2014).

Starsiege: Tribes

«Starsiege: Tribes» — это научно-фантастический шутер от первого лица, созданный в конце 1998 года. В то время эта игра считалась одной из лучших благодаря возможности вести динамичное сражение относительно большому количеству игроков. Некоторые игровые режимы разрешали одновременное участие до 128 игроков в локальных сетях или Интернете. Чтобы получить представление о степени сложности реализации таких игр, примите во внимание, что в тот период подавляющее большинство игроков подключалось к Интернету через коммутируемые линии. Пользователи коммутируемых соединений в лучшем случае имели модем, под-

держивающий скорость передачи данных до 56,6 кбит/с. Более того, игра «Tribes» поддерживала подключения со скоростью всего 28,8 кбит/с. По современным меркам это чрезвычайно низкая скорость. Еще один фактор: коммутируемые соединения страдали относительно большими задержками — задержки в несколько сотен миллисекунд были обычным делом.

Может показаться, что сетевая модель для игр с низкими требованиями к пропускной способности уже устарела. Но несмотря на это модель, использовавшаяся в «Tribes», по-прежнему остается востребованной в наши дни. Этот раздел описывает исходную сетевую модель игры «Tribes» лишь в общих чертах — более подробное ее описание можно найти в статье Фронмайера и Гифта, ссылка на которую приводится в конце главы.

Не пугайтесь, если какие-то идеи, рассматриваемые здесь, останутся для вас непонятными. Цель этого раздела — дать общее представление об архитектуре сетевой многопользовательской игры, чтобы вы могли оценить количество технических проблем, которые необходимо было решить. Темы, затрагиваемые в этом разделе, более подробно будут рассматриваться на протяжении всей книги. Кроме того, игра «RoboCat Action», создание которой разбирается в этой книге, использует модель, похожую на сетевую модель «Tribes».

Первое, что необходимо сделать при проектировании сетевой игры, — выбрать *протокол взаимодействий*, то есть определить соглашения, в соответствии с которыми два компьютера будут обмениваться данными. В главе 2 «Интернет» рассказывается, как функционирует Интернет, и описываются наиболее широко используемые протоколы. В главе 3 «Сокеты Беркли» мы поговорим о библиотеке, используемой для упрощения организации взаимодействий посредством этих протоколов. В рамках материала данной главы скажем только, что ради высокой эффективности «Tribes» использует *ненадежный* протокол. То есть протокол, который не гарантирует доставку данных, отправленных в сеть.

Однако использование ненадежного протокола может вызывать проблемы, когда требуется разослать данные, важные для всех игроков в игре. Поэтому инженеры должны были решить проблему сортировки отправляемых данных по степени важности. Разработчики «Tribes» в конечном итоге пришли к схеме деления данных на четыре категории:

1. **Негарантированные данные.** Как можно догадаться, эти данные не оказывают существенного влияния на ход игры. При уменьшении пропускной способности алгоритм игры может решить остановить передачу этих данных.
2. **Гарантированные данные.** Для этих данных гарантируется их доставка адресату, причем в определенном порядке следования. К этой категории относятся данные, оказывающие существенное влияние на ход игры, например событие выстрела из оружия.
3. **Данные «с самым последним состоянием».** К этой категории относятся данные, для которых важно иметь самую последнюю их версию, например число попаданий в определенного игрока. Информация о числе ранений игрока перестает быть актуальной и нуждается в обновлении, когда игрок получает очередное ранение.

4. **Гарантированно скоростные данные.** Этим данным устанавливается самый высокий приоритет, чтобы обеспечить максимально высокую скорость передачи с гарантированной доставкой. Примером такого рода данных является информация о перемещениях игрока, которые обычно происходят очень быстро и потому должны быстро передаваться.

В сетевой модели «Tribes» пришлось реализовать множество практических решений, чтобы обеспечить передачу данных этих четырех типов.

Другим важным архитектурным решением стала реализация модели «*клиент-сервер*» вместо модели «*точка-точка*». В модели «клиент-сервер» все игроки подключаются к центральному серверу, тогда как в модели «точка-точка» каждый игрок связывается с каждым другим игроком. Как обсуждается в главе 6 «Топологии сетей и примеры игр», модель «точка-точка» требует пропускной способности $O(n^2)$. То есть пропускная способность должна расти в квадратичной прогрессии с увеличением числа пользователей. При значении n, равном 128, полоса пропускания, выделенная для одного игрока, окажется ничтожно малой. Чтобы избежать этой проблемы, в «Tribes» была реализована модель «клиент-сервер». В этой конфигурации требования к пропускной способности для каждого игрока выражаются постоянной величиной, а сервер должен иметь полосу пропускания всего лишь $O(n)$. Однако это означает, что сервер должен находиться в сети, допускающей существование сразу нескольких входящих подключений, что в те времена было возможно только на предприятиях и в университетах.

Далее, сетевые операции в «Tribes» были разбиты на несколько уровней, образующих сетевую модель «Tribes» в виде своеобразного «многослойного пирога». Эта модель изображена в табл. 1.1. Ниже кратко описывается каждый из уровней модели.

Таблица 1.1. Основные компоненты сетевой модели «Tribes»

<table>
<tr><td colspan="4">Уровень модели игры</td></tr>
<tr><td>Диспетчер фантомов</td><td>Диспетчер перемещений</td><td>Диспетчер событий</td><td>Остальное...</td></tr>
<tr><td colspan="4">Диспетчер потоков</td></tr>
<tr><td colspan="4">Диспетчер соединений</td></tr>
<tr><td colspan="4">Модуль обслуживания пакетов</td></tr>
</table>

Модуль обслуживания пакетов

Пакет — это набор данных определенного формата, пересылаемый по сети. В модели «Tribes» *модуль обслуживания пакетов* образует самый нижний уровень. Это единственный уровень, зависящий от особенностей платформы. В сущности, этот уровень служит оберткой для стандартного API сокетов, с помощью которого осуществляется конструирование и передача пакетов различных форматов. Реализация этого уровня похожа на реализацию систем, представленных в главе 3 «Сокеты Беркли».

Так как в «Tribes» используется ненадежный протокол, разработчикам пришлось добавить механизм обработки данных, требующих гарантированной доставки. По аналогии с решением, обсуждаемым в главе 7 «Задержки, флуктуации и надежность», в «Tribes» реализован собственный уровень поддержки надежности. Однако этот уровень надежности не используется модулем обслуживания пакетов; вместо него ответственность за поддержку надежности возлагается на высокоуровневые диспетчеры, такие как *диспетчер фантомов* (ghost manager), *диспетчер перемещений* (move manager) и *диспетчер событий* (event manager).

Диспетчер соединений

Задача *диспетчера соединений* заключается в абстракции сетевого соединения двух компьютеров. Он принимает данные от уровня, лежащего выше, — *диспетчера потоков* — и передает их на уровень ниже — *модулю обслуживания пакетов*.

Диспетчер соединений все еще остается ненадежным. Он *не гарантирует* доставку переданных ему пакетов, но гарантирует возврат *уведомления о доставке* — другими словами, дает возможность проверить состояние запроса, переданного диспетчеру соединений. Благодаря этому уровень, находящийся выше диспетчера соединений (диспетчер потоков), может узнать, были ли доставлены отправленные им данные.

Уведомление о доставке реализовано с помощью битового поля скользящего окна квитирования. В оригинальной статье с описанием сетевой модели «Tribes» отсутствует подробная информация о реализации диспетчера соединений, но она напоминает реализацию системы, обсуждаемой в главе 7 «Задержки, флуктуации и надежность».

Диспетчер потоков

Основной задачей *диспетчера потоков* является передача данных диспетчеру соединений. Один из важнейших аспектов этой задачи — определение максимально допустимой скорости передачи данных. Она зависит от качества подключения к Интернету. В оригинальной статье приводится пример, где пользователь с модемным соединением на скорости 28,8 кбит/с может установить скорость передачи, равную 10 пакетам в секунду, при максимальном размере пакета 200 байт, что соответствует скорости примерно 2 кбайт/с. Эта информация о соединении с клиентом (скорость и размер) передается на сервер, чтобы гарантировать, что сервер не «затопит» соединение с клиентом слишком большим объемом данных.

Так как все другие системы посылают свои данные через диспетчер потоков, последний также отвечает за ранжирование запросов по приоритетам. Информация от диспетчеров перемещений, событий и фантомов получает высший приоритет, в том случае, если полоса пропускания имеет существенные ограничения. Когда диспетчер потоков определяет, какие данные следует отправить, он передает пакеты диспетчеру соединений. В свою очередь диспетчеры более высокого уровня извещаются диспетчером потоков о состоянии доставки.

Из-за ограничений на интервал следования и размер пакетов, определяемых диспетчером потоков, в один пакет могут быть включены разнородные данные. Напри-

мер, пакет может нести некоторые данные от диспетчера перемещений, некоторые данные от диспетчера событий и некоторые данные от диспетчера фантомов.

Диспетчер событий

Диспетчер событий управляет очередью событий, генерируемых уровнем модели игры. Эти события можно считать простейшей формой *удаленного вызова процедур* (Remote Procedure Call, RPC) — функций, которые могут выполняться на удаленных компьютерах. Подробнее механизмы RPC обсуждаются в главе 5 «Репликация объектов».

Например, когда игрок производит выстрел, диспетчеру событий наверняка будет передано событие «игрок выстрелил». Это событие затем может быть отправлено на сервер, который проверит корректность события и выполнит программный код выстрела из оружия. На диспетчер событий также возлагается ответственность ранжирования событий по приоритетам — он будет пытаться произвести запись максимального числа высокоприоритетных событий, пока не будет выполнено любое из следующих условий: «пакет заполнен», «очередь событий опустела» или «слишком много активных событий».

Диспетчер событий также фиксирует факт доставки событий, отмеченных как надежные. Благодаря этому в нем легко реализуется надежная передача. Если доставка надежного события не была подтверждена, диспетчер событий может просто поставить событие в начало очереди и попытаться отправить его повторно. Конечно, не все события будут отмечены как надежные. Для ненадежных событий нет необходимости фиксировать факт доставки.

Диспетчер фантомов

Диспетчер фантомов является, пожалуй, самой важной системой для поддержки до 128 игроков. На высоком уровне задача диспетчера фантомов заключается в хранении *дубликатов* (фантомов) *динамических* объектов, предполагаемых релевантными для данного конкретного клиента. Иными словами, сервер посылает клиенту информацию о динамических объектах, но только о тех из них, о существовании которых, по мнению сервера, клиент должен знать. Уровень модели игры определяет, о чем клиент *должен* знать и о чем ему было бы знать *желательно*. Это обеспечивает естественное разделение игровых объектов по приоритетам: объекты, о которых клиент должен знать, имеют высший приоритет, а объекты, о которых знать желательно, — низший. Определить релевантность того или иного объекта можно множеством разных способов. Некоторые из них рассматриваются в главе 9 «Масштабируемость». Но вообще говоря, определение релевантности объектов во многом зависит от специфики игры.

Независимо от того, как определяется релевантность объектов, диспетчер фантомов должен передать клиенту максимально возможное число релевантных объектов. Очень важно, чтобы диспетчер фантомов гарантировал успешную передачу самых актуальных данных всем клиентам. Причина такого требования в том, что информация в дубликатах игровых объектов часто содержит сведения об уровне

здоровья, вооружении, количестве боеприпасов и т. д. — то есть сведения, актуальность которых очень важна.

Когда объект становится *релевантным* (или оказывается в области видимости), диспетчер фантомов записывает некоторую информацию в объект, который соответственно называется *фантомной записью* (ghost record). Эта запись содержит такие элементы, как уникальный идентификатор (ID), маска состояния, приоритет и изменение состояния (независимо от того, отмечен ли объект как находящийся в области видимости).

Перед передачей фантомных записей объекты сначала упорядочиваются по изменениям состояния, а затем по уровню приоритета. После того как диспетчер фантомов определит, какие объекты должны передаваться, их данные добавляются в исходящий пакет с использованием приема, который описывается в главе 5 «Репликация объектов».

Диспетчер перемещений

Задачей *диспетчера перемещений* является максимально быстрая передача информации о перемещениях игрока. Если вам доводилось играть в многопользовательские игры с быстро меняющимися сценами, вы понимаете, насколько важна точность информации о перемещениях. Если информация о позиции игрока запаздывает, другие участники игры могут пытаться стрелять в то место, где игрок был совсем недавно, а не туда, где он действительно находится, из-за чего могут складываться ошибочные представления об обстановке в игре. Быстрое обновление информации о перемещениях может служить важным способом снижения восприятия задержек игроками.

Другая причина, по которой диспетчер перемещений имеет высший приоритет, обусловлена частотой обновления входных данных, составляющей 30 раз в секунду. Это означает, что через каждую 1/30 секунды становится доступной новая информация, которую нужно передать как можно быстрее. Высший приоритет также означает, что с появлением данных о перемещениях диспетчер потоков должен добавлять их в исходящие пакеты в первую очередь. Каждый клиент обязан передавать свою информацию о перемещениях на сервер. Затем сервер использует эту информацию в модели игры и отправляет клиенту подтверждение приема сведений о перемещениях.

Остальные системы

В модели игры «Tribes» есть еще несколько систем, хотя они не так важны для понимания общей архитектуры. Например, имеется *диспетчер блоков данных* (datablock manager), обслуживающий передачу статических игровых объектов. Он отличается от диспетчера фантомов, обслуживающего динамические объекты. Примером статического объекта может служить огневая установка (орудийная или пулеметная башня) — объект, который сам никуда не перемещается, но игрок может взаимодействовать с ним.

Age of Empires

Так же как «Tribes», *стратегия в реальном времени* (Real-Time Strategy, RTS) «Age of Empires» была выпущена в конце 1990-х. Разработчики «Age of Empires» столкнулись с теми же ограничениями полосы пропускания и задержками, свойственными коммутируемым подключениям к Интернету. В игре «Age of Empires» использована модель *детерминированного соответствия* (deterministic lockstep model) сетевых взаимодействий. В этой модели все компьютеры соединены друг с другом, то есть используется модель соединений «точка-точка». Все узлы одновременно и *детерминированно* выполняют код модели игры. Детерминированность достигается постоянным взаимодействием узлов для синхронизации на всем протяжении игры. Модель детерминированного соответствия все еще широко используется в современных играх-стратегиях реального времени, несмотря на то что ей уже очень много лет. Пример игры «RoboCat RTS», который приводится далее в книге, как раз реализует модель детерминированного соответствия.

Одним из существенных отличий сетевой многопользовательской модели RTS от FPS (шутеров от первого лица) является количество релевантных объектов. В «Tribes» даже при наличии 128 игроков в любой момент времени лишь часть этих игроков так или иначе влияют на игровую обстановку конкретного клиента. То есть диспетчеру фантомов в «Tribes» очень редко приходится пересылать информацию о более чем 20–30 фантомах сразу.

В стратегиях реального времени, таких как «Age of Empires», ситуация иная. Несмотря на небольшое количество игроков (оригинальная игра поддерживает не более восьми игроков), каждый игрок может управлять большим количеством объектов. Оригинальная игра «Age of Empires» позволяет каждому игроку управлять 50 объектами, а более поздние версии игры повышают это ограничение до 200. Если взять за основу число 50, это будет означать, что в массовой битве восьми игроков одновременно может использоваться до 400 объектов. Даже при том, что естественно возникает вопрос о возможности применения некоторой системы определения релевантности, чтобы уменьшить число объектов, подлежащих синхронизации, необходимо учитывать самый худший сценарий. Как быть, если в конце игры встретились армии всех восьми игроков? В этом случае релевантными могут оказаться несколько сотен объектов. Синхронизировать такое количество объектов очень сложно, даже если для каждого из них передавать минимально возможный объем информации.

Для решения этой проблемы инженеры из проекта «Age of Empires» решили синхронизировать не объекты, а команды, выполняемые каждым игроком. Это тонкое, но очень важное отличие — даже профессиональный игрок в RTS не способен выполнить более 300 команд в минуту. То есть даже в самой экстремальной ситуации игре придется передавать от каждого игрока не более нескольких команд в секунду. Для этого требуется намного меньшая полоса пропускания, чем в случае синхронизации информации о нескольких сотнях объектов. Однако учитывая, что игра не передает никакой информации об объектах, каждый экземпляр должен независимо выполнять команды, передаваемые каждым игроком. А так как каждый экземпляр действует независимо, крайне важно, чтобы все они оставались синхро-

низованными друг с другом. Это обстоятельство влечет самую большую проблему в реализации модели детерминированного соответствия.

Таймеры ходов

Так как каждый экземпляр игры действует независимо, имеет смысл использовать преимущества топологии «точка-точка». Как обсуждается в главе 6 «Топологии сетей и примеры игр», одним из преимуществ модели «точка-точка» является более высокая скорость доставки данных. Это объясняется отсутствием промежуточного звена «в лице» сервера. Однако при этом каждый игрок должен посылать свою информацию уже не единственному серверу, а всем другим игрокам. Так, например, если игрок A дал команду атаковать, все другие экземпляры игры должны получить эту команду, иначе развитие событий в них пойдет по другому сценарию.

Существует еще один важный фактор, который необходимо учитывать: игроки могут иметь компьютеры с разной производительностью и разные по качеству подключения к Интернету. Вернемся к примеру, где игрок A дает команду начать атаку: очень важно, чтобы команда о начале атаки не начала выполняться немедленно у игрока A. Вместо этого игрок A должен дождаться, пока все игроки — B, C и D — будут готовы обработать эту команду. Возникает парадоксальная ситуация: если экземпляр игры игрока A будет ждать слишком долго, прежде чем начать выполнение команды, будет создаваться впечатление, что игра «тормозит».

Проблема была решена за счет введения в очередь команд *таймера хода* (turn timer). В этом случае сначала выбирается интервал срабатывания таймера — в игре «Age of Empires» этот интервал по умолчанию равен 200 мс. В течение этих 200 мс все команды сохраняются в буфер. По истечении интервала 200 мс все команды игрока передаются по сети всем другим игрокам. Другой ключевой особенностью такой системы является задержка хода на два интервала. Это означает, например, что команды, запущенные игроком в ходе с порядковым номером 50, не будут выполнены ни одним экземпляром, пока не придет время хода с номером 52. Для случая с 200-миллисекундным таймером хода это означает, что *задержка реакции* (input lag) на команду игрока составит 600 мс. Задержка на два хода позволяет всем другим игрокам принять и подтвердить команды, которые должны быть выполнены на определенном ходе. Несмотря на то что понятие ходов кажется противоречащим сути стратегий реального времени, признаки этого технического решения можно заметить в самых разных стратегиях, включая «StarCraft II». Конечно, современные игры могут позволить себе роскошь сократить интервалы срабатывания таймеров ходов, потому что качество подключения к сети у большинства пользователей ныне намного выше, чем в конце 1990-х.

С решением, реализованным на основе таймера ходов, связано еще одно важное обстоятельство, которое необходимо учитывать. Представьте, что у одного из игроков произошла длительная задержка и он оказался не в состоянии действовать синхронно с 200-миллисекундным таймером. Игра могла бы приостановиться на время, чтобы ликвидировать отставание, — в конечном счете она могла бы «выбросить» игрока, если он продолжит задерживать остальных. «Age of Empires» также пытается компенсировать подобный сценарий, динамически меняя скорость

отображения в зависимости от состояния сети, — то есть компьютер с особенно медленным подключением к Интернету может выделить больше времени на прием данных из сети и меньше — на отображение графики. Более полную информацию о динамической регулировке таймера ходов можно найти в оригинальной статье Беттнера и Террано, ссылка на которую приводится в конце главы.

Решение, основанное на передаче команд, имеет еще одно преимущество: оно не требует большого объема памяти и вычислительной мощности для сохранения команд, запускаемых в ходе игры. Это дает возможность реализовать сохранение матча с последующим его воспроизведением, как это сделано в «Age of Empires II». Воспроизведение — очень популярная функция в стратегиях, она дает возможность оценить ход игры и получить более полное представление о возможных стратегиях. Чтобы реализовать подобное сохранение и воспроизведение в решениях, основанных на синхронизации информации об объектах вместо синхронизации информации о командах, потребовалось бы значительно больше времени и вычислительных ресурсов.

Синхронизация

Одних только таймеров ходов недостаточно, чтобы гарантировать синхронизацию между узлами. Поскольку каждый компьютер принимает и обрабатывает команды независимо, крайне важно, чтобы каждый из участников получал один и тот же результат. В своей статье Беттнер и Террано пишут: «Сложность обнаружения ошибок рассинхронизации связана с накоплением очень тонких отличий с течением времени. Олень может чуть-чуть сместиться, если по случайности корм для него окажется немного в другом месте, — и уже через минуту охотник может пойти чуть другим путем или его стрела пролетит мимо цели и он вернется домой без мяса».

Рассмотрим конкретный пример, вытекающий из того факта, что большинство игр вносят элемент случайности в игровую ситуацию. Представьте, что игра выполняет случайные проверки, чтобы определить, может ли лучник выстрелить по пехоте. Легко допустить, что экземпляр игрока A определил, что пехота находится в досягаемости лучника, а экземпляр игрока B — что она еще слишком далеко. Решение этой проблемы заключается в приставке «псевдо» в названии *генератор псевдослучайных чисел* (Pseudorandom Number Generator, PRNG). Поскольку все генераторы псевдослучайных чисел используют некоторое начальное число, можно гарантировать, что оба игрока — A и B — получат одну и ту же последовательность псевдослучайных чисел, если синхронизировать начальное значение генератора между экземплярами игры. Но имейте в виду, что синхронизация начального значения генератора гарантирует лишь получение одной и той же последовательности чисел. Поэтому важно гарантировать не только чтобы каждый экземпляр использовал одно и то же начальное значение, но и чтобы каждый экземпляр выполнил одинаковое число обращений к генератору псевдослучайных чисел, иначе генераторы в разных экземплярах рассинхронизируются. Проблема синхронизации PRNG в конфигурации «точка-точка» подробно рассматривается в главе 6 «Топологии сетей и примеры игр».

Проверка синхронизации дает еще одно неявное преимущество — она уменьшает возможность мошенничества. Например, если один игрок присвоит себе (мошен-

ническим путем, естественно) 500 дополнительных ресурсов, другие экземпляры немедленно обнаружат рассинхронизацию в состоянии игры. В этом случае легко можно реализовать исключение мошенника из игры. Однако, как и любая другая система, синхронизация позволяет выявить не все факты мошенничества. То обстоятельство, что каждый экземпляр моделирует каждый игровой объект, дает возможность реализовать мошеннический доступ к информации, которая не должна отображаться. То есть так называемый *взлом карты*, раскрывающий всю карту игры, все еще остается типичной проблемой в большинстве игровых стратегий. Этот и другие вопросы безопасности обсуждаются в главе 10 «Безопасность».

В заключение

Сетевые многопользовательские игры имеют длинную историю. Они начинались как игры, выполняющиеся в сетях больших ЭВМ, например «Empire» (1973), которая работала в сети PLATO. Позднее появились текстовые многопользовательские миры. Эти многопользовательские миры затем распространились на электронные доски объявлений, позволяющие подключаться к ним по коммутируемым телефонным линиям.

В начале 1990-х мир компьютерных игр штурмом взяли сетевые игры во главе с «Doom» (1993). Эти игры позволяли соединять несколько компьютеров в локальной сети и играть за или против друг друга. С развитием Интернета, в конце 1990-х большую популярность завоевали онлайн-игры, такие как «Unreal» (1998). В начале 2000-х онлайн-игры также стали появляться на игровых консолях. Современные онлайн-игры поддерживают участие в одном игровом сеансе сотен, если не тысяч игроков.

«Starsiege: Tribes» (1998) реализует сетевую архитектуру, которая все еще остается актуальной и используется в современных сюжетных играх. В ней используется клиент-серверная модель, когда каждый игрок подключается к серверу, осуществляющему координацию игры. На самом нижнем уровне находится модуль обслуживания пакетов, абстрагирующий операцию передачи пакетов в сеть. Выше располагается диспетчер соединений, обслуживающий соединения с игроками и обеспечивающий передачу уведомлений о доставке пакетов вышележащим уровням. Диспетчер потоков принимает данные от диспетчеров более высокого уровня (включая диспетчеры событий, фантомов и перемещений) и, опираясь на систему приоритетов, добавляет эти данные в исходящие пакеты. Диспетчер событий принимает важные события, такие как «игрок выстрелил», и гарантирует рассылку этих данных всем заинтересованным сторонам. Диспетчер фантомов обеспечивает отправку изменений в объектах, релевантных, по его мнению, для конкретного игрока. Диспетчер перемещений рассылает всем игрокам актуальную информацию о происходящих перемещениях.

«Age of Empires» (1997) реализует модель детерминированного соответствия. Все компьютеры, участвующие в игре, напрямую соединяются друг с другом. Однако вместо информации о каждом игровом объекте игра рассылает всем узлам команды. Эти команды затем обрабатываются узлами независимо друг от друга. Для синхронизации компьютеров используется таймер ходов — перед отправкой команд

в сеть они сначала накапливаются в буфере в течение определенного периода. Эти команды не выполняются до момента, пока не пройдет два периода таймера, что дает достаточно времени, чтобы каждый узел разослал и принял команды, которые подлежит выполнить во время хода. Кроме того, важно, чтобы каждый узел выполнял модель игры детерминированным образом, вследствие чего, например, возникает необходимость синхронизации генераторов псевдослучайных чисел.

Вопросы для повторения

1. Чем отличаются локальные и сетевые многопользовательские игры?
2. Назовите три разновидности соединений в локальных сетях.
3. Какая основная проблема возникает при адаптации для Интернета сетевой игры, выполнявшейся в локальной сети?
4. Расшифруйте аббревиатуру МПМ (MUD) и перечислите, в какие игры это развилось.
5. Чем MMO отличается от стандартной онлайн-игры?
6. Какие системы в модели «Tribes» обеспечивают надежность?
7. Опишите, как диспетчер фантомов в модели «Tribes» восстанавливает минимально необходимую повторную передачу при потере пакета.
8. В «Age of Empires» используется модель «точка-точка». Какую роль играет в этой модели таймер ходов? Какая информация передается по сети между узлами?

Для дополнительного чтения

Bettner, Paul and Mark Terrano. «1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond». Статья, представленная на конференции «Game Developer's Conference», Сан-Франциско, Калифорния, 2001.

Frohnmayer, Mark and Tim Gift. «The Tribes Engine Networking Model». Статья, представленная на конференции «Game Developer's Conference», Сан-Франциско, Калифорния, 2001.

Koster, Raph. «Online World Timeline». Веб-сайт «Raph Koster's Website». Последнее изменение: 20 февраля 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». В сборнике «Cyberspace: First Steps», edited by Michael Benedikt, с. 273–301. Cambridge: MIT Press, 1991.

Wasserman, Ken and Tim Stryker. «Multimachine Games». *Byte Magazine*, декабрь 1980, с. 24–40.

2 Интернет

В этой главе представлен обзор протоколов семейства TCP/IP и связанных с ними, а также стандартов, имеющих отношение к взаимодействиям в Интернете. Основное внимание уделяется тем из них, которые имеют отношение к программированию многопользовательских игр.

Происхождение: коммутация пакетов

Современный Интернет далеко ушел от сети с четырьмя узлами, с которой он начинался в конце 1969 года. Оригинальное название этой сети — ARPANET. Она была создана в США, в Управлении по перспективным научным исследованиям (Advanced Research Projects Agency) с целью обеспечить ученым, работающим в разных географических точках, доступ к мощным компьютерам, также географически удаленным друг от друга.

Поставленная цель достигалась в ARPANET применением недавно изобретенной технологии, получившей название *коммутация пакетов* (packet switching). До появления технологии коммутации пакетов удаленные системы передавали информацию, используя процедуру, известную как *коммутация линий* (circuit switching). Системы, использующие коммутацию линий, передавали информацию через согласованный канал, создаваемый путем соединения отдельных линий в более длинную и сохранявшийся на протяжении всего периода передачи. Например, чтобы отправить довольно большой объем данных, скажем телефонный звонок, из Нью-Йорка в Лос-Анджелес система коммутации линий должна была выделить несколько небольших линий между промежуточными городами, соединить их в одну и передать по ней информацию. В результате образовывался протяженный канал, сохранявшийся до завершения передачи информации. В данном случае могли быть зарезервированы линии от Нью-Йорка до Чикаго, от Чикаго до Денвера и от Денвера до Лос-Анджелеса. В свою очередь, эти линии в действительности состояли из еще более коротких линий между более близкими городами. Линии оставались

выделенными до окончания передачи информации, то есть пока не завершится телефонный звонок. После этого система могла задействовать линии для других попыток передачи информации. Это решение обеспечивало высокое качество связи, но ограничивало удобство использования, так как в каждый конкретный момент времени выделенные линии могли использоваться только для одной цели, как показано на рис. 2.1.

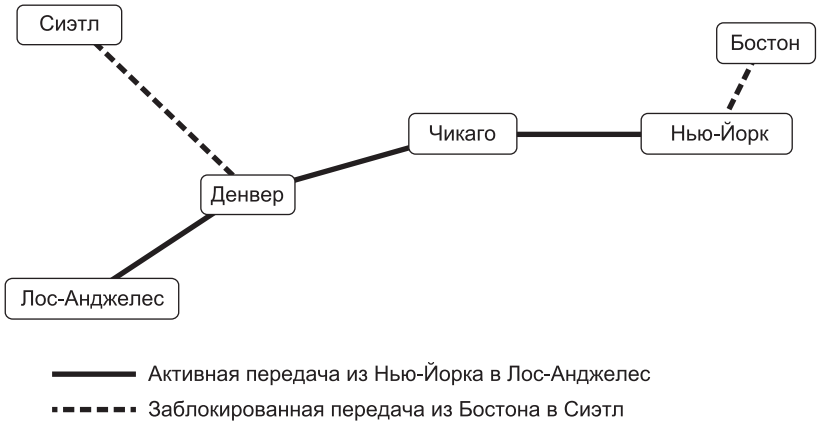


Рис. 2.1. Коммутация линий

Технология коммутации пактов, обеспечивая удобство использования, устраняет необходимость резервирования линий для выделенной передачи. Это достигается путем деления передачи на маленькие фрагменты, которые называются *пакетами*, и отправки их по совместно используемым линиям с применением процедуры, получившей название *передача с промежуточным накоплением* (store and forward). Каждый узел сети связан с другими узлами посредством линии, которая может передавать пакеты между узлами. Каждый узел накапливает входящие пакеты и передает их дальше, ближайшему узлу в заданном направлении. Например, информация, подлежащая передаче из Нью-Йорка в Лос-Анджелес, разбивается на множество очень коротких пакетов, которые затем отправляются из Нью-Йорка в Чикаго. Когда узел в Чикаго принимает пакет, он проверяет его адрес назначения и отправляет в Денвер. Процедура повторяется до тех пор, пока пакет не достигнет Лос-Анджелеса, а затем — телефона получателя. Важное отличие от технологии коммутации линий заключается в возможности ведения сразу нескольких телефонных разговоров с использованием одних и тех же линий. Одновременно по этим же линиям могут передаваться другие звонки из Нью-Йорка в Лос-Анджелес, из Бостона в Сиэтл или между любыми другими городами, расположенными «по дороге». По линиям можно одновременно передавать пакеты между множеством разных пунктов, что намного удобнее, как показано на рис. 2.2.

Однако коммутация пакетов — это всего лишь идея. Узлам в сети нужен набор формальных протоколов, определяющих, как в действительности должны упаковываться данные и пересылаться по сети. Для сети ARPANET этот набор прото-

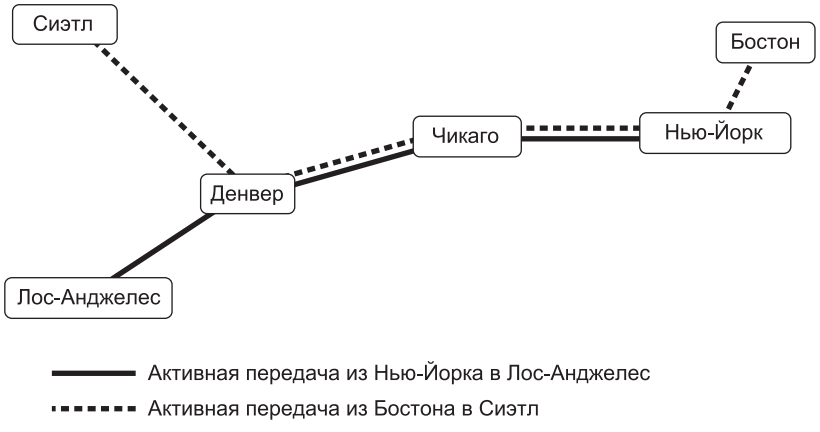


Рис. 2.2. Коммутация пакетов

колов был определен в документе, известном как «BBN Report 1822» и часто называемом *протоколом 1822*. С годами ARPANET продолжала разрастаться и стала частью сети, ныне известной как *Интернет*. В течение этого времени непрерывно развивались и протоколы из отчета 1822, превратившись в итоге в те протоколы, которые управляют работой современного Интернета. Все вместе они образуют набор протоколов, известный как *семейство TCP/IP*.

Многоуровневая модель TCP/IP

Семейство TCP/IP одновременно поражает своей красотой и пугает. Его красота обусловлена тем, что в теории это семейство состоит из нескольких независимых абстрактных уровней, каждый из которых поддерживается множеством взаимозаменяемых протоколов, решающих соответствующие задачи и передающих данные. А страх вызывают частые скандальные нарушения правил самими авторами протоколов в угоду производительности, масштабируемости или другим требованиям, которые объясняются их высокой сложностью.

Наша с вами задача как разработчиков многопользовательских игр — понять достоинства и недостатки семейства TCP/IP. Это поможет нам научиться писать функциональные и эффективные игры. Обычно достаточно уметь взаимодействовать с самыми верхними уровнями стека, но для достижения большей эффективности этих взаимодействий полезно знать и понимать, как функционируют нижние уровни и какое влияние они оказывают на верхние.

Существует множество моделей, описывающих взаимодействия между уровнями при обмене информацией в Интернете. В документе «RFC 1122», определившем требования к узлам Интернета на ранних этапах его развития, используется четыре уровня: *уровень сетевого доступа* (link layer), *межсетевой уровень* (IP layer), *транспортный уровень* (transport layer) и *прикладной уровень* (application layer). Альтернативная *модель взаимодействий открытых систем* (Open Systems Interconnection, OSI) определяет семь уровней: *физический уровень* (physical layer),

канальный уровень (data link layer), *сетевой уровень* (network layer), *транспортный уровень* (transport layer), *сеансовый уровень* (session layer), *представительский уровень* (presentation layer) и *прикладной уровень* (application layer). Чтобы оставить в центре внимания только то, что необходимо разработчикам игр, в этой книге используется комбинированная, пятиуровневая модель, включающая физический, канальный, сетевой, транспортный и прикладной уровни, как показано на рис. 2.3. Каждый уровень решает определенные задачи и удовлетворяет потребностям уровня, расположенного непосредственно над ним. В число этих задач входит:

- прием блока данных для передачи от вышележащего уровня;
- упаковка данных с добавлением заголовка (header) и, иногда, колонтитула (footer) данного уровня;
- передача данных нижележащему уровню для дальнейшей обработки;
- прием переданных данных от нижележащего уровня;
- распаковка данных с удалением заголовка;
- передача данных вышележащему уровню для дальнейшей обработки.

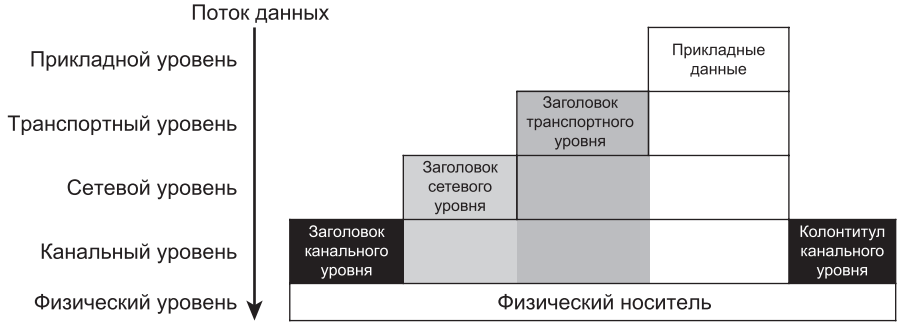


Рис. 2.3. Многоуровневая модель TCP/IP с точки зрения разработчика игр

Однако модель не определяет, как каждый уровень решает свои задачи. В реальности существует множество протоколов, которые можно использовать на каждом уровне. Некоторые из них являются ровесниками семейства TCP/IP, а другие появились совсем недавно. Знакомые с объектно-ориентированным программированием могут рассматривать каждый уровень как интерфейс, а каждый протокол или набор протоколов — как реализацию этого интерфейса. В идеале особенности реализации уровня не должны влиять на верхние уровни в стеке, но, как отмечалось выше, это не всегда так. В оставшейся части главы представлен обзор уровней и некоторых из наиболее известных протоколов, их реализующих.

Физический уровень

В самом низу многослойного пирога находится наиболее простой поддерживающий уровень: *физический*. Задача физического уровня — обеспечить физическую связь между компьютерами или узлами (хостами) в сети. Для передачи информации

необходим физический носитель. Витая пара категории 6, телефонные линии, коаксиальный кабель и волоконно-оптический кабель — все это примеры физических носителей, обеспечивающих соединения, необходимые физическому уровню.

Обратите внимание, что физический носитель не обязательно должен быть чем-то материальным. Любой обладатель мобильного телефона, планшета или ноутбука может заметить, что радиоволны также являются превосходным физическим носителем информации. В скором будущем квантовая запутанность также может стать физическим носителем для передачи информации на гигантские расстояния с огромной скоростью, и когда это произойдет, многослойный пирог Интернета будет готов использовать его в качестве реализации физического уровня.

Канальный уровень

Канальный уровень — это то место, откуда берет начало информатика многослойного пирога. Его задача — реализовать способ взаимодействий между физически связанными узлами. То есть канальный уровень должен предоставлять метод, посредством которого один узел может упаковать информацию и передать ее на физическом уровне, так чтобы узел-получатель мог, приняв пакет, извлечь из него переданную информацию.

Единица передачи на канальном уровне называется *кадром* (frame). Используя канальный уровень, узлы посылают друг другу кадры. Если говорить точнее, в число задач канального уровня входит:

- определение способа идентификации узлов, чтобы кадр мог достичь адресата;
- определение формата кадра, включающего адрес получателя и отправляемые данные;
- определение максимального размера кадра, чтобы верхние уровни знали, какой объем данных можно отправить в одной передаче;
- определение способа физического преобразования кадра в электрические сигналы, которые можно передать через физический носитель и, при благоприятном стечении обстоятельств, принять на стороне получателя.

Обратите внимание, что доставка кадра узлу назначения лишь возможна, но не гарантирована. Существует множество факторов, влияющих на целостность электрического сигнала, фактически достигающего места назначения. Повреждение физического носителя, разного рода электромагнитные помехи или отказ оборудования могут привести к потере кадра. Канальный уровень не предполагает приложения каких-либо дополнительных усилий, чтобы определить факт доставки кадра адресату или повторную его передачу, если доставка не увенчалась успехом. По этой причине взаимодействия на канальном уровне называют ненадежными. Любой высокоуровневый протокол, гарантирующий надежную доставку данных, должен сам реализовать эти гарантии.

Для каждой разновидности физического носителя, выбранного для реализации физического уровня, имеется соответствующий протокол или группа протоколов, обеспечивающих все необходимые функции на канальном уровне. Например, узлы,

соединенные витой парой, могут взаимодействовать посредством одного из протоколов Ethernet, такого как 1000BASET. Узлы, соединенные радиоканалом, могут взаимодействовать посредством одного из радиопротоколов Wi-Fi (например, 802.11g, 802.11n, 802.11ac) или одного из радиопротоколов мобильной связи, таких как 3G или 4G. В табл. 2.1 перечислены некоторые распространенные комбинации физических носителей и протоколов канального уровня.

Таблица 2.1. Комбинации физических носителей и протоколов канального уровня

Физический носитель	Протокол канального уровня
Витая пара	Ethernet 10BASET, Ethernet 100BASET, Ethernet 1000BASET
Витой медный провод	Ethernet over Copper (EoC)
Радиодиапазон 2.4 ГГц	802.11b, 802.11g, 802.11n
Радиодиапазон 5 ГГц	802.11n, 802.11ac
Радиодиапазон 850 МГц	3G, 4G
Волоконно-оптический кабель	Fiber Distributed Data Interface (FDDI), Ethernet 10GBASESR, Ethernet 10GBASELR
Коаксиальный кабель	Ethernet over Coax (EoC), Data Over Cable Service Interface Specification (DOCSIS)

Из-за столь тесной связи физического и канального уровней в некоторых моделях они объединяются в один уровень. Однако некоторые физические носители поддерживаются более чем одним протоколом канального уровня, поэтому предпочтительнее рассматривать физический и канальный уровни отдельно.

Важно отметить, что интернет-соединение между двумя удаленными узлами не требует использования единого физического носителя и единого протокола канального уровня. Как будет рассказано в следующих разделах, описывающих остальные уровни, в передачу одного фрагмента данных может быть вовлечено несколько носителей и протоколов канального уровня. Соответственно, в передачу данных для сетевой компьютерной игры также может быть вовлечено множество протоколов канального уровня, перечисленных в табл. 2.1. К счастью, благодаря абстракции семейства TCP/IP особенности работы используемого протокола канального уровня оказываются надежно скрытыми от игры. По этой причине мы не будем углубляться в детальное исследование функционирования каждого из имеющихся протоколов канального уровня. Однако существует одна группа протоколов, которые, с одной стороны, ярко иллюстрируют работу канального уровня, а с другой — обязательно встретятся на профессиональном пути каждого программиста сетевых игр. Это протоколы *Ethernet*.

Ethernet/802.3

Ethernet — это не какой-то один протокол, а целая группа протоколов, документированных в официальном сборнике стандартов Ethernet, опубликованном в 1980 году компаниями DEC, Intel и Xerox. Все вместе современные протоколы

Ethernet определяются стандартом IEEE 802.3. Существуют варианты Ethernet, действующие в оптоволоконных линиях, витых парах или многожильных витых медных проводах. Есть варианты, работающие с разными скоростями: на момент написания этих строк большинство настольных компьютеров поддерживали гигабитный Ethernet, но уже существовали и набирали популярность стандарты Ethernet, поддерживающие скорости передачи до 10 Гбит/с.

Для идентификации узлов в Ethernet используются аппаратные адреса (адреса управления доступом к среде — Media Access Control), или *MAC-адреса*. MAC-адрес — это уникальное 48-битное число, присваиваемое каждому аппаратному устройству, подключаемому к сети Ethernet. Обычно это устройство называют *контроллером сетевого интерфейса* (Network Interface Controller, NIC). Первоначально сетевые контроллеры имели вид платы расширения, но благодаря бурному распространению Интернета в последние пару десятилетий они стали встраиваться в материнские платы. Если узлу требуется более одного соединения с одной или несколькими сетями, существует возможность вставить в компьютер дополнительные платы сетевых контроллеров, и такой узел будет иметь несколько MAC-адресов, по одному для каждого сетевого контроллера.

Чтобы обеспечить глобальную уникальность MAC-адресов, производители сетевых контроллеров «зашивают» MAC-адреса в контроллеры на аппаратном уровне. Первые 24 бита составляют уникальный идентификатор организации (Organizationally Unique Identifier, OUI), выданный IEEE для идентификации производителя. Уникальность оставшихся 24 бит, присваиваемых аппаратным устройствам, должен гарантировать производитель. Из вышесказанного следует, что каждый сетевой контроллер должен иметь аппаратно зашитый глобально-уникальный идентификатор, по которому к этому контроллеру можно обратиться.

MAC-адрес оказался настолько удачной идеей, что используется не только протоколами Ethernet. В действительности он используется большинством протоколов канального уровня IEEE 802, включая Wi-Fi и Bluetooth.

ПРИМЕЧАНИЕ С момента появления MAC-адресов они претерпели два важных изменения. Во-первых, MAC-адрес перестал быть по-настоящему уникальным аппаратным идентификатором, потому что многие современные сетевые контроллеры позволяют произвольно изменять их MAC-адреса. Во-вторых, для решения различных назревших проблем организация IEEE ввела понятие 64-битного MAC-адреса, получившего название расширенного уникального идентификатора (Extended Unique Identifier, EUI64). Везде, где необходимо, 48-битный MAC-адрес может быть преобразован в EUI64 вставкой двух байт `0xFF 0xFE` сразу после OUI (уникального идентификатора организации).

На рис. 2.4 изображена структура Ethernet-пакета, который обертывает Ethernet-кадр канального уровня и добавляет уникальный MAC-адрес, присвоенный каждому сетевому узлу.

Преамбула и *начальный разделитель кадра* (Start Frame Delimiter, SFD) одинаковые для каждого пакета и состоят из восьми байтов: `0x55 0x55 0x55 0x55 0x55 0x55 0x55 0xD5`. Это двоичный шаблон, помогающий аппаратуре сетевого контроллера синхронизироваться и подготовиться к приему кадра. Преамбула и начальный разделитель обычно выбрасываются из пакета аппаратурой сетевого контроллера,

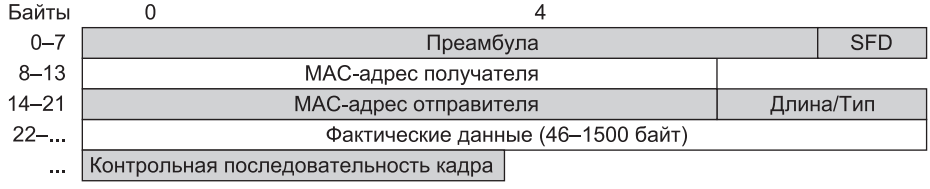


Рис. 2.4. Структура Ethernet-пакета

а оставшиеся байты, включая кадр, передаются модулю реализации протокола Ethernet для обработки.

За начальным разделителем кадра следуют шесть байтов, представляющих MAC-адрес получателя кадра. Существует специальный MAC-адрес получателя, `FF:FF:FF:FF:FF:FF`, известный как *широковещательный адрес*, указывающий, что кадр предназначен всем узлам в локальной сети.

Поле «длина/тип» имеет несколько назначений и может представлять длину или тип. Если поле используется для представления длины, оно хранит объем фактических данных кадра в байтах. Если поле используется для представления типа, оно хранит номер *EtherType*, однозначно определяющий протокол, который должен использоваться для интерпретации фактических данных. Модуль Ethernet, принимающий это поле, должен определить, как правильно его интерпретировать. Чтобы помочь в этом, стандарт Ethernet определяет, что максимальный объем фактических данных не может превышать 1500 байт. Это число известно как *максимальный размер пакета* (Maximum Transmission Unit, MTU), потому что это максимальный объем данных, который можно отправить в одной передаче. Стандарт также определяет минимальное значение EtherType как `0x0600` (в десятичном представлении — 1536). То есть если поле «длина/тип» содержит число, меньшее или равное 1500, оно представляет длину, а если большее или равное 1536 — тип.

ПРИМЕЧАНИЕ Хотя это является нарушением стандарта, многие современные сетевые контроллеры Ethernet поддерживают кадры, длина которых превышает 1500 байт. Эти *jumbo-кадры* (jumbo frames) могут иметь размер до 9000 байт. Для поддержки этой возможности контроллеры определяют в заголовке кадра EtherType, а вычисление размера кадра происходит аппаратно.

Фактические данные — это данные, подлежащие передаче в кадре. Обычно это пакет сетевого уровня, переданный канальному уровню для доставки соответствующему узлу.

Поле *контрольной последовательности кадра* (Frame Check Sequence, FCS) хранит значение контрольной суммы CRC32, сгенерированное на основе двух полей адресов, поля длины/типа, фактических данных и дополнения. Благодаря этому после приема данных аппаратура Ethernet может проверить целостность кадра и отбросить его, если будет установлен факт его повреждения. Несмотря на то что протокол Ethernet не гарантирует доставку данных, он прилагает большие усилия, чтобы предотвратить доставку поврежденных данных.

Специфика способа передачи пакетов Ethernet через физический уровень зависит от носителя и не представляет интереса для разработчиков многопользовательских

игр. Достаточно будет сказать, что каждый узел в сети принимает кадр, читает его и определяет, является ли он его получателем. Если является, то из кадра будут извлечены фактические данные и обработаны в соответствии со значением поля длины/типа.

ПРИМЕЧАНИЕ На ранних этапах в большинстве небольших сетей Ethernet для соединения множества узлов использовались устройства, известные как *концентраторы* (hubs). Еще раньше использовался длинный коаксиальный кабель, протянутый между компьютерами. В таких сетях электрические сигналы, соответствующие Ethernet-пакету, достигали буквально каждого узла, и уже сам узел определял, является ли он получателем пакета. Такое техническое решение оказалось неэффективным для больших сетей. По мере снижения стоимости аппаратуры в большинстве современных сетей для соединения узлов стали использоваться устройства, известные как *сетевые коммутаторы* (switches). Коммутаторы запоминают MAC-адреса, а иногда и IP-адреса узлов, подключенных к портам, поэтому большинство пакетов может посылаться по кратчайшим путям к их получателям и отпадает необходимость передавать их каждому узлу в сети.

Сетевой уровень

Канальный уровень обеспечивает простой и однозначный способ передачи данных одному или нескольким адресуемым узлам. Поэтому может показаться странным, зачем нужны все остальные уровни в семействе TCP/IP. Как оказывается, канальный уровень имеет несколько недостатков, которые могут быть устранены на уровнях, лежащих выше:

- Адресация с использованием аппаратного MAC-адреса ограничивает гибкость. Вообразите, что имеется очень популярный веб-сервер, который ежедневно посещают тысячи пользователей. Если бы доступ к веб-серверу был возможен только посредством канального уровня, запросы должны были бы направляться по MAC-адресу его сетевого контроллера. А теперь представьте, что однажды сетевой контроллер вышел из строя. Новый сетевой контроллер, установленный взамен сгоревшего, будет иметь другой MAC-адрес, и сервер уже не сможет принимать запросы от пользователей. Очевидно, что поверх MAC-адресов должна иметься какая-то другая, легко настраиваемая система адресации.
- Канальный уровень не поддерживает сегментацию Интернета на более мелкие, локальные сети. Если бы весь Интернет использовал только канальный уровень, все компьютеры должны были бы быть подключены к единой протяженной сети. Вспомните, что протокол Ethernet доставляет каждый кадр каждому узлу в сети и позволяет узлу самому определить, является ли он получателем. Если бы в Интернете использовался только протокол Ethernet, каждый кадр должен был бы достичь каждого узла на планете, подключенного к сети. В этом случае очень небольшое число пакетов поставило бы Интернет на колени. Кроме того, не было бы никакой возможности определять свою политику безопасности в разных областях сети. Иногда желательно, чтобы широковещательные сообщения передавались только узлам в местном офисе,

или доступ к общим файлам был возможен только для компьютеров в одном здании. С использованием только канального уровня введение подобных ограничений было бы невозможно.

- Канальный уровень не поддерживает возможность взаимодействий узлов с использованием разных протоколов канального уровня. Главной идеей поддержки множества протоколов физического и канального уровня является возможность использования оптимальных реализаций для разных сетей. Однако протоколы канального уровня не определяют способы взаимодействий между разными протоколами. И снова возникает необходимость в системе адресации, расположенной над системой аппаратных адресов канального уровня.

Задача канального уровня — обеспечить инфраструктуру для логической адресации, чтобы можно было легко менять аппаратные компоненты узла, группировать узлы в отдельные подсети и позволять узлам в удаленных подсетях, использующих разные физические носители и протоколы канального уровня, передавать сообщения друг другу.

IPv4

В настоящее время для реализации необходимых функций сетевого уровня наиболее широко используется *интернет-протокол версии 4* (Internet protocol version 4, IPv4). IPv4 решает поставленные задачи, определяя:

- систему логической адресации, позволяющую присваивать сетевым узлам имена;
- систему логического разделения адресного пространства в физических подсетях;
- систему маршрутизации для передачи данных между подсетями.

IP-адреса и структура пакета

Основой IPv4 являются IP-адреса. IP-адрес в IPv4 — это 32-битное число, обычно отображаемое для удобочитаемости в виде четырех 8-битных чисел, разделенных точками. Например, сервер *www.usc.edu* имеет IP-адрес 128.125.253.146, а *www.mit.edu* — IP-адрес 23.193.142.184. Благодаря уникальности IP-адресов в Интернете узел-отправитель может напрямую послать пакет узлу-получателю, просто указав IP-адрес получателя в заголовке пакета. Существует исключение из правила уникальности IP-адресов, описываемое ниже в разделе «NAT».

Вместе с IP-адресами протокол IPv4 определяет структуру пакетов IPv4. Пакет состоит из заголовка с информацией, необходимой для поддержки функций сетевого уровня, и фактических данных, содержащих информацию для передачи, полученную от уровня выше. Структура пакета IPv4 приводится на рис. 2.5.

Версия (4 бита) определяет версию протокола, поддерживающую данный пакет. Для IPv4 имеет значение 4.

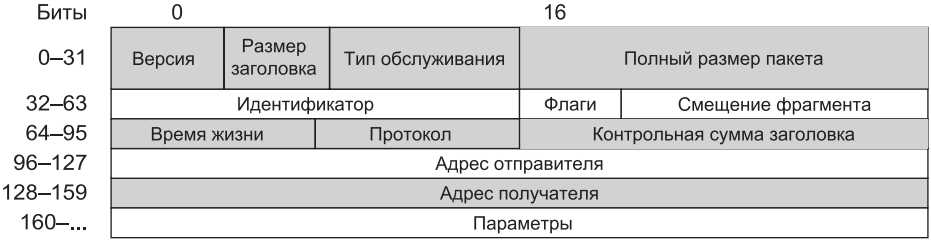


Рис. 2.5. Структура IPv4-пакета

Размер заголовка (4 бита) определяет размер заголовка в 32-битных словах. Из-за дополнительных полей в конце IP-заголовок может иметь переменный размер. Поле размера точно определяет, где заканчивается заголовок и начинаются фактические данные. Так как поле имеет размер всего 4 бита, его значение не может превышать 15, то есть заголовок не может иметь размер больше пятнадцати 32-битных слов, или 60 байт. Учитывая, что объем обязательной информации в заголовке составляет 20 байт, это поле не может иметь значение меньше 5.

Тип обслуживания (8 бит) используется для разных нужд, от контроля нагруженности до идентификации разных служб. За дополнительной информацией обращайтесь к RFC 2474 и RFC 3168, ссылки на которые приводятся в разделе «Для дополнительного чтения».

Полный размер пакета (16 бит) определяет размер всего пакета в байтах, включая заголовок и фактические данные. Так как максимальное число, которое можно представить 16 битами, равно 65 535, максимальный размер пакета ограничен 65 535 байтами. Учитывая, что минимальный размер IP-заголовка составляет 20 байт, максимальный объем фактических данных в пакете IPv4 равен 65 515 байтам.

Идентификатор фрагмента (16 бит), **флаги фрагмента** (3 бита) и **смещение фрагмента** (13 бит) используются для сборки фрагментированных пакетов, как описывается в разделе «Фрагментация».

Время жизни (Time to Live, TTL, 8 бит) используется для ограничения числа пересылок пакета, как описывается далее в разделе «Подсети и косвенная маршрутизация».

Протокол (8 бит) определяет, какой протокол должен использоваться для интерпретации содержимого в области фактических данных. Он напоминает поле EtherType в кадре Ethernet тем, что классифицирует данные, упакованные уровнем выше.

Контрольная сумма заголовка (16 бит) содержит контрольную сумму для проверки целостности заголовка IPv4. Обратите внимание, что эта контрольная сумма относится только к заголовку. Проверка целостности фактических данных должна осуществляться уровнем выше. Часто в этом нет необходимости, потому что многие протоколы канального уровня уже проверяют контрольную сумму (например, поле контрольной последовательности кадра в заголовке Ethernet) и гарантируют целостность всего кадра.

Адрес отправителя (32 бита) — IP-адрес отправителя пакета, а **адрес получателя** (32 бита) — либо IP-адрес узла-получателя пакета, либо специальный адрес, соответствующий множеству узлов.

ПРИМЕЧАНИЕ Может показаться странным, почему размер заголовка определяется в 32-битных словах, а размер пакета — в 8-битных. Это обусловлено стремлением к экономии пропускной способности. Так как любые возможные заголовки имеют размер, кратный 4 байтам, число, определяющее размер, всегда будет кратно 4, то есть последние 2 бита в этом числе всегда будут равны 0. Определив размер заголовка в 32-битных словах, удается сэкономить 2 бита. Экономия пропускной способности — золотое правило в программировании многопользовательских игр.

Прямая маршрутизация и протокол разрешения адресов

Чтобы понять, как IPv4 позволяет пакетам перемещаться между сетями с разными протоколами канального уровня, необходимо сначала разобраться с тем, как осуществляется доставка пакетов в сети с единственным протоколом канального уровня. Протокол IPv4 обеспечивает адресацию пакетов посредством IP-адресов. Чтобы канальный уровень мог доставить пакет нужному адресату, он должен быть завернут в кадр с адресом, понятным канальному уровню. Рассмотрим, как узел А мог бы отправить данные узлу Б (рис. 2.6).

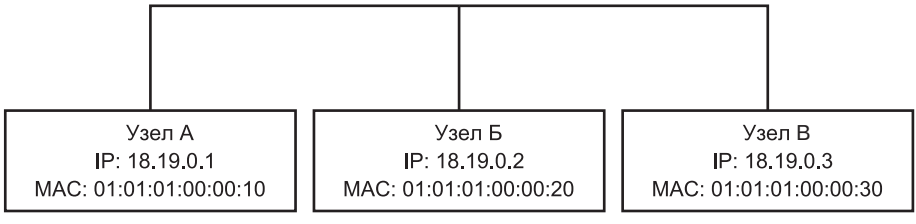


Рис. 2.6. Сеть с тремя узлами

На рис. 2.6 изображен пример сети с тремя узлами, подключенными к сети посредством Ethernet, каждый из которых имеет единственный сетевой контроллер. Узел А, имеющий адрес 18.19.0.1, хочет передать сетевому уровню пакет для узла Б, IP-адрес которого — 18.19.0.2. То есть узел А подготавливает пакет IPv4 с IP-адресом отправителя 18.19.0.1 и IP-адресом получателя 18.19.0.2. Теоретически сетевой уровень затем должен вручить пакет канальному уровню, чтобы тот выполнил фактическую доставку. К сожалению, модуль Ethernet не сможет доставить пакет, используя только IP-адрес, так как этот адрес понятен только сетевому уровню. Канальный уровень должен каким-то образом определить MAC-адрес, соответствующий IP-адресу 18.19.0.2. К счастью, существует протокол канального уровня, который называется *протоколом разрешения адресов* (Address Resolution Protocol, ARP) и который как раз и решает эту задачу.

ПРИМЕЧАНИЕ Технически ARP принадлежит канальному уровню, потому что посылает пакеты, используя адреса канального уровня, и не нуждается в маршрутизации между сетями, которая осуществляется на сетевом уровне. Однако из-за того, что протокол вторгается на территорию абстракции сетевого уровня, используя IP-адреса, его лучше рассматривать как мост между уровнями, а не как протокол канального уровня.

ARP состоит из двух основных частей: структуры пакета, используемого для запроса IP-адреса у сетевого контроллера по его MAC-адресу, и таблицы для хранения этих пар адресов. Пример ARP-таблицы приводится в табл. 2.2.

Таблица 2.2. ARP-таблица отображения IP-адресов в MAC-адреса

IP-адрес	**MAC-адрес**
18.19.0.1	01:01:01:00:00:10
18.19.0.3	01:01:01:00:00:30

Когда реализации протокола IP требуется послать пакет, используя канальный уровень, она должна сначала получить из ARP-таблицы MAC-адрес, связанный с IP-адресом получателя. Если искомый MAC-адрес присутствует в таблице, модуль IP сконструирует кадр канального уровня, используя полученный MAC-адрес, и передаст его реализации канального уровня для доставки. Если MAC-адрес отсутствует в таблице, модуль ARP попытается определить соответствующий MAC-адрес, послав ARP-пакет (рис. 2.7) всем узлам в сети, доступной канальному уровню.

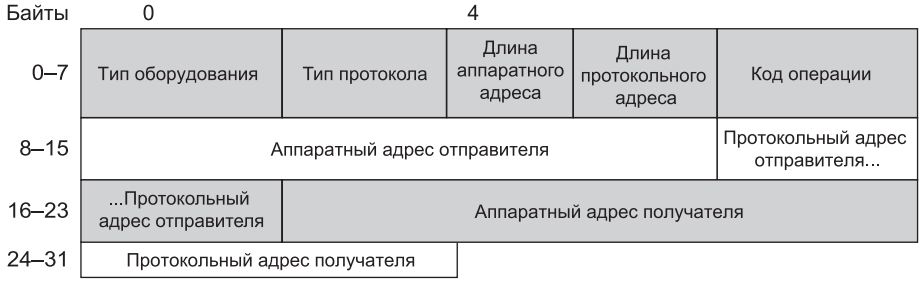


Рис. 2.7. Структура ARP-пакета

Тип оборудования (16 бит) определяет тип оборудования, реализующего канальный уровень. Для Ethernet тип оборудования равен 1.

Тип протокола (16 бит) совпадает со значением EtherType, используемым протоколом сетевого уровня. Например, протоколу IPv4 соответствует число `0x0800`.

Длина аппаратного адреса (8 бит) — длина аппаратного адреса канального уровня в байтах. Обычно соответствует длине MAC-адреса, равной 6 байтам.

Длина протокольного адреса (8 бит) — длина логического адреса, используемого на сетевом уровне. Для IPv4 длина IP-адреса равна 4 байтам.

Код операции (16 бит) — число 1 или 2, определяет назначение пакета — запрос на получение информации (1) или ответ (2).

Аппаратный адрес отправителя (переменной длины) — аппаратный адрес отправителя данного пакета, а **протокольный адрес отправителя** (переменной длины) — логический адрес отправителя пакета, используемый на сетевом уровне. Размеры этих адресов определяются значениями полей, описанных выше.

Аппаратный адрес получателя (переменной длины) и **протокольный адрес получателя** (переменной длины) — соответствующие адреса получателя пакета. В случае с запросом аппаратный адрес получателя неизвестен и игнорируется получателем.

Продолжим предыдущий пример: если узел А не знает MAC-адреса узла Б, он подготавливает пакет ARP-запроса с `1` в поле «Код операции», `18.19.0.1` в поле «Протокольный адрес отправителя», `01:01:01:00:00:10` в поле «Аппаратный адрес отправителя» и `18.19.0.2` в поле «Протокольный адрес получателя». Затем заворачивает этот ARP-пакет в кадр Ethernet и отправляет его по широковещательному адресу Ethernet `FF:FF:FF:FF:FF:FF`. Как мы уже говорили, этот адрес указывает, что кадр Ethernet должен быть доставлен всем узлам в сети и проанализирован ими.

Получив этот пакет, узел В не должен отвечать на него, потому что его IP-адрес не соответствует протокольному адресу получателя в пакете. А вот узел Б, IP-адрес которого совпадает с искомым, напротив, должен ответить ARP-пакетом, содержащим его собственный адрес в качестве отправителя и адрес узла А в качестве получателя. Приняв этот пакет, узел А добавит в ARP-таблицу MAC-адрес узла Б, затем завернет ожидающий IP-пакет в кадр Ethernet и отправит его узлу Б по его MAC-адресу.

ПРИМЕЧАНИЕ Когда узел А рассылает первоначальный ARP-запрос всем узлам в сети, он включает в него оба своих адреса — MAC и IP. Это дает возможность всем другим узлам в сети добавить информацию об узле А в свои ARP-таблицы, даже если в данный момент эта информация им не нужна. Она пригодится, когда потребуется вступить в связь с узлом А, и избавит от необходимости посылать ARP-пакет запроса.

Возможно, вы заметили, что эта система создает некую уязвимость в системе безопасности. Злонамеренный узел может посылать ARP-пакеты с любыми IP-адресами. Не имеющий возможности проверить аутентичность ARP-информации сетевой коммутатор мог бы непреднамеренно переправлять злонамеренному узлу пакеты, предназначенные другому узлу. Эта уязвимость позволяет не только подслушивать сетевой трафик, но и препятствовать доставке перехваченных пакетов истинному адресату, полностью нарушая работу сети.

Подсети и косвенная маршрутизация

Представьте две компании, назовем их Альфа и Бета. Каждая имеет большую компьютерную сеть: сеть Альфа и сеть Бета. Сеть Альфа содержит 100 узлов, от А1 до А100, и сеть Бета содержит 100 узлов, от Б1 до Б100. Обе компании хотели бы связать свои сети, чтобы иметь возможность обмениваться сообщениями, но простое соединение двух сетей кабелем Ethernet на канальном уровне влечет за собой пару проблем. Как вы помните, пакеты Ethernet должны передаваться всем узлам в сети. Соединение сетей Альфа и Бета на канальном уровне повлечет необходимость передачи каждого пакета всем 200 узлам вместо 100, что фактически приведет к удвоению трафика в объединенной сети. Кроме того, увеличивается угроза безопасности, так как из сети Альфа в сеть Бета будут передаваться все пакеты, а не только те, что действительно предназначены для узлов в сети Бета.

Чтобы компании Альфа и Бета могли связать свои сети без потери эффективности, на сетевом уровне предусмотрена возможность маршрутизации пакетов между узлами в сетях, не связанных непосредственно на канальном уровне. Фактически сам Интернет изначально задумывался как объединение таких небольших сетей, разбросанных по всей стране и связанных между собой протяженными соединениями. Дословное значение слова «интернет» — «междусеть», то есть соединение сетей. Обеспечение взаимодействий между сетями — это задача сетевого уровня. На рис. 2.8 показано соединение сетей Альфа и Бета на сетевом уровне.

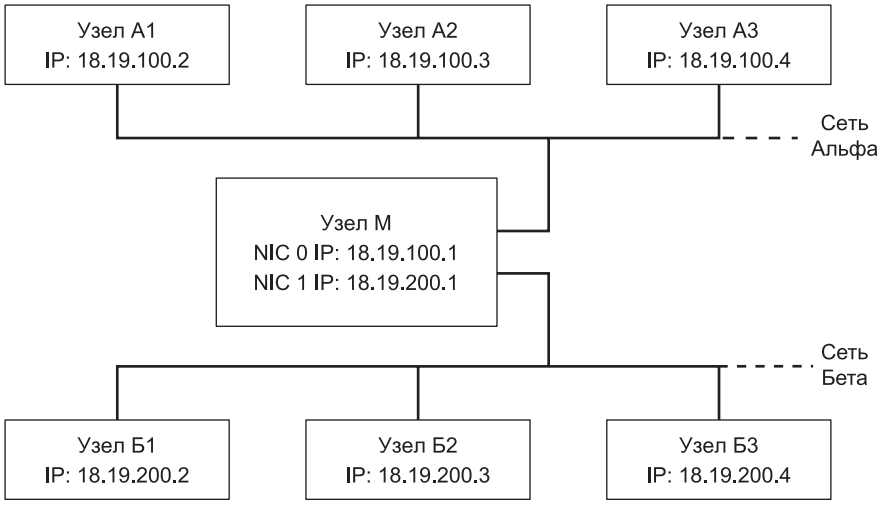


Рис. 2.8. Соединение сетей Альфа и Бета

Узел М — это специальный узел, который называется *маршрутизатором* (router). Маршрутизатор имеет несколько сетевых контроллеров, каждый со своим IP-адресом. В данном случае один контроллер подключен к сети Альфа, а другой — к сети Бета. Обратите внимание, что все IP-адреса в сети Альфа начинаются с `18.19.100`, а все IP-адреса в сети Бета — с `18.19.200`. Чтобы понять, как мы это можем использовать, необходимо детальнее исследовать подсети и разобраться с идеей *маски подсети*.

Маска подсети — это 32-битное число, обычно записываемое в виде четырех чисел, разделенных точками, по аналогии с IP-адресами. Говорят, что узлы принадлежат одной подсети, если поразрядная операция И (AND) между их IP-адресами с маской подсети возвращает один и тот же результат. Например, если подсеть определяется маской `255.255.255.0`, тогда IP-адреса `18.19.100.1` и `18.19.100.2` являются допустимыми IP-адресами этой подсети (табл. 2.3). Однако адрес `18.19.200.1` не принадлежит подсети, потому что операция поразрядного И (AND) между адресом с маской подсети вернет другой результат.

В двоичном представлении маска подсети обычно имеет вид последовательности единиц, за которой следует последовательность нулей, что делает ее более удобочитаемой и упрощает выполнение поразрядной операции И (AND) в уме.

В табл. 2.4 перечислены наиболее типичные маски подсетей и число уникальных адресов узлов в этих подсетях. Обратите внимание, что в любой подсети два адреса всегда зарезервированы и не используются для узлов. Один из них — *адрес сети*, который является результатом поразрядной операции И (AND) между маской подсети и любым IP-адресом в этой подсети, другой — *широковещательный адрес*, который является результатом поразрядной операции ИЛИ (OR) между адресом сети и поразрядным дополнением маски подсети. То есть каждый бит в адресе сети, не определяющий подсеть, должен быть установлен в 1. Пакеты, посылаемые по широковещательному адресу подсети, должны доставляться всем узлам в подсети.

Таблица 2.3. IP-адреса и маски подсетей

Узел	IP-адрес	Маска подсети	IP-адрес И маска подсети
А1	18.19.100.1	255.255.255.0	18.19.100.0
А2	18.19.100.2	255.255.255.0	18.19.100.0
Б1	18.19.200.1	255.255.255.0	18.19.200.0

Таблица 2.4. Примеры масок подсетей

Маска подсети	Маска подсети в двоичном представлении	Значащих битов	Максимальное число узлов
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

Так как подсеть, по определению, — это группа узлов с IP-адресами, для которых поразрядная операция И (AND) с маской подсети возвращает один и тот же результат, любая подсеть определяется маской подсети и ее адресом сети. Например, сеть Альфа как подсеть определяется сетевым адресом `18.19.100.0` с маской подсети `255.255.255.0`.

Широко используется сокращенная форма записи, известная как *бесклассовая адресация* (Classless Inter-Domain Routing, CIDR). В двоичном представлении маска подсети обычно имеет вид последовательности из *n* единиц, за которой идет последовательность из 32 – *n* нулей. Поэтому подсеть можно записать как сетевой адрес, символ `/` и число значащих битов в маске подсети. Например, сеть Альфа, изображенную на рис. 2.8, можно описать в нотации CIDR как `18.19.100.0/24`.

ПРИМЕЧАНИЕ Слово «бесклассовая» в словосочетании «бесклассовая адресация» объясняется тем, что обычно адресация и присваивание блоков адресов производятся на основе трех классов сетей. Сети класса A имеют маску подсети `255.0.0.0`, сети класса B — маску подсети `255.255.0.0`, сети класса C — маску подсети `255.255.255.0`. Более полную информацию о бесклассовой адресации (CIDR) можно найти в документе RFC 1518, упомянутом в разделе «Для дополнительного чтения».

Путем определения подсетей спецификация IPv4 позволяет организовать передачу пакетов между узлами в разных сетях. Делается это при помощи *таблицы маршрутизации* (routing table), присутствующей в модуле IP каждого узла. В частности, когда модуль IPv4 узла запрашивает отправку IP-пакета удаленному узлу, он должен выяснить, использовать ли таблицу ARP и выполнить отправку непосредственно или задействовать какой-то обходной маршрут. С этой целью в каждом модуле IPv4 имеется таблица маршрутизации. Для каждой достижимой подсети в таблице маршрутизации присутствует строка с информацией о том, как пакеты должны доставляться в эту подсеть. В табл. 2.5, 2.6 и 2.7 приводятся возможные варианты таблиц маршрутизации для узлов А1, Б1 и М в сети, изображенной на рис. 2.8.

Таблица 2.5. Таблица маршрутизации узла А1

№ п/п	Подсеть назначения	Шлюз	Сетевой контроллер
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)

Таблица 2.6. Таблица маршрутизации узла Б1

№ п/п	Подсеть назначения	Шлюз	Сетевой контроллер
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)

Таблица 2.7. Таблица маршрутизации узла М

№ п/п	Подсеть назначения	Шлюз	Сетевой контроллер
1	18.19.100.0/24		NIC 0 (18.19.100.1)
2	18.19.200.0/24		NIC 1 (18.19.200.1)

Колонка «Подсеть назначения» содержит информацию о подсети, содержащей целевой IP-адрес. Колонка «Шлюз» определяет IP-адрес следующего узла в текущей подсети, которому следует послать данный пакет через канальный уровень. Шлюз обязательно должен быть доступен непосредственно. Если поле «Шлюз» пустое, значит, вся подсеть назначения доступна непосредственно и пакет можно переслать целевому узлу через канальный уровень. Наконец, колонка «Сетевой контроллер» идентифицирует сетевой контроллер, который фактически отправит пакет. Это тот самый механизм, посредством которого можно принять пакет на канальном уровне одной сети и передать его на канальный уровень другой сети.

Когда узел А1 с адресом `18.19.100.2` пытается послать пакет узлу Б1 с адресом `18.19.200.2`, выполняется следующая последовательность действий:

1. Узел А1 конструирует IP-пакет с адресом отправителя `18.19.100.2` и адресом получателя `18.19.200.2`.
2. Модуль IP узла А1 просматривает строки в таблице маршрутизации сверху вниз, пока не найдет первую строку с подсетью назначения, содержащей IP-

адрес `18.19.200.2`. В нашем случае это строка 2. Обратите внимание, что порядок следования строк имеет важное значение, так как данному адресу может соответствовать несколько строк.

3. Шлюз, указанный в строке 2, имеет адрес `18.19.100.1`, поэтому узел А1 с помощью ARP и своего модуля Ethernet заворачивает пакет в кадр Ethernet и посылает его по MAC-адресу, соответствующему IP-адресу `18.19.100.1`. Так пакет достигает узла М.
4. Модуль Ethernet узла М, обслуживающий его сетевой контроллер NIC 0 с IP-адресом `18.19.100.1`, принимает пакет, определяет, что в области фактических данных находится IP-пакет, и передает его модулю IP.
5. Модуль IP узла М видит, что пакет адресован узлу с IP-адресом `18.19.200.1`, и поэтому пытается переслать этот пакет по указанному адресу.
6. Модуль IP узла М просматривает таблицу маршрутизации, пока не найдет строку с подсетью назначения, содержащей IP-адрес `18.19.200.1`. В данном случае это строка 2.
7. Поле «Шлюз» в строке 2 пустое, значит, целевая подсеть доступна непосредственно. Однако колонка «Сетевой контроллер» указывает, что должен использоваться сетевой контроллер NIC 1 с IP-адресом `18.19.200.1`. Этот сетевой контроллер подключен к сети Бета.
8. Модуль IP узла М передает пакет модулю Ethernet, обслуживающему сетевой контроллер NIC 1. Модуль Ethernet использует ARP, чтобы завернуть пакет в кадр Ethernet и послать его по MAC-адресу, соответствующему IP-адресу `18.19.200.1`.
9. Модуль Ethernet узла Б1 принимает пакет, определяет, что в области фактических данных находится IP-пакет, и передает его модулю IP.
10. Модуль IP узла Б1 видит, что IP-адрес получателя совпадает с его собственным адресом, и передает фактические данные из пакета на следующий уровень для дальнейшей обработки.

Этот пример показывает, как две независимые сети могут взаимодействовать, используя косвенную маршрутизацию, но как быть, если понадобится посылать пакеты из этих сетей узлам в Интернете? В этом случае для них нужно сначала получить *действительный* IP-адрес и шлюз от поставщика услуг Интернета (Internet Service Provider, ISP).

Для данного примера представим, что был получен IP-адрес `18.181.0.29` и доступ к шлюзу `18.181.0.1` поставщика услуг. Администратор сети должен теперь добавить еще один сетевой контроллер в узел М и настроить его с полученным от провайдера IP-адресом. Наконец, он должен дополнить таблицы маршрутизации узла М и всех узлов в сети. На рис. 2.9 изображена новая конфигурация сети, а в табл. 2.8, 2.9 и 2.10 — исправленные таблицы маршрутизации.

ПРИМЕЧАНИЕ Поставщик услуг Интернета — это не какая-то конструкция, охватывающая весь Интернет, а просто большая организация с собственным блоком действительных IP-адресов. Что самое интересное, главной задачей этой организации является разделение IP-адресов на подсети и передача в аренду этих подсетей другим организациям.

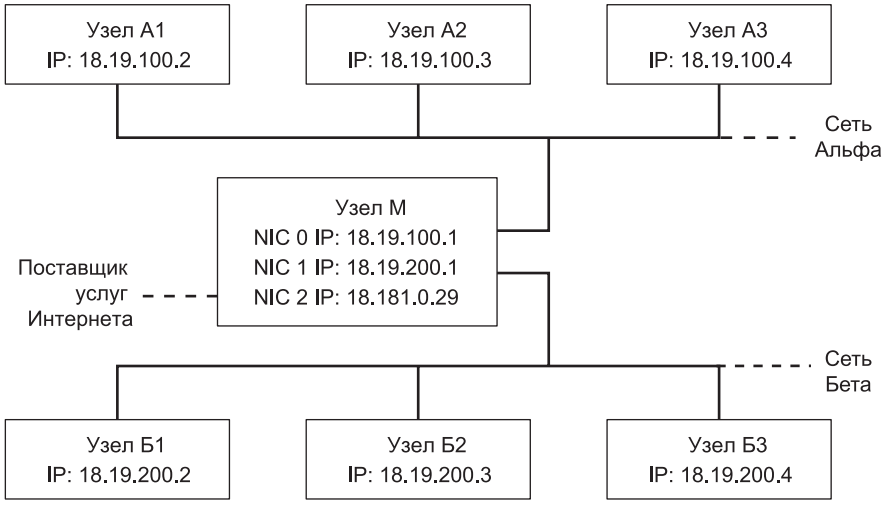


Рис. 2.9. Подключение сетей Альфа и Бета к Интернету

Таблица 2.8. Таблица маршрутизации узла А1 с доступом в Интернет

№ п/п	Подсеть назначения	Шлюз	Сетевой контроллер
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)

Таблица 2.9. Таблица маршрутизации узла Б1 с доступом в Интернет

№ п/п	Подсеть назначения	Шлюз	Сетевой контроллер
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)

Таблица 2.10. Таблица маршрутизации узла М с доступом в Интернет

№ п/п	Подсеть назначения	Шлюз	Сетевой контроллер
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)

Адрес `0.0.0.0/0` называют *адресом по умолчанию*, потому что он определяет подсеть, содержащую все IP-адреса. Если адрес назначения пакета, принятого узлом M, не соответствует ни одной из трех первых строк, он точно будет соответствовать подсети в последней строке. В данном случае пакет будет передан через новый сетевой контроллер на шлюз поставщика услуг Интернета, который направит пакет по маршруту от шлюза к шлюзу, заканчивающийся узлом с адресом назначения. Аналогично узлы A1 и Б1 также получили новые записи с адресом по умолчанию, направляющие пакеты в Интернет через узел M, который в свою очередь будет передавать их шлюзу поставщика услуг Интернета.

Каждый раз, когда пакет попадает на шлюз и перенаправляется дальше, его поле TTL (время жизни) в заголовке IPv4 уменьшается на единицу. Когда значение TTL достигнет 0, пакет сбрасывается узлом, модуль IP которого выполнил заключительное уменьшение. Это предотвращает зацикливание пакетов в Интернете, если окажется так, что пакет начнет курсировать по закольцованному маршруту. После изменения TTL необходимо повторно вычислить контрольную сумму заголовка, что приводит к дополнительным затратам времени на узлах, обрабатывающих пакеты и переправляющих их дальше.

ОСОБЫЕ IP-АДРЕСА

Следует упомянуть два особых IP-адреса. Первый называют *петлевым* (loopback), или *локальным* (localhost), адресом: `127.0.0.1`. Если модулю IP будет предложено отправить пакет по адресу `127.0.0.1`, он никуда его не отправит. Модуль IP будет действовать, как если бы только что получил пакет, и передаст его уровню выше для обработки. Технически весь блок адресов `127.0.0.0/8` должен быть петлевым, но некоторые операционные системы настраивают брандмауэр так, что он пропускает только пакеты на адрес `127.0.0.1`.

Следующий адрес — *широковещательный адрес нулевой сети* `255.255.255.255`. Данный адрес указывает, что пакет должен быть доставлен всем узлам, доступным локальному канальному уровню, и не должен передаваться через маршрутизаторы. Обычно это реализуется завертыванием пакета в кадр канального уровня и передачей его по широковещательному MAC-адресу `FF:FF:FF:FF:FF:FF`.

Значение 0 в поле TTL не единственная причина, по которой пакет может быть сброшен. Например, если пакеты поступают в сетевой контроллер маршрутизатора настолько быстро, что тот не справляется с их обработкой, контроллер может просто игнорировать их. Кроме того, если пакеты поступают на маршрутизатор на несколько сетевых контроллеров, но все должны переправляться через единственный контроллер, недостаточно быстрый, чтобы справиться с таким потоком, некоторые из пакетов могут быть сброшены. Это лишь часть причин, по которым IP-пакеты могут исчезать по пути от отправителя к получателю. Из этого следует, что все протоколы сетевого уровня, включая IPv4, ненадежны. Это означает, что нет никаких гарантий доставки получателю отправленных пакетов IPv4. Более того, если пакеты все же достигнут получателя, это не означает, что они будут получены в том же порядке, в каком были отправлены. Заторы в сети могут вынудить маршрутизаторы отправить часть пакетов для одного и того же получателя по одному

маршруту, а часть — по другому. Эти маршруты могут иметь разную протяженность, из-за чего последние пакеты могут прибыть первыми. Иногда один и тот же пакет может быть отправлен по нескольким маршрутам, из-за чего он будет получен адресатом несколько раз! Под ненадежностью понимается отсутствие гарантий доставки и совпадения порядка доставки с порядком отправки.

Фрагментация

Как упоминалось выше, максимальный объем фактических данных в кадре Ethernet составляет 1500 байт. Однако, как отмечалось прежде, максимальный размер пакета IPv4 составляет 65 535 байт. Вследствие этого возникает вопрос: коль скоро IP-пакеты должны заворачиваться в кадры на канальном уровне, как их размер может превышать значение MTU канального уровня? Ответ заключается в возможности *фрагментации*. Если модулю IP предложить передать пакет, размер которого превышает MTU целевого канального уровня, то такой пакет будет разбит на более мелкие пакеты с размерами, равными MTU.

Фрагменты IP-пакета обрабатываются точно так же, как обычные IP-пакеты, но некоторые поля в их заголовках устанавливаются особым образом. К этим полям относятся: идентификатор фрагмента, флаги фрагмента и смещение фрагмента. Когда модуль IP разбивает IP-пакет на группу фрагментов, для каждого фрагмента создается новый IP-пакет с соответствующими значениями полей в заголовке.

Поле идентификатора фрагмента (16 бит) хранит число, идентифицирующее исходный фрагментированный пакет. Все фрагменты одного пакета имеют одинаковое значение этого поля.

Поле смещения фрагмента (13 бит) определяет смещение данного фрагмента от начала исходного пакета в 8-байтовых блоках. Это число обязательно будет различаться во всех фрагментах в группе. Необычная схема нумерации выбрана, чтобы уместить любое смещение внутри 65 535-байтного пакета в 13 бит. Для этого необходимо, чтобы все смещения были кратны 8 байтам из-за невозможности выразить смещения с большей точностью.

Поле флагов фрагмента (3 бита) получает значение `0x4` во всех фрагментах, кроме последнего. Это значение называют *флагом наличия дополнительных фрагментов* (more fragments flag), оно указывает, что в данной группе существуют другие фрагменты. Если узел примет пакет с этим флагом, он должен дождаться доставки всех фрагментов группы, прежде чем собрать пакет и передать его на вышележащий уровень. Этот флаг не требуется устанавливать в заключительном фрагменте, потому что он имеет ненулевое значение в поле смещения, что также указывает на принадлежность группе фрагментов. В действительности флаг должен быть сброшен в последнем фрагменте, чтобы сообщить об отсутствии других фрагментов, следующих за этим.

ПРИМЕЧАНИЕ Поле флагов фрагмента имеет еще одно назначение. Отправитель IP-пакета может установить в нем значение `0x2`, известное как признак *не фрагментировать*. Этот признак указывает, что пакет не должен фрагментироваться ни при каких условиях. Вместо этого, если модуль IP не сможет передать пакет в канальный уровень из-за того, что его значение MTU меньше размера пакета, этот пакет должен быть сброшен, но не фрагментирован.

В табл. 2.11 показаны соответствующие поля заголовков в большом IP-пакете и в трех пакетах, полученных в результате фрагментирования исходного пакета, чтобы передать его через канал Ethernet.

Таблица 2.11. IPv4-пакет, подлежащий фрагментации

Поле	Значения в исходном пакете	Значения во фрагменте 1	Значения во фрагменте 2	Значения во фрагменте 3
Версия	4	4	4	4
Размер заголовка	20	20	20	20
Общий размер	3020	1500	1500	60
Идентификатор	0	12	12	12
Флаги фрагмента	0	0x4	0x4	0
Смещение фрагмента	0	0	185	370
Время жизни	64	64	64	64
Протокол	17	17	17	17
Адрес отправителя	18.181.0.29	18.181.0.29	18.181.0.29	18.181.0.29
Адрес получателя	181.10.19.2	181.10.19.2	181.10.19.2	181.10.19.2
Фактические данные	3000 байт	1480 байт	1480 байт	40 байт

Поле идентификатора во всех фрагментах содержит значение 12, указывающее, что все три фрагмента относятся к одному и тому же исходному пакету. Число 12 — произвольное, но оно, скорее всего, говорит о том, что это 12-й фрагментированный пакет, отправленный узлом. В первом фрагменте установлен флаг наличия дополнительных фрагментов и смещение, равное 0, указывающее, что он содержит данные, находящиеся в начале исходного пакета. Обратите внимание, что поле общего размера имеет значение 1500. Модуль IP обычно стремится создавать фрагменты максимально возможного размера, чтобы уменьшить их количество. Так как заголовок IP-пакета занимает 20 байт, для фактических данных остается 1480 байт. Из этого следует, что данные для второго фрагмента должны начинаться со смещения 1480. Однако из-за того, что смещения фрагментов измеряются в 8-байтных блоках, а 1480/8 = 185, в действительности значение смещения в данном случае равно 185. Во втором фрагменте также установлен флаг наличия дополнительных фрагментов. Наконец, третий фрагмент имеет смещение 370 и сброшенный флаг наличия дополнительных фрагментов, указывающий, что это последний фрагмент. Общий размер третьего фрагмента равен всего лишь 60 байтам, так как исходный пакет нес в себе 3000 байт данных, имея общий размер 3020 байт. Из них 1480 байт оказалось в первом фрагменте, 1480 байт — во втором и 40 — в третьем.

После отправки фрагменты могут быть подвергнуты дальнейшему фрагментированию. Такое случается, когда на пути к узлу-получателю встречается канальный уровень с еще меньшим значением MTU.

Чтобы пакет был должным образом обработан получателем, все фрагменты должны достигнуть его и быть собранными в исходный нефрагментированный пакет. Из-за заторов в сети, динамического изменения таблиц маршрутизации или по другим причинам пакеты могут достигнуть узла назначения не по порядку, возможно, перемежаясь с другими пакетами от того же или другого узла отправителя. Всякий раз, когда первый фрагмент достигает получателя, модуль IP этого узла имеет достаточно информации, чтобы определить, что фрагмент действительно является фрагментом, а не полным пакетом: это следует из установленного флага наличия других фрагментов или ненулевого значения в поле смещения. В этот момент модуль IP получателя создает буфер размером 64 кбайт (максимальный размер пакета) и копирует данные из фрагмента в буфер с учетом смещения. Буфер маркируется IP-адресом отправителя и идентификационным номером фрагмента пакета, чтобы потом, когда поступят другие фрагменты с соответствующим адресом отправителя и идентификационным номером, модуль IP мог скопировать в буфер новые данные. Когда поступает пакет со сброшенным флагом наличия других фрагментов, получатель вычисляет общий размер исходного пакета, складывая размер фактических данных в пакете и его смещение. После поступления всех данных, принадлежащих пакету, модуль IP передает восстановленный пакет следующему уровню для дальнейшей обработки.

СОВЕТ Несмотря на то что механизм фрагментации IP-пакетов позволяет посылать огромные пакеты, отправлять их не рекомендуется. Во-первых, потому, что из-за этого увеличивается объем данных, фактически передаваемых по сети. Как можно видеть в табл. 2.11, пакет размером 3020 байт был разбит на два пакета по 1500 байт и один пакет размером в 60 байт данных, всего — 3060 байт. Это не особенно большой прирост, но он имеет свойство накапливаться. Во-вторых, если по пути потеряется хотя бы один фрагмент, получатель будет вынужден сбросить весь пакет. Это обстоятельство увеличивает вероятность потери больших пакетов, состоящих из множества фрагментов. По этой причине желательно вообще избегать фрагментации, гарантировав, что размеры IP-пакетов не превысят величину MTU канального уровня. Добиться этого порой очень непросто, потому что на пути между двумя узлами могут встречаться разные протоколы канального уровня: представьте, что пакет передается из Нью-Йорка в Японию. Весьма вероятно, что между двумя узлами встретится хотя бы один канальный уровень, обслуживаемый протоколом Ethernet, поэтому разработчики игр часто исходят из предположения, что минимальный размер MTU для всего маршрута составит 1500 байт. Эти 1500 байт должны включать 20 байт IP-заголовка, фактические данные и любую другую информацию, необходимую обертывающему протоколу, такому как VPN или IPSec, который может использоваться на маршруте. По этой причине разумно ограничить объем фактических данных 1300 байтами.

На первый взгляд может показаться, что лучше ограничить размер пакета еще меньшим значением, например 100 байтами. В конце концов, если вероятность фрагментации 1500-байтного пакета невелика, то уж вероятность фрагментации 100-байтного пакета вообще стремится к нулю, разве не так? Да, это так, но не забывайте, что к каждому пакету прикладывается 20-байтный заголовок. Игра, посылающая пакеты размером всего по 100 байт, будет расходовать 20 % пропускной способности на одни только IP-заголовки, что выглядит неоправданным расточительством. По этой причине, как только у вас будут основания полагать, что минимальное значение MTU не будет ниже 1500, старайтесь формировать пакеты с размером, близким к 1500. В этом случае только 1,3 % пропускной способности будет расходоваться на IP-заголовки, что намного лучше, чем 20 %!

IPv6

Протокол IPv4 с его 32-битными адресами позволяет адресовать 4 миллиарда узлов. Благодаря возможности организации локальных сетей и механизму преобразования адресов (обсуждаются далее в этой главе) можно существенно увеличить число узлов сети, способных обмениваться информацией через Интернет. Однако из-за особенностей распределения IP-адресов и быстрого роста числа персональных компьютеров, мобильных устройств и подключенных к Интернету бытовых приборов 32-битное адресное пространство близко к исчерпанию. Для преодоления этого затруднения, а также проблемы неэффективности IPv4, всплывшей за долгие годы его использования, был разработан протокол IPv6.

В течение следующих нескольких лет IPv6, вероятно, останется неинтересным для разработчиков игр. Как сообщила компания Google, в июле 2014-го примерно 4 % пользователей обращались к ее сайтам с использованием IPv6, что, вероятно, может служить хорошим индикатором, демонстрирующим, как много конечных пользователей использует устройства, подключенные к Интернету через IPv6. Как следствие, игры все еще должны уметь обрабатывать все странности и особенности IPv4, которые был призван исправить IPv6. Однако с распространением игровых платформ следующего поколения, таких как Xbox One, протокол IPv6 в конечном итоге вытеснит IPv4, и поэтому имеет смысл провести небольшое исследование, чтобы понять, что собой представляет IPv6.

Самой заметной новой особенностью IPv6 является длина новых IP-адресов, составляющая 128 бит. Адреса IPv6 записываются как восемь четырехзначных шестнадцатеричных чисел, разделенных двоеточиями. В табл. 2.12 представлены типичные адреса IPv6 в трех допустимых формах.

Таблица 2.12. Типичные формы представления адресов IPv6

Форма	**Адрес**
Полная	2001:4a60:0000:8f1:0000:0000:0000:1013
С сокращением незначащих нулей	2001:4a60:0:8f1:0:0:0:1013
С однократным сокращением сплошных последовательностей нулей	2001:4a60:0:8f1::1013

Незначащие (ведущие) нули в каждом секстете можно отбросить. Кроме того, можно удалить одну сплошную последовательность нулевых секстетов, заменив ее двойным двоеточием. Так как адрес всегда занимает 16 байт, из сокращенной формы легко получить исходную, подставив на место отсутствующих цифр нули.

Первые 64 бита в адресе IPv6 обычно представляют сеть и называются *префиксом сети* (network prefix), а последние 64 бита представляют конкретный узел и называются *идентификатором интерфейса* (interface identifier). Когда необходимо, чтобы сетевой узел, например действующий в роли сервера, имел неконфликтующий IP-адрес, администратор сети может вручную назначить идентификатор интерфейса подобно тому, как назначаются адреса IPv4. Если узлу не требуется, чтобы удаленные клиенты легко могли найти его, он может выбрать случайный иденти-

фикатор интерфейса и объявить об этом в сети, так как вероятность конфликтов в 64-битном пространстве очень невелика. Но чаще идентификатор интерфейса присваивается производителем автоматически, как расширенный уникальный идентификатор EUI-64 (Extended Unique Identifier), который уже гарантированно является уникальным.

Протокол обнаружения соседей (Neighbor discovery protocol, NDP) служит заменой протоколу ARP, а также выполняет некоторые функции протокола DHCP, который описывается далее в этой главе. С помощью NDP маршрутизаторы сообщают свои префиксы сетей и таблицы маршрутизации, а узлы запрашивают и сообщают свои IP-адреса и адреса канального уровня. Более полную информацию о протоколе NDP можно найти в документе RFC 4861, ссылка на который приводится в разделе «Для дополнительного чтения».

Еще одно замечательное отличие от IPv4: протокол IPv6 больше не поддерживает фрагментацию пакетов на уровне маршрутизации. Это дает возможность убрать из IP-заголовка все поля, имеющие отношение к фрагментации, и немного повысить пропускную способность. Если пакет IPv6, достигший маршрутизатора, окажется слишком велик для исходящего канального уровня, маршрутизатор просто сбросит пакет и сообщит отправителю, что пакет слишком большой. После этого отправитель может повторить попытку, уменьшив размер пакета.

Более полную информацию о протоколе IPv6 можно найти в документе RFC 2460, ссылка на который приводится в разделе «Для дополнительного чтения».

Транспортный уровень

Задача сетевого уровня — обеспечить возможность взаимодействия между узлами в удаленных сетях, а задача *транспортного уровня* — обеспечить возможность взаимодействия отдельных процессов, выполняющихся на этих узлах. Так как на одном узле может выполняться множество процессов, не всегда достаточно знать, что узел А послал IP-пакет узлу Б: когда узел Б примет IP-пакет, он должен знать, какому процессу передать его содержимое для дальнейшей обработки. Чтобы решить эту проблему, на транспортном уровне вводится понятие *портов*. Порт — это 16-битное целое беззнаковое число, представляющее конечную точку взаимодействия на конкретном узле. Если сравнить IP-адрес с названием улицы и номером дома в почтовом адресе, то порт можно сравнить с номером квартиры в этом доме. Продолжая эту аналогию, процесс можно рассматривать как жильца, который достает корреспонденцию из одного или нескольких почтовых ящиков в этом доме. С помощью модуля транспортного уровня процессу можно *присвоить* определенный номер порта и сообщить модулю транспортного уровня, что данный процесс будет обрабатывать любые сообщения, поступающие на этот порт.

Как уже упоминалось, все порты являются 16-битными числами. Теоретически процессу можно присвоить любой порт и использовать его для любых взаимодействий. Однако если двум процессам на одном и том же узле попытаться присвоить один и тот же порт, возникнет проблема. Представьте, что двум программам — веб-

серверу и серверу электронной почты — присвоен один и тот же номер порта — 20. В этом случае, если транспортный уровень примет данные для порта 20, должен ли он передать эти данные обоим процессам? Если да, тогда веб-сервер может попытаться интерпретировать входящую электронную почту как веб-запрос или почтовый сервер может попытаться интерпретировать входящий веб-запрос как электронную почту. Это в конечном счете приведет к тому, что либо веб-пользователь, либо отправитель письма будет весьма обескуражен. По этой причине большинство реализаций требуют наличия специального флага на тот случай, когда один и тот же порт присваивается множеству процессов.

Чтобы помочь процессам избежать конфликтов из-за номеров портов, *организация по присвоению имен и адресов в Интернете* (Internet Corporation for Assigned Names and Numbers, ICANN), известная также как *полномочный орган по цифровым адресам в Интернете* (Internet Assigned Numbers Authority, IANA), ведет реестр номеров портов, в котором разработчики различных приложений и протоколов могут зарезервировать порты для своего использования. В реестр может быть включен только один регистрант для каждого порта и протокола транспортного уровня. Номера портов 1024–49 151 известны как *пользовательские*, или *регистрируемые*, *порты*. Разработчик любого протокола или приложения может послать в IANA официальный запрос на присвоение ему номера порта из этого диапазона, который после рассмотрения может быть удовлетворен. Если номер пользовательского порта зарегистрирован в IANA для определенного приложения или протокола, никакие другие приложения или протоколы не должны использовать его, хотя большинство реализаций транспортного уровня не препятствуют этому.

Порты с номерами от 0 до 1023 считаются *системными*, или *зарезервированными*, *портами*. Эти порты подобны пользовательским, но их регистрация в IANA ограничена и производится с соблюдением более строгих правил. Особенность этих портов в том, что большинство операционных систем позволяют присваивать их только процессам, обладающим привилегиями *суперпользователя* и они могут использоваться только для целей, требующих повышенных мер безопасности.

Наконец, порты 49 152–65 535 известны как *динамические*. Они не регистрируются в IANA и законно могут использоваться любыми процессами. Если процесс попытается присвоить себе динамический порт и обнаружит, что тот уже используется, он будет проверять другие динамические порты, пока не найдет свободный. Как добропорядочные пользователи Интернета при создании многопользовательских игр вы должны использовать только динамические порты или зарегистрировать в IANA пользовательский порт, если это понадобится.

После того как приложение получит порт в свое распоряжение, оно сможет использовать протокол транспортного уровня для фактической передачи данных. В табл. 2.13 перечислены некоторые примеры протоколов транспортного уровня, а также соответствующие им номера протоколов. Как разработчики игр мы в основном будем иметь дело с протоколами UDP и TCP.

СОВЕТ Полные адреса отправителей или получателей часто записываются как комбинация IP-адреса и номера порта, перечисленные через двоеточие. То есть полный адрес пакета с IP-адресом `18.19.20.21` и номером порта `80` в заголовке можно записать как `18.19.20.21:80`.

Таблица 2.13. Примеры протоколов транспортного уровня

Название	Аббревиатура	Номер протокола
Transmission control protocol (протокол управления передачей)	TCP	6
User datagram protocol (протокол пользовательских датаграмм)	UDP	17
Datagram congestion control protocol (протокол датаграмм с определением заторов)	DCCP	33
Stream control transmission protocol (протокол передачи с управлением потоком)	SCTP	132

UDP

Протокол пользовательских датаграмм (User Datagram Protocol, UDP) — это легковесный протокол, обертывающий данные и передающий их из порта одного узла в порт другого узла. Датаграмма UDP состоит из 8-байтового заголовка, за которым следуют фактические данные. Структура UDP-заголовка изображена на рис. 2.10.

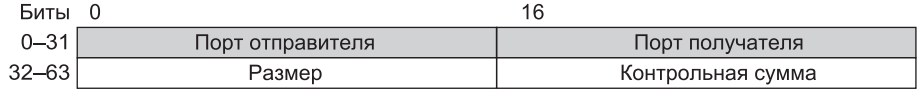


Рис. 2.10. Структура UDP-заголовка

Порт отправителя (16 бит) — номер порта, с которого датаграмма была отправлена. Может пригодиться, если предполагается, что получатель должен ответить на полученную датаграмму.

Порт получателя (16 бит) — целевой номер порта датаграммы. Модуль UDP доставит датаграмму процессу, присвоившему этот порт.

Размер (16 бит) — размер UDP-заголовка и фактических данных.

Контрольная сумма (16 бит) — необязательная контрольная сумма, вычисляемая на основе UDP-заголовка, фактических данных и некоторых полей IP-заголовка. Если контрольная сумма не вычислялась, это поле должно содержать нулевое значение. Часто это поле игнорируется, потому что проверка данных осуществляется на нижележащих уровнях.

UDP — очень простой протокол. Каждая датаграмма считается независимой сущностью, не предполагающей наличия общего состояния между двумя узлами. UDP-пакет можно сравнить с открыткой, брошенной в почтовый ящик и забытой. Протокол UDP не прилагает никаких усилий, чтобы ограничить трафик в перегруженной сети, обеспечить доставку в определенном порядке, как и вообще доста-

вить данные. Этим он сильно отличается от следующего протокола транспортного уровня — TCP, который мы рассмотрим далее.

TCP

Протокол UDP поддерживает передачу отдельных датаграмм между узлами. В отличие от него *протокол управления передачей* (Transmission Control Protocol, TCP) позволяет создать между узлами постоянное соединение и обеспечивает надежную передачу потока данных. Ключевое слово здесь — «надежную». В отличие от всех протоколов, которые мы рассматривали ранее, TCP прилагает максимум усилий, чтобы гарантировать доставку указанному получателю всех отправленных данных в заданном порядке. Как результат, он имеет заголовок большего размера, чем UDP, и требует нетривиальной поддержки состояния соединения на обеих его сторонах. Это дает получателям возможность подтверждать принятые данные, а отправителям — выполнять повторную передачу неподтвержденных пакетов.

Единица передачи данных в TCP называется *сегментом*. Это название отражает тот факт, что TCP предназначен для передачи больших потоков данных, каждый сегмент которых обертывается пакетом нижнего уровня. Сегмент состоит из TCP-заголовка, за которым следуют данные этого сегмента. Структура TCP-заголовка изображена на рис. 2.11.

Рис. 2.11. Структура TCP-заголовка

Порт отправителя (16 бит) и **порт получателя** (16 бит) — номера портов транспортного уровня.

Порядковый номер (32 бита) — монотонно увеличивающийся числовой идентификатор. Каждый байт, переданный протоколом TCP, имеет порядковый номер, который служит идентификатором этого байта. Благодаря этому отправитель может отметить посылаемые данные, а получатель — подтвердить их получение. В качестве порядкового номера сегмента обычно выбирается порядковый номер первого байта в этом сегменте. Исключением из этого правила является процедура установки соединения, которая описывается в разделе «Трехэтапное согласование».

Номер подтверждения (32 бита) содержит порядковый номер следующего байта, который ожидает получить отправитель. Он используется для подтверждения

приема всех данных с порядковыми номерами меньше этого числа. Так как TCP гарантирует доставку всех данных в определенном порядке, порядковый номер следующего байта, который ожидает получить узел, всегда на единицу больше порядкового номера предыдущего, уже принятого байта. Запомните: отправитель этого номера подтверждает получение не порядкового номера с этим значением, а всех порядковых номеров меньше этого значения.

Размер заголовка (4 бита) определяет размер заголовка в 32-битовых словах. Протокол TCP позволяет включать в конец заголовка некоторые необязательные поля, поэтому размер заголовка может колебаться от 20 до 64 байт.

Флаги (9 бит) хранят метаданные о заголовке. Они обсуждаются ниже, где это необходимо.

Размер окна приема (16 бит) хранит размер свободного пространства в буфере, которое еще может использовать отправитель для входящих данных. Используется для управления потоком данных, о чем рассказывается далее.

Указатель важности (16 бит) хранит расстояние от первого байта в данном сегменте до первого байта внеочередных данных. Анализируется, только если установлен флаг URG.

ПРИМЕЧАНИЕ Для определения 8-битного блока информации во многих документах RFC, включая те, что определяют основные протоколы транспортного уровня, вместо слова «байт», имеющего слишком свободное толкование, используется более однозначный термин «октет». Некоторые устаревшие платформы использовали байты, содержащие больше или меньше 8 бит, поэтому стандартизация термина «октет» помогла обеспечить совместимость между платформами. В настоящее время эта проблема потеряла свою остроту, потому что на всех платформах, с которыми приходится сталкиваться разработчикам игр, под байтом подразумевается 8 бит.

Надежность

Рисунок 2.12 иллюстрирует обобщенный способ, которым TCP гарантирует надежную передачу данных между узлами. Вкратце, узел-отправитель посылает уникально идентифицируемый пакет узлу-получателю. Затем отправитель ждет от получателя ответного пакета, подтверждающего прием пакета. Если подтверждение не приходит в течение определенного промежутка времени, отправитель повторяет передачу пакета. Так продолжается до тех пор, пока все данные не будут отправлены и подтверждены.

В действительности алгоритм этого процесса выглядит несколько сложнее, и мы остановимся на нем подробнее, так как он может служить отличным образцом организации надежной передачи данных. Поскольку стратегия TCP предусматривает повторную передачу данных и слежение за ожидаемыми порядковыми номерами, каждый узел должен хранить сведения обо всех активных TCP-соединениях. В табл. 2.14 перечислены некоторые переменные состояния и их стандартные обозначения, определяемые документом RFC 793. Процедура инициализации состояния начинается с трехэтапного согласования между двумя узлами.

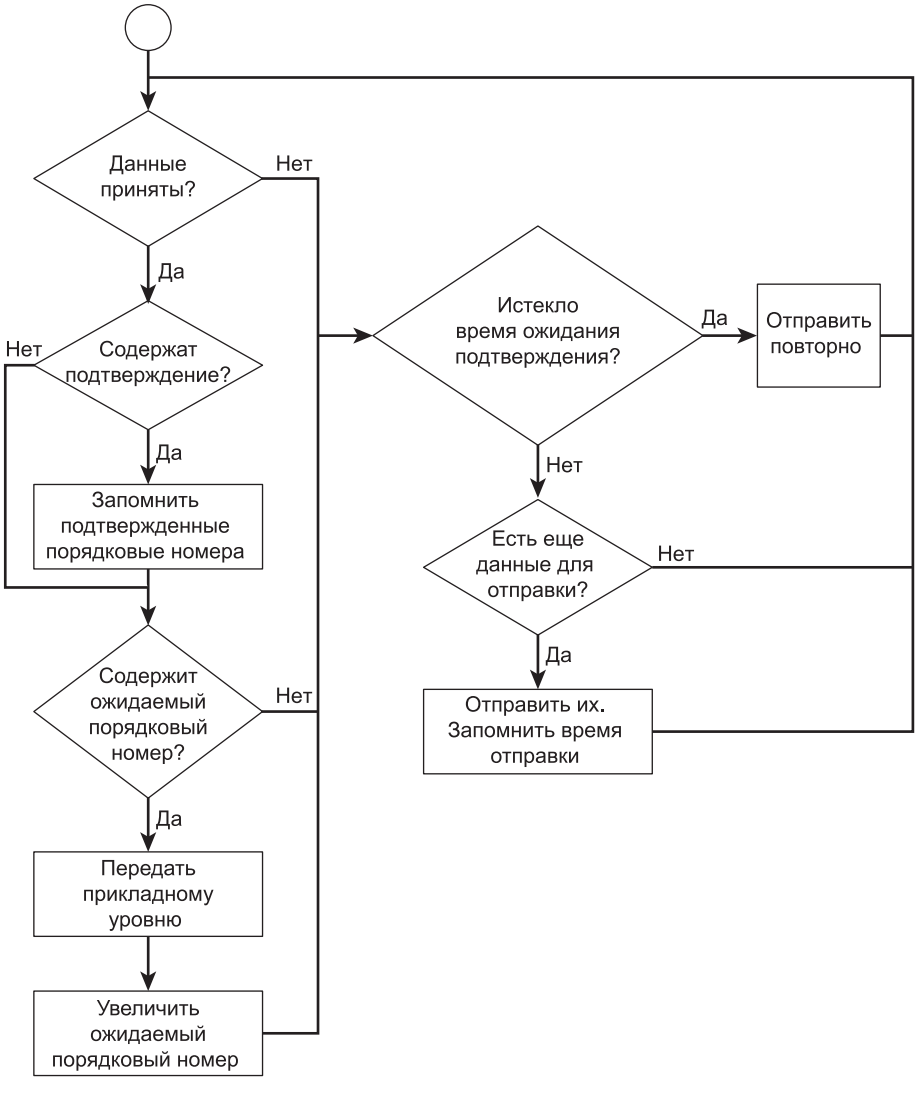


Рис. 2.12. Блок-схема алгоритма надежной передачи данных протоколом TCP

Таблица 2.14. Переменные состояния TCP

Переменная	Обозначение	Определение
Send Next	SND.NXT	Порядковый номер следующего сегмента для отправки
Send Unacknowledged	SND.UNA	Порядковый номер самого старого байта, отправленного узлом и пока не подтвержденного

Таблица 2.14 (*окончание*)

Переменная	Обозначение	Определение
Send Window	SND.WND	Текущий объем данных, который узел может отправить до приема подтверждения для неподтвержденных данных
Receive Next	RCV.NXT	Следующий порядковый номер, который ожидает получить узел
Receive Window	RCV.WND	Текущий объем данных, который узел может принять до исчерпания места в приемном буфере

Трехэтапное согласование

Рисунок 2.13 иллюстрирует процесс трехэтапного согласования между узлами А и Б. Узел А инициирует соединение, посылая первый сегмент. Этот сегмент имеет флаг `SYN` и случайно выбранный начальный порядковый номер 1000. Сегмент сообщает узлу Б, что узел А желает инициировать TCP-соединение с начальным порядковым номером 1000 и что узел Б должен инициализировать ресурсы, необходимые для поддержания соединения.

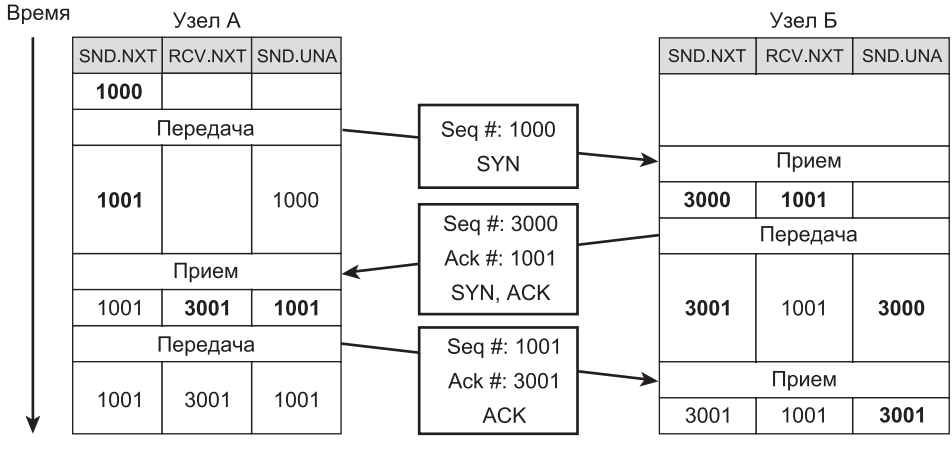


Рис. 2.13. Трехэтапное согласование

Узел Б, если желает и имеет возможность открыть соединение, отвечает пакетом с установленными флагами `SYN` и `ACK`. Он подтверждает порядковый номер, присланный узлом А, увеличивая его на единицу и записывая в поле «Номер подтверждения». Это означает, что следующий сегмент, который узел Б ожидает получить от узла А, должен содержать порядковый номер, на единицу больший, чем предыдущий. Кроме того, узел Б случайным образом выбирает собственный порядковый номер (в нашем случае — 3000) для начала передачи потока данных узлу А. Обратите особое внимание, что узлы А и Б выбирают случайные начальные порядковые номера. С соединением связаны два отдельных потока данных:

один — от узла А к узлу Б, для которого используются порядковые номера, устанавливаемые узлом А, и другой — от узла Б к узлу А, в котором порядковые номера устанавливаются узлом Б. Присутствие флага SYN в сегменте как бы говорит: «Эй, там! Мне есть что передать вам, начиная с байта, отмеченного порядковым номером, на единицу большим, чем порядковый номер в этом сегменте». Наличие флага ACK и номера подтверждения во втором сегменте означает: «Я готов. Кстати, я принял все данные, отправленные до этого порядкового номера, поэтому возвращаю вам порядковый номер, который ожидаю получить от вас в следующем сегменте». Когда узел А примет этот сегмент, ему останется только подтвердить начальный порядковый номер, присланный узлом Б, посылкой сегмента с установленным флагом ACK и номером подтверждения, на единицу большим порядкового номера, присланного узлом Б, то есть 3001.

ПРИМЕЧАНИЕ Когда TCP-сегмент содержит флаг SYN или FIN, порядковый номер увеличивается на один дополнительный байт, чтобы показать присутствие флага. Иногда это называют *фантомным байтом TCP*.

Надежность гарантируется подтверждением приема отправленных данных. Если время ожидания подтверждения истекло и узел А не принял сегмент SYN-ACK, предполагается, что узел Б не принял сегмент с флагом SYN или его ответ потерялся. В любом случае узел А может повторить отправку начального сегмента. Если узел Б принял сегмент с флагом SYN и в первый и во второй раз, предполагается, что узел А не получил ответа SYN-ACK, поэтому он может повторить отправку сегмента SYN-ACK.

Передача данных

Для передачи данных узлы могут включать их в каждый исходящий сегмент. Каждый сегмент маркируется порядковым номером первого байта фактических данных в последовательности. Каждый байт имеет свой порядковый номер, а это означает, что порядковый номер сегмента должен совпадать с порядковым номером предыдущего сегмента, увеличенным на число байтов в предыдущем сегменте. Кроме того, когда сегмент с данными принимается получателем, он посылает пакет подтверждения с номером подтверждения, соответствующим следующему ожидаемому порядковому номеру. Обычно он совпадает с порядковым номером предыдущего сегмента, увеличенным на число байтов в предыдущем сегменте. На рис. 2.14 показано, как протекает простая передача без потери сегментов. Узел А посылает 100 байт в первом сегменте, узел Б подтверждает и посылает 50 своих байтов, узел А посылает еще 200 байт, и затем узел Б подтверждает получение этих 200 байт, не посылая никаких дополнительных данных.

Ситуация осложняется, когда происходит потеря сегментов или они доставляются в неправильном порядке. На рис. 2.15 узел А посылает узлу Б сегмент 1301, который теряется на полпути. Узел А ждет пакета подтверждения с номером подтверждения 1301. Когда время ожидания истекает, а узел А так и не получает подтверждения, он считает, что что-то пошло не так: где-то между узлами А и Б потерялся либо сегмент 1301, либо пакет подтверждения. В любом случае узел А принимает решение повторять передачу сегмента 1301, пока не будет получено подтвержде-

ние от узла Б. Чтобы повторить передачу, узел А должен хранить копию сегмента с данными под рукой, и это является ключевой особенностью работы TCP: модуль TCP должен хранить копию посылаемых байтов, пока не получит подтверждение от получателя. Только после получения подтверждения модуль TCP может стереть этот сегмент из памяти.

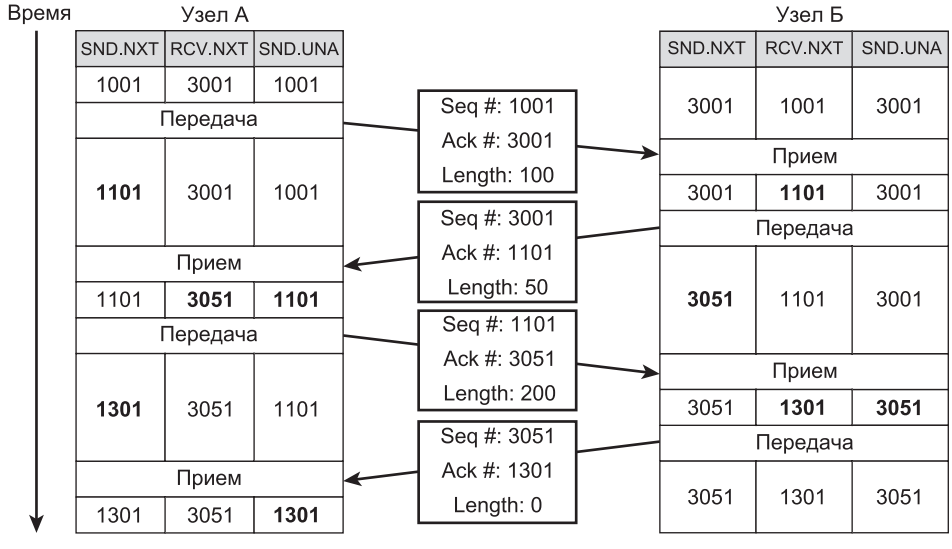


Рис. 2.14. Передача данных протоколом TCP без потери пакетов

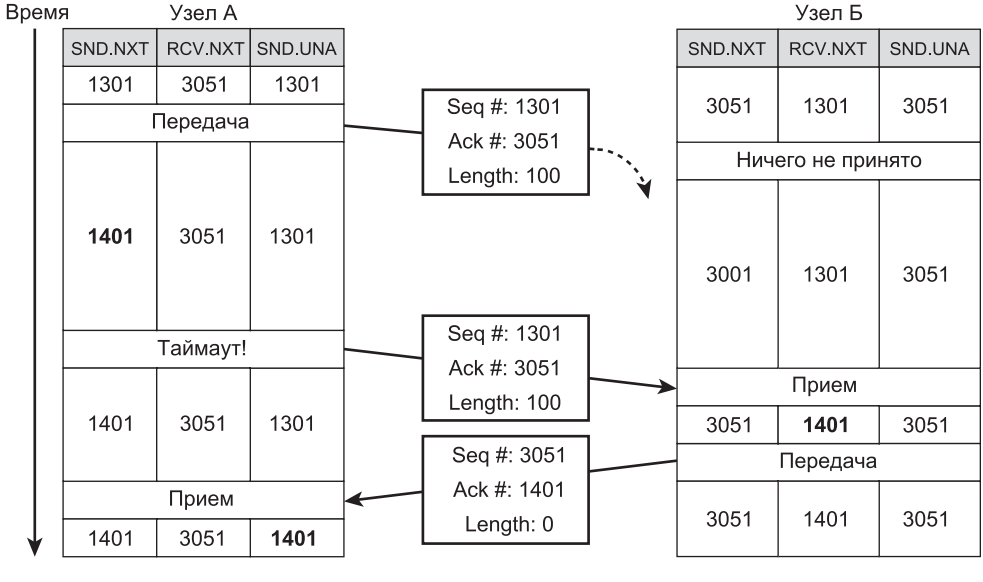


Рис. 2.15. Потеря TCP-пакета и повторная его передача

Протокол TCP гарантирует доставку данных в определенном порядке, то есть если узел примет пакет с порядковым номером, который не ожидался, у него есть два пути. Самый простой — сбросить пакет и ждать повторной передачи, когда он окажется на своем месте. Альтернативный — сохранить сегмент в буфере без подтверждения и без передачи прикладному уровню. То есть скопировать сегмент в локальный буфер потока, в позицию, соответствующую порядковому номеру. Затем, когда все предыдущие сегменты будут получены, подтвердить сегмент, пришедший не по порядку, и передать его прикладному уровню для обработки, не дожидаясь повторной передачи отправителем.

В предыдущих примерах узел A всегда ждет подтверждения, прежде чем послать очередную порцию данных. Такой порядок работы не соответствует действительности и описан, только чтобы упростить примеры. Нет никаких требований, которые вынуждали бы узел A останавливать передачу в ожидании подтверждения после отправки каждого сегмента. Фактически, если бы такое требование существовало, протокол TCP невозможно было бы использовать для передачи информации на большие расстояния.

Как вы наверняка помните, величина MTU для Ethernet составляет 1500 байт. Из них не менее 20 байт занимает заголовок IPv4, заголовок TCP занимает еще как минимум 20 байт, то есть всего в нефрагментированном сегменте TCP, передаваемом через Ethernet, можно отправить до 1460 байт данных. Это число называют *максимальным размером сегмента* (Maximum Segment Size, MSS). Если бы протокол TCP ожидал подтверждения после отправки каждого сегмента, его пропускная способность была бы весьма и весьма ограничена. В этом случае ее можно было бы вычислить как значение MSS, деленное на время доставки сегмента от отправителя до получателя, плюс время доставки подтверждения от получателя до отправителя (*время передачи/подтверждения* — Round Trip Time, RTT). Время передачи/подтверждения через всю страну составляет примерно 30 мс. То есть максимальная пропускная способность, достижимая протоколом TCP на таких расстояниях, могла бы составить 1500 байт/0,03 с, или 50 кбайт/с, независимо от скоростных возможностей канального уровня. Эта скорость могла бы показаться весьма приличной в 1993 году, но не сегодня!

Для решения проблемы протоколу TCP позволено послать сразу множество сегментов, не дожидаясь подтверждения. Однако количество таких сегментов не может быть бесконечным, так как возникает другая проблема. Когда данные транспортного уровня достигают узла, они сохраняются в буфере, пока процесс, связанный с соответствующим портом, не извлечет их. В этот момент данные удаляются из буфера. Неважно, каким объемом памяти обладает узел, сам буфер имеет некоторый фиксированный размер. Легко представить, что сложный процесс на медленном процессоре может не справляться с обработкой данных, поступающих слишком быстро. То есть когда буфер заполнится, вновь прибывающие данные будут отбрасываться. Для протокола TCP это означает, что получаемые данные не будут подтверждаться и быстродействующий отправитель повторит передачу данных с высокой скоростью. Весьма вероятно, что большая часть вновь отправленных данных опять будет сброшена, потому что принимающий узел имеет все тот же медленный процессор, на котором выполняется все тот же сложный процесс.

В результате значительная доля трафика будет потеряна, что является недопустимым разбазариванием ресурсов Интернета.

Чтобы устранить эту катастрофическую неэффективность, TCP реализует процедуру, известную как *управление потоком данных* (flow control). Управление потоком не позволяет быстродействующему узлу «завалить» пакетами менее быстродействующего потребителя. Каждый TCP-заголовок содержит поле с размером окна приема, значение которого указывает, какой объем приемного буфера доступен отправителю пакетов. С его помощью принимающая сторона сообщает передающему узлу максимальный объем данных, который тот может послать, прежде чем приостановиться в ожидании подтверждения. Рисунок 2.16 иллюстрирует обмен пакетами между быстродействующим передающим узлом А и медленным принимающим узлом Б.

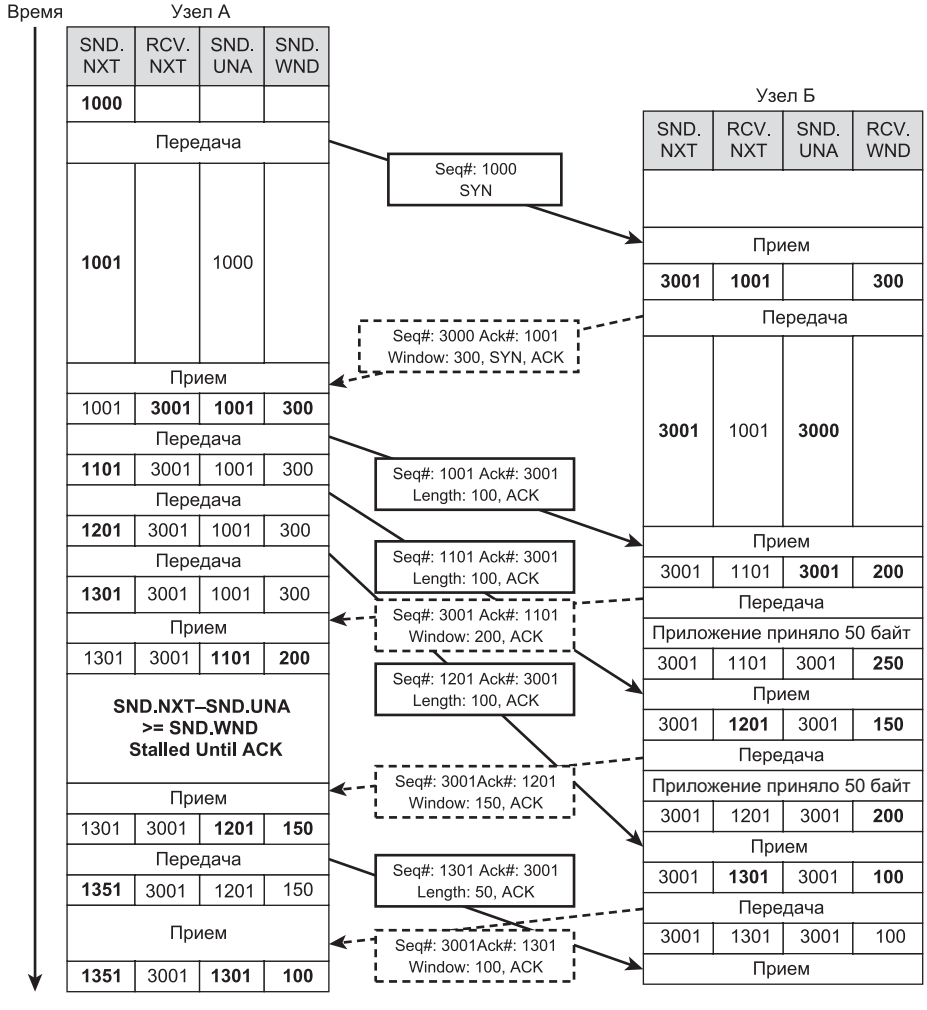


Рис. 2.16. Управление потоком данных в TCP

Исключительно для демонстрации максимальный размер сегмента (MSS) в нашем примере ограничен 100 байтами. Узел Б в начальном пакете `SYN-ACK` указывает размер окна приема 300 байт, поэтому узел А посылает только три 100-байтных сегмента и приостанавливается в ожидании подтверждения от узла Б. Когда узел Б посылает первое подтверждение, он знает, что теперь в его буфере занято 100 байт, которые не могут быть обработаны достаточно быстро, поэтому он сообщает узлу А, что окно приема имеет размер 200 байт. Узел А помнит, что еще 200 байт уже находятся в пути к узлу Б, поэтому в ответ ничего не посылает. Он вынужден простаивать, ожидая еще одного подтверждения от узла Б. К моменту, когда узел Б готов подтвердить второй пакет, 50 байт данных уже извлечено приложением из буфера, то есть теперь в буфере 150 байт занято и 150 байт свободно. Посылая подтверждение узлу А, он сообщает, что окно приема имеет размер 150 байт. Узел А в этот момент помнит, что все еще не подтверждено получение 100 байт, но приемное окно имеет размер 150 байт, поэтому он посылает узлу Б дополнительный 50-байтный сегмент.

Управление потоком данных продолжает действовать по этой схеме: узел Б всегда сообщает узлу А, какой объем данных он может сохранить, чтобы узел А не послал данных больше, чем узел Б сможет сохранить в буфере. Учитывая вышесказанное, теоретический предел полосы пропускания для потока данных TCP можно вычислить по следующей формуле:

$$\text{Ширина полосы пропускания} \times \frac{\text{Размер окна приема}}{\text{Время передачи/подтверждения}}.$$

Из-за слишком маленького окна приема может образоваться узкое место в передаче TCP. Во избежание этого размер окна приема должен быть достаточно большим, чтобы теоретическая пропускная способность всегда была больше максимальной скорости передачи канального уровня между узлами.

Обратите внимание, что на рис. 2.16 узел Б посылает узлу А два подтверждения подряд. Это не самое эффективное использование полосы пропускания, потому что номер подтверждения во втором пакете `ACK` подтверждает также все байты, которые были подтверждены первым пакетом `ACK`. На передачу одних только IP- и TCP-заголовков было впустую потрачено 40 байт полосы пропускания от узла Б к узлу А. А если учесть расходы на формирование кадра канального уровня, напрасные затраты оказываются еще больше. Чтобы устранить такое неэффективное расходование ресурсов, правила TCP предусматривают возможность *отложенного подтверждения* (delayed acknowledgment). Согласно определению протокола, узел, принявший TCP-сегмент, не обязан немедленно подтверждать его — он может выждать до 500 мс или до момента получения следующего сегмента, в зависимости от того, какое событие наступит раньше. В предыдущем примере если узел Б получит сегмент с порядковым номером 1101 в течение 500 мс с момента получения сегмента с порядковым номером 1001, он должен послать подтверждение только для сегмента 1101. Для тяжелых потоков данных такой прием эффективно устраняет половину подтверждений и всегда дает принимающему узлу время извлечь какую-то часть данных из буфера, благодаря чему последний может вернуть увеличенный размер окна приема в своем подтверждении.

Механизм управления потоком данных помогает протоколу TCP защитить «слабых» потребителей от заваливания данными, но он ничего не делает, чтобы предотвратить такое же переполнение медленных сетей и маршрутизаторов. Трафик в сетях во многом напоминает движение автомобилей по крупной магистрали. В особенно популярных маршрутизаторах затор образуется как на оживленных перекрестках, так и в местах съезда с магистралей. Чтобы избежать перегрузки сетей, протокол TCP реализует механизм *управления заторами* (congestion control), который по своим функциям напоминает светофоры. Для уменьшения нагрузки модуль TCP сознательно ограничивает объем неподтвержденных данных, который может находиться в пути. Это похоже на работу механизма управления потоком данных, только вместо размера окна приема, устанавливаемого принимающей стороной, здесь используется значение, вычисленное на основе числа подтвержденных или сброшенных пакетов. Точный алгоритм зависит от конкретной реализации, но обычно он является некоторой разновидностью системы с аддитивным увеличением и мультипликативным уменьшением. То есть, устанавливая соединение, модуль TCP определяет окно перегрузки с размером, кратным MSS. Обычно выбирается размер, равный двум MSS. Далее, с получением подтверждения для каждого сегмента размер окна перегрузки увеличивается на величину MSS. В идеальных соединениях, при подтверждении всех пакетов, окно перегрузки будет удваиваться в размерах. Однако если пакет будет потерян, модуль TCP быстро уменьшит окно перегрузки наполовину, предполагая, что потеря вызвана заторами в сети. В результате этого достигается равновесие, при котором отправитель посылает данные со скоростью, не вызывающей такого увеличения трафика, при котором пакеты начинают теряться.

Протокол TCP может также способствовать уменьшению заторов в сети, посылая пакеты с размерами, максимально близкими к MSS. Так как к каждому пакету прилагается 40 байт заголовков, передача нескольких маленьких сегментов оказывается менее эффективной, чем объединение сегментов в один большой фрагмент и передача его по мере готовности. Это означает, что модуль TCP должен иметь исходящий буфер для накопления данных, посылаемых уровнем, находящимся выше. Чтобы определить, когда накапливать данные и когда, наконец, отправить сегмент, многие реализации TCP используют свод правил под названием *алгоритм Нейгла* (Nagle's algorithm). Традиционно, если данные уже были отправлены, но еще не были подтверждены, производится накопление данных, пока их объем не превысит размер MSS или окна перегрузки, — в зависимости от того, что меньше. Как только это произойдет, посылается наибольший сегмент, допускаемый этими двумя пределами.

СОВЕТ Алгоритм Нейгла — беда для игроков, предпочитающих игры, которые используют TCP в качестве протокола транспортного уровня. Позволяя снизить потребление полосы пропускания, он может существенно увеличить задержку перед отправкой данных. Если игре, действующей в режиме реального времени, требуется часто посылать серверу небольшие изменения, может пройти множество игровых кадров, пока накопится достаточное число изменений, чтобы заполнить MSS. В результате у игрока может сложиться впечатление, что игра «тормозит», хотя это всего лишь эффект работы алгоритма Нейгла. По этой причине большинством реализаций TCP поддерживается параметр, позволяющий отключить эту функцию управления заторами.

Отключение

Разрыв соединения TCP требует запроса на завершение и подтверждения с каждой стороны. Завершив передачу данных, узел посылает пакет с установленным флагом `FIN`, сообщая, что готов прекратить передачу данных. Все данные в исходящем буфере, включая пакет `FIN`, посылаются и пересылаются, пока не будет получено подтверждение. При этом модуль TCP не будет принимать новые исходящие данные от вышележащего уровня. Однако данные, поступающие от другого узла, все еще могут приниматься и будут подтверждаться. Когда другая сторона закончит передачу, она также отправит пакет `FIN`. Когда узел, инициировавший процедуру закрытия соединения, получит от другого узла пакет `FIN` и пакет `ACK` в ответ на свой пакет `FIN` или когда истечет время ожидания пакета `ACK`, модуль TCP полностью закроет соединение и удалит всю информацию о нем.

Прикладной уровень

На самом верху слоеного пирога TCP/IP находится прикладной уровень, где действует программный код многопользовательской игры. Прикладной уровень также служит прибежищем для множества основных протоколов Интернета, опирающихся на транспортный уровень, и здесь мы исследуем некоторые из них.

DHCP

Присваивание уникальных адресов IPv4 узлам в локальной подсети может оказаться непростой задачей для администраторов, особенно когда в работу включаются ноутбуки и смартфоны. *Протокол динамической конфигурации сетевого узла* (Dynamic Host Configuration Protocol, DHCP) решает эту проблему, давая возможность узлу запросить параметры настройки при подключении к сети.

Подключаясь к сети, узел создает сообщение `DHCPDISCOVER` с собственным MAC-адресом и посылает его по протоколу UDP на адрес `255.255.255.255:67`. Поскольку это сообщение получит каждый узел в подсети, любой имеющийся в ней сервер DHCP также примет его. Сервер DHCP при наличии свободного IP-адреса для клиента подготовит пакет `DHCPOFFER` с предлагаемым IP-адресом и MAC-адресом клиента. В этот момент клиент пока не имеет IP-адреса, поэтому сервер не может послать пакет ему непосредственно. Вместо этого он пошлет широковещательный пакет на UDP-порт 68. Все клиенты DHCP примут пакет, и каждый проверит MAC-адрес в сообщении, чтобы определить, не адресовано ли сообщение ему. Когда нужный клиент получит сообщение, он прочитает предложенный IP-адрес и решит, принимает он предложение или нет. В случае положительного решения он отвечает широковещательным сообщением `DHCPREQUEST` с просьбой закрепить за ним предложенный адрес. Если предложение все еще в силе, сервер ответит опять же широковещательным сообщением `DHCPACK`, одновременно подтверждающим присваивание IP-адреса клиенту и передающим любую дополнительную информацию о сети, такую как маска подсети, адрес маршрутизатора и любые рекомендуемые серверы имен DNS.

Точный формат пакетов DHCP и дополнительную информацию можно найти в документе RFC 2131, ссылка на который приводится в разделе «Для дополнительного чтения».

DNS

Протокол системы доменных имен (Domain Name System, DNS) осуществляет преобразование имен доменов и поддоменов в IP-адреса. Когда конечному пользователю понадобится выполнить поиск в Google, ему не придется вводить адрес `74.125.224.112` в веб-браузере — он может просто ввести www.google.com. Чтобы преобразовать доменное имя в IP-адрес, веб-браузер клиента пошлет DNS-запрос по IP-адресу сервера имен, на использование которого настроен компьютер.

Сервер имен (name server) хранит карту соответствий доменных имен и IP-адресов. Например, в этой карте имени www.google.com может соответствовать IP-адрес `74.125.224.112`. В Интернете существуют тысячи серверов имен, но большинство из них хранит информацию только о небольшом подмножестве доменов и поддоменов Интернета. Если запрашиваемый сервер имен не владеет информацией о требуемом домене, он обычно имеет ссылку на более осведомленный сервер имен, к которому и обращается с аналогичным запросом. Результат второго запроса обычно сохраняется в кэше, чтобы при следующем обращении к этому же домену сервер имен мог сразу же вернуть ответ.

Запросы и ответы DNS обычно посылаются по протоколу UDP на порт 53. Их форматы определяются в документе RFC 1035, ссылка на который приводится в разделе «Для дополнительного чтения».

NAT

До сих пор все обсуждавшиеся IP-адреса были *глобально маршрутизируемыми* (publically routable). IP-адрес считается глобально маршрутизируемым, если любой правильно настроенный маршрутизатор в Интернете может построить маршрут пакета так, что тот в конечном счете достигнет узла с этим IP-адресом. Для этого любой глобально маршрутизируемый адрес должен быть присвоен только одному узлу. Если один и тот же IP-адрес будет присвоен двум узлам, пакет, предназначенный одному узлу, может быть передан другому. Если один из таких узлов выполнит запрос к веб-серверу, ответ может быть отправлен другому узлу.

Чтобы обеспечить уникальность глобально маршрутизируемых адресов, организация ICANN и ее филиалы распределяют отдельные блоки IP-адресов между крупными организациями, такими как корпорации, университеты и поставщики услуг Интернета, которые затем могут передавать эти адреса своим сотрудникам и клиентам, гарантируя уникальность каждого адреса.

Так как IPv4 имеет лишь 32-битное адресное пространство, теоретически существует 4,294,967,296 глобальных IP-адресов. Из-за невероятного роста числа сетевых устройств и особенностей распределения IP-адресов организацией ICANN это количество оказалось недостаточным. Часто сетевой администратор

или пользователь может обнаружить, что число узлов его сети превышает число имеющихся глобальных IP-адресов. Например, у каждого из нас как разработчиков видеоигр наверняка имеется смартфон, ноутбук и игровая консоль, но только один глобальный IP-адрес, полученный в аренду у поставщика услуг Интернета. Насколько было бы неудобно, если бы каждое устройство требовало назначения ему выделенного глобального IP-адреса? Каждый раз, подключая новый гаджет к Интернету, мы могли бы вступать в состязание с другими пользователями за получение нового IP-адреса у поставщика услуг Интернета и, соответственно, платить за этот адрес.

К счастью, благодаря механизму *преобразования сетевых адресов* (Network Address Translation, NAT) существует возможность подключения к Интернету целой подсети, имея единственный глобальный IP-адрес. Чтобы настроить преобразование адресов (NAT) для сети, каждому узлу в этой сети нужно присвоить *локально маршрутизируемый* (privately routable) IP-адрес. В табл. 2.15 перечислены некоторые блоки IP-адресов, которые организация IANA зарезервировала для частного использования, гарантируя, что ни один адрес из этих блоков никогда не будет использоваться как глобальный IP-адрес. То есть любой пользователь может сконструировать свою локальную сеть, используя локально маршрутизируемые IP-адреса, без необходимости проверять их уникальность. Обеспечивать уникальность адресов не требуется, потому что адреса не являются глобально маршрутизируемыми. То есть ни один маршрутизатор в Интернете не имеет информации о том, как достигнуть узла с локальным IP-адресом, поэтому совершенно неважно, сколько локальных сетей используют одни и те же локальные IP-адреса.

Таблица 2.15. Блоки локальных IP-адресов

Диапазон IP-адресов	Подсеть
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

Чтобы понять, как выполняется преобразование сетевых адресов, рассмотрим домашнюю сеть любителя видеоигр, изображенную на рис. 2.17. Игровая консоль, смартфон и ноутбук — всем этим устройствам владельцем сети присвоены уникальные локальные IP-адреса, для этого ему не потребовалось согласовывать свои действия с каким-либо поставщиком услуг. Сетевому контроллеру маршрутизатора, обращенному в локальную сеть, также присвоен локальный IP-адрес, а его сетевому контроллеру, обращенному в Интернет, присвоен глобально маршрутизируемый IP-адрес, выданный поставщиком услуг Интернета. Так как локально адресуемый сетевой контроллер подключен к локальной сети, его называют портом *локальной сети* (Local Area Network, LAN), а так как глобально адресуемый сетевой контроллер подключен к глобальной сети, его называют портом *глобальной сети* (Wide Area Network, WAN).

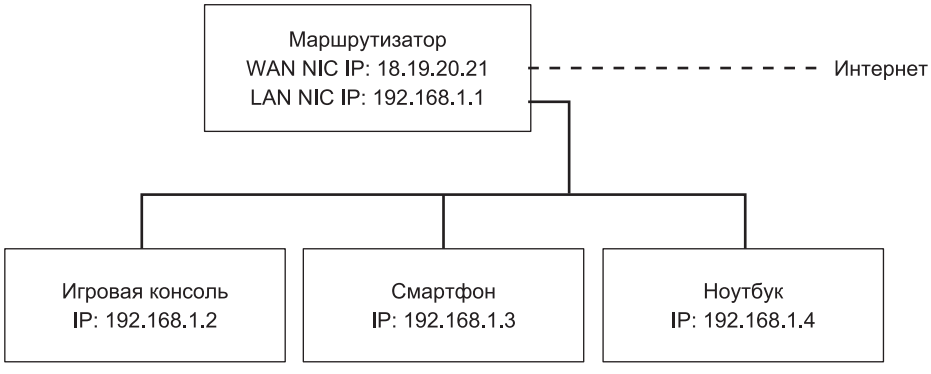


Рис. 2.17. Локальная сеть, находящаяся за механизмом преобразования сетевых адресов

Для этого примера предположим, что на узле с глобально маршрутизируемым IP-адресом `12.5.3.2` действует игровой сервер, использующий порт 200. На игровой консоли с локальным IP-адресом `192.168.1.2` выполняется игра, использующая порт 100. Игровой консоли потребовалось послать сообщение на сервер по протоколу UDP, поэтому она конструирует датаграмму, как показано на рис. 2.18, с адресом отправителя `192.168.1.2:100` и адресом получателя `12.5.3.2:200`. Если допустить, что на маршрутизаторе отключено преобразование сетевых адресов, консоль пошлет датаграмму в локальный порт маршрутизатора, который перенаправит его в глобальный порт, подключенный к Интернету. В конечном счете пакет достигнет сервера. И в этот момент возникнет проблема. Так как пакет имеет адрес отправителя `192.168.1.2`, сервер не сможет послать ответ обратно. Как вы помните, адрес `192.168.1.2` является локальным, поэтому ни один маршрутизатор в Интернете не сможет проложить маршрут к этому адресу. Даже если на каком-то маршрутизаторе окажется совершенно бессмысленная для него информация о маршруте к этому IP-адресу, весьма мало вероятно, что пакет достигнет игровой консоли, так как в Интернете существуют многие тысячи узлов с локальным IP-адресом `192.168.1.2`.

Чтобы решить проблему, модуль NAT на маршрутизаторе может изменить IP-пакет, заменив локальный IP-адрес `192.168.1.2` глобальным IP-адресом маршрутиза-

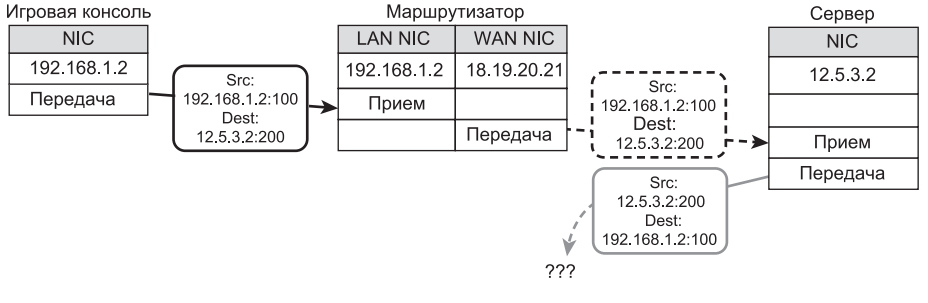


Рис. 2.18. Маршрутизатор без поддержки преобразования сетевых адресов

тора `18.19.20.21`. Это решит часть проблемы, но не всю: простая подмена IP-адреса создаст ситуацию, изображенную на рис. 2.19. Для сервера датаграмма будет выглядеть так, как если бы она была отправлена непосредственно с глобального IP-адреса маршрутизатора, поэтому он благополучно отправит ответ маршрутизатору. Однако маршрутизатор не помнит, кто отправил исходную датаграмму, поэтому он не знает, кому направить ответ.

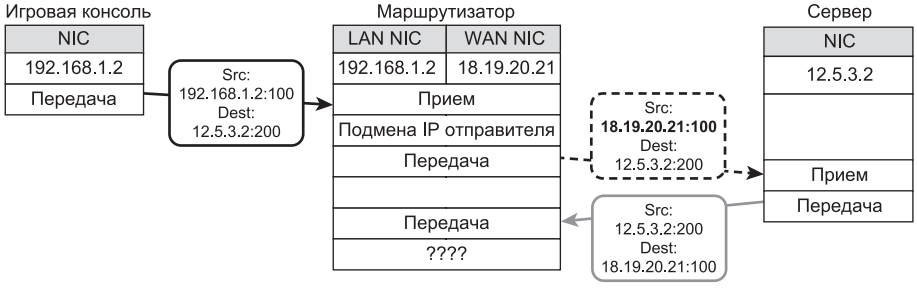


Рис. 2.19. Маршрутизатор, подменяющий адрес отправителя

Чтобы вернуть ответ нужному локальному узлу, маршрутизатор должен иметь некоторый механизм, определяющий получателя входящего пакета в локальной сети. Самый простой способ: создать таблицу и записывать в нее IP-адреса отправителей всех исходящих пакетов. Затем, при получении ответа с внешнего IP-адреса, просмотреть таблицу, найти внутренний узел, пославший пакет по этому адресу, и затем записать в пакет IP-адрес внутреннего узла. Однако это решение окажется неработоспособным, если сразу несколько внутренних узлов будут посылать пакеты одному и тому же внешнему узлу. Маршрутизатор не сможет определить, какие входящие пакеты какому из внутренних узлов предназначены.

Решение, используемое всеми современными маршрутизаторами с поддержкой NAT, заключается в том, чтобы сломать барьер между сетевым и транспортным уровнем. Подменяя не только IP-адреса в IP-заголовке, но и номера портов в заголовке транспортного уровня, маршрутизатор может создать более точную систему отображения и маркировки и сохранить ее в таблице NAT. Взгляните на рис. 2.20, где показано, как движется пакет от игровой консоли к серверу и как ответ благополучно возвращается на игровую консоль.

Когда пакет, отправленный игровой консолью, достигнет маршрутизатора, модуль NAT запишет адрес и порт отправителя в новую строку в таблице NAT, выберет случайный, прежде не использовавшийся, номер порта для идентификации комбинации адреса и порта отправителя и запишет это число в ту же строку. Затем он заменит адрес и порт отправителя в пакете, записав свой глобальный IP-адрес и вновь выбранный номер порта. Измененный пакет достигнет сервера, который отправит ответ на глобальный IP-адрес маршрутизатора и вновь выбранный номер порта. Далее модуль NAT по номеру порта найдет первоначальные IP-адрес и порт отправителя, запишет их в пакет ответа и отправит нужному узлу.

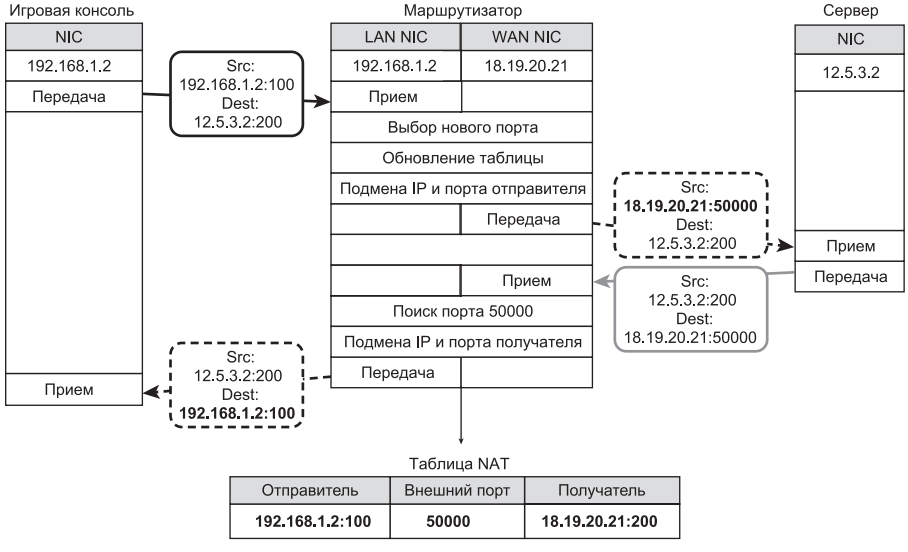


Рис. 2.20. Маршрутизатор с поддержкой NAT, подменяющий адрес и порт

ПРИМЕЧАНИЕ Для большей безопасности многие маршрутизаторы сохраняют в таблице NAT также IP-адрес и порт получателя. В этом случае, когда пакет с ответом вернется в маршрутизатор, модуль NAT сначала найдет в таблице запись, используя номер порта отправителя в пакете, а затем проверит совпадение IP-адреса и порта отправителя в пакете с ответом с IP-адресом и портом получателя в оригинальном исходящем пакете. Если они не совпадают, значит, происходит что-то подозрительное и пакет просто уничтожается.

Передача через NAT

Возможность преобразования сетевых адресов — чудесный подарок пользователям Интернета, но он может стать источником головной боли для разработчиков многопользовательских игр. Из-за большого количества пользователей, имеющих собственные локальные сети дома и использующих NAT для подключения компьютеров и игровых консолей к Интернету, довольно часто возникает ситуация, изображенная на рис. 2.21. Игрок А владеет узлом А, расположенным за маршрутизатором NAT А. Он хочет запустить на узле А игровой сервер так, чтобы его друг, игрок Б, мог подключаться к этому серверу. Игрок Б владеет узлом Б за маршрутизатором NAT Б. Из-за преобразования сетевых адресов игрок Б не сможет инициировать соединение с узлом А. Если узел Б пошлет пакет маршрутизатору, за которым находится узел А, пытаясь подключиться к игровому серверу, маршрутизатор NAT А не найдет соответствующую запись в своей таблице NAT и просто уничтожит пакет.

Существует несколько решений этой проблемы. Одно из них требует, чтобы игрок А вручную настроил на своем маршрутизаторе переадресацию портов.

Однако это не самое лучшее решение, и оно требует наличия у игрока технических знаний и навыков. Второе решение выглядит изящнее и безопаснее. Оно известно как *простое прохождение UDP через NAT* (Simple Traversal of UDP through NAT, STUN).

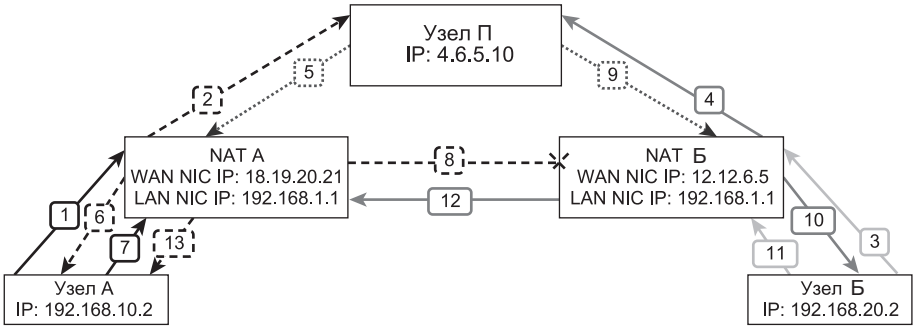


Рис. 2.21. Типичная сетевая конфигурация, используемая игроками

При использовании методики STUN узлы взаимодействуют с третьим узлом, таким как сервер Xbox Live или PlayStation Network. Третья сторона сообщает узлам, как инициировать соединение друг с другом, чтобы создать необходимые записи в таблицах NAT маршрутизаторов и продолжать взаимодействия напрямую. На рис. 2.22 показаны потоки взаимодействий, а на рис. 2.23 — пакеты, участвующие в обмене, и сгенерированные таблицы NAT. Допустим, что игра прослушивает UDP-порт 200, поэтому все взаимодействия с узлами, не являющимися маршрутизаторами, будут проходить через порт 200.

Рис. 2.22. Потоки данных при использовании методики STUN

Сначала узел А пошлет пакет с порта 200 посреднической службе с IP-адресом `4.6.5.10` (узел П), сообщая, что хотел бы зарегистрироваться в роли сервера. Когда пакет попадет на маршрутизатор NAT А, он создаст запись в своей таблице NAT и изменит пакет, заменив адрес отправителя своим глобальным IP-адресом, а порт отправителя — случайным числом, в нашем примере `60000`. Затем маршрутизатор NAT А отправит пакет узлу П. Узел П примет пакет и узнает, что игрок А, играющий на узле А с адресом `18.19.20.21:60000`, желает зарегистрироваться как сервер многопользовательской игры.

Действия

Номер пакета	Отправитель	Адрес отправителя	Адрес получателя	Получатель пакета	Результат
1	Узел А	192.168.10.2:200	4.6.5.10:200	NAT А	Создание строки 1 в таблице NAT А, NAT А изменит пакет
2	Узел А	18.19.20.21:60000	4.6.5.10:200	Узел П	Узел П зарегистрирует узел А как игровой сервер с адресом 18.19.20.21:60000
3	Узел Б	192.168.10.2:200	4.6.5.10:200	NAT Б	Создание строки 1 в таблице NAT Б, NAT Б изменит пакет
4	Узел Б	12.12.6.5:62000	4.6.5.10:200	Узел П	Узел П зарегистрирует узел Б как клиента с адресом 12.12.6.5:62000
5	Узел П	4.6.5.10:200	18.19.20.21:60000	NAT А	Соответствует строке 1 в таблице NAT А, NAT А изменит пакет
6	NAT А	4.6.5.10:200	192.168.10.2:200	Узел А	Узел А узнает глобальный адрес узла Б и посылает пакет
7	Узел А	192.168.10.2:200	12.12.6.5:62000	NAT А	Создание строки 2 в таблице NAT А, повторно используется порт из строки 1, NAT А изменит пакет
8	Узел А	18.19.20.21:60000	12.12.6.5:62000	NAT Б	NAT Б не ожидает получения пакета и сбрасывает его
9	Узел П	4.6.5.10:200	12.12.6.5:62000	NAT Б	Соответствует строке 1 в таблице NAT Б, NAT Б изменит пакет
10	NAT Б	4.6.5.10:200	192.168.20.2:200	Узел Б	Узел Б узнает глобальный адрес узла А и посылает пакет
11	Узел Б	192.168.20.2:200	18.19.20.21:60000	NAT Б	Создание строки 2 в таблице NAT Б, повторно используется порт из строки 1, NAT Б изменит пакет
12	NAT Б	12.12.6.5:62000	18.19.20.21:60000	NAT А	Соответствует строке 2 в таблице NAT А, NAT А изменит пакет
13	NAT А	12.12.6.5:62000	192.168.10.2:200	Узел А	Благополучная передача от узла Б до узла А

Таблица NAT А

№ п/п	Отправитель	Внешний порт	Получатель
1	192.168.10.2:200	60000	4.6.5.10:200
2	192.168.10.2:200	60000	12.12.6.5:62000

Таблица NAT Б

№ п/п	Отправитель	Внешний порт	Получатель
1	192.168.20.2:200	62000	4.6.5.10:200
2	192.168.20.2:200	62000	18.19.20.21:60000

Рис. 2.23. STUN-пакеты и таблицы NAT

Затем узел Б пошлет узлу П пакет, сообщающий, что игрок Б хотел бы подключиться к игре игрока А. Когда пакет попадет на маршрутизатор NAT Б, тот создаст запись в своей таблице NAT и изменит пакет, подобно тому как это сделал маршрутизатор А. Затем измененный пакет будет отправлен узлу П, который узнает из пакета, что узел Б с адресом `12.12.6.5:62000` хотел бы соединиться с узлом А. В этот момент узел П уже знает глобальный адрес маршрутизатора NAT А, а также порт получателя, через который маршрутизатор NAT А перешлет пакет узлу А. Узел П мог бы послать эту информацию узлу Б в пакете ответа и потребовать, чтобы узел Б попытался использовать ее для соединения. Но не забывайте, что некоторые маршрутизаторы проверяют происхождение входящих пакетов, чтобы гарантировать, что эти пакеты ожидаются с указанного адреса. Маршрутизатор NAT А пока ожидает получить только пакеты от узла П. Если узел Б сейчас попытается подключиться к узлу А, маршрутизатор NAT А заблокирует пакет, потому что он не ожидает получить ответ от узла Б.

К счастью, узел П также знает глобальный IP-адрес маршрутизатора NAT Б и номер порта, через который можно отправить пакет узлу Б. Поэтому он посылает данную информацию узлу А. Маршрутизатор NAT А пропускает эту информацию, потому что в его таблице NAT есть запись, сообщающая, что узел А ждет ответа от узла П. Затем узел А посылает пакет узлу Б, используя информацию, полученную от узла П. Может показаться странным, что сервер пытается соединиться с клиентом, тогда как обычно все делается наоборот. Еще более странной данная ситуация будет выглядеть, если вспомнить, что маршрутизатор NAT Б не ожидает получить пакет от узла А и потому просто не пропустит его. Зачем тогда напрасно посылать пакет? Таким способом узел А вынуждает маршрутизатор NAT А создать запись в своей таблице NAT!

Пакет, отправленный узлом А узлу Б, попадет в маршрутизатор NAT А. Маршрутизатор обнаружит, что адрес узла А `192.168.1.2:200` уже есть в таблице с внешним портом `60000`, поэтому он выберет этот порт для исходящего пакета. Затем создаст еще одну запись в таблице, указывающую, что с адреса `192.168.1.2:200` отправлен пакет на адрес `12.12.6.5:62000`. Эта дополнительная запись является ключевой. Пакет почти наверняка никогда не достигнет узла Б, но после его отправки узел П сможет послать ответ узлу Б, сообщая, что тот может подключиться непосредственно к узлу А по адресу `18.19.20.21:60000`. Узел Б так и поступит, и когда пакет достигнет маршрутизатора NAT А, тот увидит, что это действительно ожидаемый входящий пакет с адреса `12.12.6.5:62000`. Он изменит пакет, подставив адрес получателя `192.168.1.2:200`, и отправит его узлу А. С этого момента узлы А и Б смогут взаимодействовать непосредственно друг с другом, используя глобальные IP-адреса и номера портов, которыми они обменялись.

ПРИМЕЧАНИЕ Существует еще несколько интересных фактов о преобразовании сетевых адресов, которые стоит упомянуть. Во-первых, прием передачи через NAT, описанный выше, дает положительный результат не во всех случаях. Существуют такие реализации NAT, которые не присваивают согласованный номер порта внутреннему узлу. Эти реализации известны как *симметричный NAT* (symmetric NAT). В симметричном преобразовании сетевых адресов каждый исходящий пакет получает уникальный номер внешнего порта, даже если оригинальный IP-адрес и порт узла отправителя уже присутствуют в таблице NAT. Это обстоятельство мешает работе методики STUN, потому что маршрутизатор NAT А выберет новый внешний порт, когда узел А пошлет первый пакет

узлу Б. В этом случае, когда узел Б попытается связаться с маршрутизатором NAT А через внешний порт, который использовался при соединении узла А с узлом П, он не совпадет с портом в таблице NAT, и пакет будет сброшен.

Иногда менее безопасные реализации симметричного преобразования сетевых адресов присваивают внешние порты в строго определенном порядке, благодаря чему некоторые интеллектуальные программы могут *предсказать выбранный порт* и использовать STUN-подобную технологию для прохождения через симметричный NAT. Более безопасные реализации симметричного преобразования сетевых адресов выбирают порты случайным образом, и их поведение трудно предсказать.

Метод STUN можно применять только для пакетов UDP. Как описывается в главе 3 «Сокеты Беркли», протокол TCP использует другую систему присваивания портов и всегда пересылает данные через порт, отличающийся от порта, который прослушивается в ожидании входящих соединений. Для протокола TCP можно использовать прием, называемый *пробоем TCP* (TCP hole punching), который подходит для случаев, когда маршрутизатор с NAT поддерживает такую возможность. В документе RFC 5128, ссылка на который приводится в разделе «Для дополнительного чтения», дается отличный обзор приемов прохождения через NAT, включая пробой TCP.

Наконец, существует еще один популярный способ преодоления маршрутизатора с NAT. Он называется *интернет-протоколом устройства-шлюза* (Internet gateway device protocol, IGDP). Этот протокол реализуют некоторые маршрутизаторы с поддержкой *универсальной автоматической настройки* (Universal Plug and Play, UPnP), чтобы дать возможность узлам в локальной сети вручную настраивать отображение внутренних портов во внешние. Однако этот способ не всегда поддерживается и менее интересен с дидактической точки зрения, поэтому он не описывается здесь. Ссылка на его описание также приводится в разделе «Для дополнительного чтения».

В заключение

В этой главе был представлен обзор внутреннего устройства Интернета. Появление технологии коммутации пакетов, позволяющей осуществлять сразу множество передач по одной линии, дало начало проекту ARPANET и в конечном счете Интернету. Семейство протоколов TCP/IP — слоеный пирог, приводящий Интернет в действие, — состоит из пяти уровней, каждый из которых обеспечивает передачу данных вышележащего уровня.

Физический уровень соответствует среде-носителю, в которой распространяется сигнал, и иногда рассматривается как часть канального уровня, находящегося над ним. Канальный уровень предоставляет способ взаимодействия между соединенными узлами. Он требует наличия системы аппаратной адресации, чтобы каждый узел мог иметь уникальный адрес, и определяет MTU — максимальный объем данных, которые можно передать одним фрагментом. Существует множество протоколов, способных предоставлять услуги канального уровня, но в этой главе подробно был рассмотрен лишь протокол Ethernet как один из наиболее важных для разработчиков игр.

Сетевой уровень, предоставляющий систему логической адресации, опирающуюся на аппаратные адреса канального уровня, позволяет взаимодействовать узлам в разных сетях. IPv4 — основной протокол сетевого уровня на сегодняшний день — реализует системы прямой и косвенной маршрутизации, а также фрагментацию пакетов, слишком больших для канального уровня. IPv6 — недавно появившийся протокол — решает проблему ограниченности адресного пространства и оптимизирует некоторые из наиболее узких мест в IPv4, связанных с передачей данных.

Транспортный уровень с его портами обеспечивает непосредственные взаимодействия между процессами, выполняющимися на удаленных узлах. TCP и UDP — основные протоколы транспортного уровня — имеют фундаментальные отличия: UDP — легковесный, не создающий постоянных соединений и ненадежный протокол, тогда как TCP более тяжелый, требующий установки соединений с сохранением состояния и гарантирующий надежную передачу всех данных в заданном порядке. TCP реализует механизмы управления потоками данных и заторами, чтобы уменьшить потери пакетов.

На вершине пирога находится прикладной уровень, содержащий DHCP, DNS и программный код вашей игры.

Чтобы создавать локальные сети с минимальными усилиями, разработана технология преобразования сетевых адресов (NAT), позволяющая использовать один глобальный IP-адрес для подключения к Интернету целой сети. Недостатком NAT является блокирование незапрашивавшихся входящих соединений, которые могли бы быть желательны для серверов, но существуют методики, такие как STUN и пробой TCP, которые помогают решить эту задачу.

В этой главе была заложена теоретическая основа для работы с Интернетом. Сведения, которые приводятся здесь, пригодятся в главе 3, где рассказывается о функциях и структурах данных, используемых для реализации взаимодействий между узлами.

Вопросы для повторения

1. Перечислите пять уровней стека TCP/IP и кратко опишите каждый. Какой уровень не рассматривается как самостоятельный в некоторых моделях?
2. Для каких целей используется протокол ARP? Как он работает?
3. Объясните, как узел с несколькими сетевыми контроллерами (например, маршрутизатор) может передавать пакеты между разными подсетями. Расскажите, как устроена таблица маршрутизации.
4. Расшифруйте аббревиатуру MTU. Что это такое? Какова величина MTU в Ethernet?
5. Объясните, как осуществляется фрагментирование пакетов. Представьте канальный уровень с MTU, равным 400, опишите заголовок пакета, который мог бы быть фрагментирован на два фрагмента, а затем опишите заголовки этих фрагментов.
6. Почему желательно избегать IP-фрагментации?
7. Почему желательно стремиться отправлять пакеты максимального размера, которые могут быть переданы без фрагментации?
8. Чем отличаются надежная и ненадежная передача данных?
9. Опишите процедуру установки соединения, используемую протоколом TCP. Какие важные данные в этот момент передаются?
10. Опишите, как TCP обеспечивает надежную передачу данных.

11. Чем отличаются глобально маршрутизируемые и локально маршрутизируемые IP-адреса?
12. Что такое NAT? Какие преимущества дает NAT? Какие недостатки имеет?
13. Объясните, как клиент за NAT может отправить пакет глобально маршрутизируемому серверу и принять ответ.
14. Что такое STUN? Для чего может использоваться? Как это работает?

Для дополнительного чтения

Bell, Gordon. (1980, сентябрь). «The Ethernet — A Local Area Network». Доступно по адресу: *http://research.microsoft.com/en-us/um/people/gbell/ethernet_blue_book_1980.pdf*. Проверено 28 января 2016.

Braden R. (Ed). (1989, октябрь). «Requirements for Internet Hosts — Application and Support». Доступно по адресу: *http://tools.ietf.org/html/rfc1123*.[1] Проверено 28 января 2016.

Braden R. (Ed). (1989, октябрь). «Requirements for Internet Hosts — Communication Layers». Доступно по адресу: *http://tools.ietf.org/html/rfc1122*.[2] Проверено 28 января 2016.

Cotton, M., L. Eggert, J. Touch, M. Westerlund, and S. Cheshire. (2011, август). «Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry». Доступно по адресу: *http://tools.ietf.org/html/rfc6335*. Проверено 28 января 2016.

Deering, S., and R. Hinden. (1998, декабрь). «Internet Protocol, Version 6 (IPv6) Specification». Доступно по адресу: *https://www.ietf.org/rfc/rfc2460.txt*.[3] Проверено 28 января 2016.

Drom, R. (1997, март). «Dynamic Host Configuration Protocol». Доступно по адресу: *http://tools.ietf.org/html/rfc2131*. Проверено 28 января 2016.

Статистика Google по использованию IPv6. (2014, 9 август). Доступно по адресу: *https://www.google.com/intl/en/ipv6/statistics.html*. Проверено 28 января 2016.

Институт информатики (Information Sciences Institute). (1981, сентябрь). «Transmission Control Protocol». Доступно по адресу: *http://www.ietf.org/rfc/rfc793.txt*.[4] Проверено 28 января 2016.

«Internet Gateway Device Protocol». (2010, декабрь). Доступно по адресу: *http://upnp.org/specs/gw/igd2/*. Проверено 28 января 2016.

Mockapetris, P. (1987, ноябрь). «Domain Names — Concepts and Facilities». Доступно по адресу: *http://tools.ietf.org/html/rfc1034*.[5] Проверено 28 января 2016.

---

[1] Перевод на русский язык: *http://rfc.com.ru/rfc1123.htm. — Примеч. пер.*
[2] Перевод на русский язык: *http://rfc.com.ru/rfc1122.htm. — Примеч. пер.*
[3] Перевод на русский язык: *http://rfc2.ru/2460.rfc. — Примеч. пер.*
[4] Перевод на русский язык: *http://rfc.com.ru/rfc793.htm. — Примеч. пер.*
[5] Перевод на русский язык: *http://www.protocols.ru/WP/?p=738. — Примеч. пер.*

Mockapetris, P. (1987, ноябрь). «Domain Names — Implementation and Specification». Доступно по адресу: *http://tools.ietf.org/html/rfc1035*.[1] Проверено 28 января 2016.

Nagle, John. (1984, январь 6). «Congestion Control in IP/TCP Internetworks». Доступно по адресу: *http://tools.ietf.org/html/rfc896*. Проверено 28 января 2016.

Narten, T., E. Nordmark, W. Simpson, and H. Soliman. (2007, сентябрь). «Neighbor Discovery for IP version 6 (IPv6)». Доступно по адресу: *http://tools.ietf.org/html/rfc4861*. Проверено 28 января 2016.

Nichols, K., S. Blake, F. Baker, and D. Black. (1998, декабрь). «Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers». Доступно по адресу: *http://tools.ietf.org/html/rfc2474*.[2] Проверено 28 января 2016.

«Port Number Registry». (2014, сентябрь 3). Доступно по адресу: *http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml*. Проверено 28 января 2016.

Postel, J., and R. Reynolds. (1988, февраль). «A Standard for the Transmission of IP Datagrams over IEEE 802 Networks». Доступно по адресу: *http://tools.ietf.org/html/rfc1042*.[3] Проверено 28 января 2016.

Ramakrishnan, K., S. Floyd, and D. Black. (сентябрь 2001). «The Addition of Explicit Congestion Notification (ECN) to IP». Доступно по адресу: *http://tools.ietf.org/html/rfc3168*.[4] Проверено 28 января 2016.

Rekhter, Y., and T. Li. (1993, сентябрь). «An Architecture for IP Address Allocation with CIDR». Доступно по адресу: *http://tools.ietf.org/html/rfc1518*.[5] Проверено 28 января 2016.

Rosenberg, J., J. Weinberger, C. Huitema, and R. Mahy. (2003, март). «STUN — Simple Traversal of User Datagram Protocol (UDP)». Доступно по адресу: *http://tools.ietf.org/html/rfc3489*. Проверено 28 января 2016.

Socolofsky, T., and C. Kale. (1991, январь). «A TCP/IP Tutorial». Доступно по адресу: *http://tools.ietf.org/html/rfc1180*.[6] Проверено 28 января 2016.

[1] Перевод на русский язык: *http://www.protocols.ru/files/RFC/rfc1035.pdf. — Примеч. пер.*

[2] Перевод на русский язык: *http://www.protocols.ru/files/RFC/rfc2474.pdf. — Примеч. пер.*

[3] Перевод на русский язык: *http://rfc2.ru/1042.rfc. — Примеч. пер.*

[4] Перевод на русский язык: *http://rfc2.ru/3168.rfc. — Примеч. пер.*

[5] Перевод на русский язык: *http://rfc.com.ru/rfc1518.htm. — Примеч. пер.*

[6] Перевод на русский язык: *http://www.protocols.ru/WP/wp-content/uploads/2014/02/rfc1180.pdf. — Примеч. пер.*

3 Сокеты Беркли

В этой главе рассказывается о конструкции, наиболее широко используемой при разработке многопользовательских игр, — сокетах Беркли. Здесь будут представлены типичные функции создания сокетов, управления ими и уничтожения, рассмотрены различия между платформами, а также типизированная обертка для работы с сокетами, дружественная по отношению к C++.

Создание сокетов

Впервые вышедший в составе BSD 4.2, программный интерфейс *Berkeley Sockets API* обеспечил стандартный способ взаимодействия с разными уровнями стека TCP/IP. После своего появления он был реализован во всех основных операционных системах и наиболее популярных языках программирования, то есть стал настоящим стандартом в сетевом программировании.

Для доступа к этому программному интерфейсу процессы создают один или несколько *сокетов* (socket) и затем читают из них или записывают в них данные. Создать сокет можно вызовом функции с недвусмысленным названием `socket`:

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

Имя параметра `af` означает «address family», или «семейство адресов», и определяет протокол сетевого уровня, который должен использоваться сокетом. Возможные значения перечислены в табл. 3.1.

Таблица 3.1. Семейства адресов, используемые при создании сокетов

Макрос	Значение
`AF_UNSPEC`	Не определено
`AF_INET`	Internet Protocol Version 4 (IPv4)

Макрос	Значение
AF_IPX	AF_IPX Internetwork Packet Exchange: один из первых протоколов сетевого уровня, продвигавшийся компанией Novell и использовавшийся в MS-DOS
AF_APPLETALK	Appletalk: одно из ранних семейств протоколов сетевого уровня, продвигавшееся компанией Apple и использовавшееся на ее компьютерах Apple и Macintosh
AF_INET6	Internet Protocol Version 6 (IPv6)

Большинство современных игр написано с поддержкой IPv4, поэтому вам, скорее всего, придется использовать AF_INET. С увеличением числа пользователей, перешедших на IPv6, потребуется также реализовать поддержку сокетов AF_INET6.

Параметр type определяет назначение пакетов, посылаемых и принимаемых через сокет. Каждый протокол транспортного уровня, который может использоваться сокетом, имеет соответствующий способ группировки и использования пакетов. В табл. 3.2 перечислены наиболее часто используемые значения этого параметра.

Таблица 3.2. Значения параметра type, используемые при создании сокетов

Макрос	Значение
SOCK_STREAM	Пакеты представляют упорядоченные сегменты, поток надежной передачи данных
SOCK_DGRAM	Пакеты представляют отдельные датаграммы
SOCK_RAW	Заголовки пакетов могут подготавливаться прикладным уровнем
SOCK_SEQPACKET	Действует подобно SOCK_STREAM, но от получателя может потребоваться читать пакеты целиком после получения

Попытка создать сокет типа SOCK_STREAM сообщает операционной системе, что сокет требует создания постоянного соединения. В ответ система выделяет ресурсы, необходимые для поддержания надежного упорядоченного потока данных. Этот тип используется для создания сокетов TCP. Тип SOCK_DGRAM, напротив, не поддерживает постоянное соединение, и для него выделяется минимальный объем ресурсов, требуемых для отправки и приема отдельных датаграмм. Такой сокет не гарантирует ни надежность, ни доставку пакетов в каком-то определенном порядке. Этот тип используется для создания сокетов UDP.

Параметр protocol определяет конкретный протокол, который сокет должен использовать для передачи данных. Здесь можно указать один из протоколов транспортного уровня или один из вспомогательных протоколов сетевого уровня, входящих в семейство протоколов Интернета. Обычно значение, переданное в параметре protocol, копируется непосредственно в поле «Протокол» IP-заголовка каждого исходящего пакета. Оно указывает принимающей операционной системе, как интерпретировать данные, завернутые в пакет. В табл. 3.3 приводятся типичные значения для параметра protocol.

Таблица 3.3. Значения параметра protocol, используемые при создании сокетов

Макрос	Требуемое значение типа	Значение
IPPROTO_UDP	SOCK_DGRAM	В пакет завернута датаграмма UDP
IPPROTO_TCP	SOCK_STREAM	В пакет завернут сегмент TCP
IPPROTO_IP / 0	Любое	Используется протокол по умолчанию для указанного типа сокета

Обратите внимание, что, получив значение 0 в параметре protocol, ОС выберет реализацию протокола по умолчанию для указанного типа сокета. То есть сокет IPv4 UDP можно создать вызовом:

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

Сокет TCP — вызовом:

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

Закрыть сокет, независимо от его типа, можно вызовом функции closesocket:

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

Закрывать сокет TCP нужно после отправки всех данных и получения подтверждения их приема. Мы рекомендуем сначала прекратить отправку данных в сокет, затем дождаться подтверждения приема всех данных и прочитать все входные данные и только после этого закрывать сокет.

Прекратить отправку или прием перед закрытием можно с помощью функции shutdown:

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

В параметре how следует передать SD_SEND — чтобы остановить отправку, SD_RECEIVE — чтобы остановить прием или SD_BOTH — чтобы остановить отправку и прием. Передача значения SD_SEND после отправки всех данных вызовет отправку пакета FIN, который известит другую сторону соединения, что она может безопасно закрыть сокет. В ответ другая сторона пришлет свой пакет FIN. Как только игра получит пакет FIN, она может безопасно закрыть сокет.

Эта функция закрывает сокет и возвращает все задействованные ресурсы операционной системе. Обязательно закрывайте все сокеты, как только они становятся ненужными.

ПРИМЕЧАНИЕ В большинстве случаев операционная система автоматически создает IP-заголовки и заголовки транспортного уровня для всех пакетов, посылаемых через сокет. Но если создать сокет типа SOCK_RAW с протоколом 0, можно вручную заполнять поля заголовков обоих уровней. Такой прием позволит непосредственно определять поля заголовков, которые обычно недоступны для редактирования. Например, в каждом исходящем пакете можно указать собственное значение TTL: именно так действует утилита traceroute. Запись значений в различные поля

заголовков вручную часто является единственной возможностью вставить нестандартные значения, что может пригодиться, в частности, при тестировании серверов, как описывается в главе 10 «Безопасность».

Поскольку сокеты типа SOCK_RAW позволяют произвольно изменять значения полей заголовков, они представляют определенную угрозу безопасности и большинство операционных систем разрешают создавать такие сокеты только программам с повышенными привилегиями.

Различия в API операционных систем

Даже при том, что сокеты Беркли являются стандартом низкоуровневого интерфейса доступа к Интернету на разных платформах, фактические реализации API в разных операционных системах имеют отличия. Существует несколько отличий и особенностей, которые важно знать перед началом разработки кроссплатформенного кода, использующего сокеты.

Первое из них — тип данных, используемый для представления самих сокетов. Функция `socket`, как можно было видеть выше, возвращает результат типа `SOCKET`, но в действительности этот тип существует только на платформах на основе Windows, таких как Windows 10 и Xbox. Немного покопавшись в заголовочных файлах Windows, можно заметить, что `SOCKET` определяется как `typedef` от `UINT_PTR`. То есть это указатель на область памяти, где хранится информация о сокете и его состоянии.

На POSIX-совместимых платформах, таких как Linux, Mac OS X и PlayStation, сокет — простое значение типа `int`. По сути, нет никакого типа данных с названием «сокет»: функция `socket` возвращает целое число. Это число представляет индекс в списке открытых файлов и сокетов операционной системы. То есть сокет очень похож на дескриптор файла, определяемый стандартом POSIX, и фактически может передаваться многим системным функциям, принимающим такие дескрипторы. Подобный подход к использованию сокетов лишает их некоторой доли гибкости, которую дают специализированные функции для работы с сокетами, но иногда существенно упрощает реализацию поддержки сети в прежде несетевых процессах. Одним существенным недостатком функции `socket`, возвращающей `int`, является недостаточная безопасность типов, так как компилятор не сможет воспрепятствовать передаче любого целочисленного выражения (например, `5 × 4`) в функцию, принимающую сокет в виде параметра. Некоторые примеры кода в этой главе решают данную проблему, так как она является универсальной для программного интерфейса сокетов Беркли на всех платформах.

Независимо от того, как платформа представляет сокет — как `int` или как `SOCKET`, — важно помнить, что сокеты всегда должны передаваться функциям *по значению*.

Второе важное различие между платформами заключается в имени заголовочного файла, содержащего определения для использования библиотеки. В Windows библиотека для работы с сокетами называется Winsock2, и потому программные модули, использующие сокеты, должны подключать заголовочный файл `WinSock2.h`. В Windows существует более старая библиотека с названием Winsock, и эта версия фактически подключается по умолчанию в файле `Windows.h`, который используется большинством программ для Windows. Библиотека Winsock была написана давно

и представляет собой более ограниченную и менее оптимизированную версию библиотеки WinSock2, но она содержит объявления некоторых основных библиотечных функций, таких как функция создания сокета, обсуждавшаяся выше. В результате возникает конфликт имен между `Windows.h` и `WinSock2.h`, подключаемых в одной единице компиляции: многократные объявления одних и тех же функций заставляют компилятор извергать сообщения об ошибках, сбивающие с толку тех, кто не знает о существовании этого конфликта. Чтобы избежать этого, необходимо либо подключать `WinSock2.h` перед `Windows.h`, либо определить макрос `WIN32_LEAN_AND_MEAN` перед подключением `Windows.h`. Макрос заставит препроцессор, кроме всего прочего, исключить директиву подключения Winsock из списка файлов, содержащихся в `Windows.h`, и тем самым предотвратить конфликт.

Файл `WinSock2.h` содержит только объявления функций и типов данных, имеющих прямое отношение к сокетам. Для получения объявлений других сетевых функций необходимо подключить другие файлы. Например, чтобы использовать функции преобразования адресов, рассматриваемые далее в этой главе, следует подключить `Ws2tcpip.h`.

В POSIX-совместимых платформах имеется только одна версия библиотеки для работы с сокетами, обычно доступная путем подключения файла `sys/socket.h`. Для использования функций для IPv4 подключите `netinet/in.h`. Чтобы использовать функции преобразования адресов, следует подключить `arpa/inet.h`. Для разрешения имен вам понадобится подключить `netdb.h`.

Инициализация и прекращение использования библиотеки поддержки сокетов на разных платформах также осуществляются по-разному. В POSIX-совместимых платформах библиотека активизируется автоматически и не требует дополнительных действий перед началом работы с сокетами. Библиотека Winsock2, напротив, требует явной инициализации и завершения и дает пользователю возможность указывать версию используемой библиотеки. Чтобы активировать библиотеку сокетов в Windows, используйте `WSAStartup`:

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

`wVersionRequested` — 2-байтное слово, младший байт которого определяет старший номер версии, а старший байт — младший номер желаемой версии Winsock. На момент выхода этой книги последней была версия с номером 2.2, который можно передать в этом параметре как `MAKEWORD(2, 2)`.

`lpWSAData` — указатель на структуру данных, которая будет заполнена с помощью функции `WSAStartup` информацией об активированной библиотеке, включая номер версии предоставленной реализации. Обычно он совпадает с затребованным номером версии и, как правило, не требует проверки.

`WSAStartup` возвращает либо `0`, сообщая об успехе, либо код ошибки, определяющий причину невозможности активировать библиотеку. Следует отметить, что ни одна из функций Winsock2 не будет работать правильно, если вызов `WSAStartup` не завершится успехом.

Чтобы завершить использование библиотеки, вызовите `WSACleanup`:

```
int WSACleanup();
```

`WSACleanup` не имеет параметров и возвращает код ошибки. Когда процесс вызывает `WSACleanup`, все ожидающие операции с сокетами завершаются и освобождаются все выделенные для них ресурсы, поэтому всегда старайтесь закрывать все сокеты перед завершением использования Winsock. Вызовы `WSAStartup` подсчитываются системой, поэтому `WSACleanup` должна вызываться столько же раз, сколько вызывалась `WSAStartup`, чтобы гарантировать освобождение всех ресурсов.

Обработка ошибок на разных платформах также различается. В случае ошибки большинство функций на всех платформах возвращают `-1`. В Windows вместо ничего не говорящего числа `-1` можно использовать макрос `SOCKET_ERROR`. Поскольку число `-1` не несет информации о причине ошибки, в Winsock2 предусмотрена функция `WSAGetLastError`, с помощью которой можно получить дополнительный код, помогающий определить фактическую ошибку:

```
int WSAGetLastError();
```

Эта функция возвращает код только самой последней ошибки, сгенерированной текущим потоком выполнения, поэтому его важно получить сразу же, как только библиотечная функция вернет `-1`. Вызов другой библиотечной функции сразу после ошибки, скорее всего, завершится ошибкой, обусловленной первой ошибкой. Это может изменить результат, возвращаемый функцией `WSAGetLastError`, и замаскировать истинную причину проблемы.

POSIX-совместимые библиотеки также дают возможность получить дополнительную информацию об ошибке. Но для этого они используют глобальную переменную `errno` из стандартной библиотеки C. Чтобы иметь возможность проверять значение `errno`, подключите файл `errno.h`. После этого к `errno` можно обращаться как к любой другой переменной. Подобно результату, возвращаемому функцией `WSAGetLastError`, значение `errno` может изменяться после вызова каждой функции, поэтому важно проверять ее сразу после получения признака ошибки.

СОВЕТ Большинство платформенно-независимых функций в библиотеке поддержки сокетов имеют имена, состоящие только из букв нижнего регистра, например `socket`. Однако большинство Windows-функций в библиотеке Winsock2 начинаются с буквы верхнего регистра и иногда имеют приставку `WSA`, показывающую, что они являются нестандартными. Разрабатывая программы для Windows, старайтесь использовать функции из Winsock2 отдельно от кроссплатформенных, чтобы последующий перенос на POSIX-совместимые платформы вызывал меньше сложностей.

В библиотеке Winsock2 присутствуют функции, не имеющие аналогов в POSIX-совместимой версии библиотеки сокетов Беркли, так же как многие POSIX-совместимые операционные системы имеют собственные платформенно-зависимые сетевые функции в дополнение к стандартным. Стандартные функции для работы с сокетами предоставляют все необходимое для типичной многопользовательской сетевой игры, поэтому в оставшейся части этой главы мы будем исследовать только стандартные кроссплатформенные функции. Примеры кода для этой книги ориентированы на операционную систему Windows, но нестандартные функции из Winsock2 используются в них только в том случае, когда необходимо провести инициализацию и остановку библиотеки, а также для проверки ошибок. Для функций, отличающихся на разных платформах, будет приводиться несколько версий кода.

Адрес сокета

Любой пакет сетевого уровня должен содержать адрес отправителя и получателя. Если данные обертываются в пакет транспортного уровня, такой пакет также должен содержать порт отправителя и получателя. Для передачи этой адресной информации в библиотеку или из библиотеки в программном интерфейсе определяется тип данных `sockaddr`:

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

`sa_family` хранит константу, определяющую тип адреса. При использовании этого адреса с сокетом значение в поле `sa_family` должно совпадать со значением параметра `af`, использованного для создания сокета. Поле `sa_data` имеет размер 14 байт и хранит фактический адрес. Оно определено как обобщенный массив байтов, поскольку предоставляет возможность сохранять адреса в разных форматах в зависимости от используемого семейства адресов. Технически это поле можно заполнять вручную, побайтно, но для этого требуется знать форматы представления адресов разных семейств. Чтобы избавить программиста от этой необходимости, в API имеются специализированные типы данных, помогающие инициализировать адреса из наиболее распространенных семейств адресов. Поскольку на момент разработки программного интерфейса не существовало ни классов, ни механизма полиморфного наследования, эти типы данных следует вручную приводить к типу `sockaddr` при передаче любой библиотечной функции, требующей адрес. Чтобы создать адрес для пакета IPv4, используйте тип `sockaddr_in`:

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

Поле `sin_family` накладывается на поле `sa_family` в структуре `sockaddr` и, соответственно, имеет тот же смысл.

Поле `sin_port` хранит 16-битную секцию адреса с номером порта.

Поле `sin_addr` хранит 4-байтный адрес IPv4. Тип `in_addr` по-разному объявляется в разных библиотеках. В некоторых платформах это простое 4-байтное целое число. Но, как правило, адреса IPv4 записываются не как 4-байтные целые числа, а как четыре отдельных байта, разделенных точками. По этой причине другие платформы определяют структуру, обертывающую объединение структур, которые можно использовать для интерпретации адресов в разных форматах:

```
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;
};
```

Используя поля `s_b1`, `s_b2`, `s_b3` и `s_b4` структуры `S_un_b` в объединении `S_un`, можно ввести адрес, указанный в удобочитаемой форме.

Поле `sin_zero` не используется и существует только для приведения размера `sockaddr_in` в соответствие с размером `sockaddr`. Для непротиворечивости все его элементы должны заполняться нулями.

СОВЕТ Создавая экземпляр любой структуры, имеющей отношение к сокетам BSD, мы рекомендуем использовать `memset` для обнуления всех ее полей. Это позволит избежать кроссплатформенных ошибок, вызванных неинициализированными полями в том случае, если одна платформа использует поля, которые игнорируются другой платформой.

Устанавливая IP-адрес как 4-байтное целое или номер порта, важно учитывать, что реализация TCP/IP и аппаратная платформа могут использовать разные стандарты упорядочения байтов в многобайтных числах. В главе 4 «Сериализация объектов» более детально обсуждается проблема упорядочения байтов на разных платформах, а пока просто запомните, что любые многобайтные числа в структуре, представляющей адрес сокета, должны преобразовываться из аппаратного порядка следования байтов, используемого на данном компьютере, в сетевой. С этой целью в программном интерфейсе сокетов имеются функции `htons` и `htonl`:

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

Функция `htons` принимает произвольное 16-битное целое без знака с аппаратным порядком следования байтов, используемым на данном компьютере, и возвращает то же число с сетевым порядком следования байтов. Функция `htonl` выполняет ту же операцию с 32-битными целыми.

На платформах, где аппаратный и сетевой порядок следования байтов совпадают, эти функции ничего не делают. Если включены соответствующие оптимизации, компилятор будет распознавать этот факт и опускать вызовы функций, не генерируя лишнего кода. На платформах, где аппаратный порядок следования байтов отличается от сетевого, возвращаемые значения будут состоять из тех же байтов, что и входные параметры, но с измененным порядком их следования. То есть если вы используете такую платформу и применяете отладчик для исследования поля `sa_port` инициализированной структуры `sockaddr_in`, десятичное значение, представляемое отладчиком, не будет совпадать с фактически используемым номером порта. Вместо этого вы увидите десятичное значение версии порта, в которой байты переставлены местами.

Иногда, например при приеме пакета, структура `sockaddr_in` автоматически заполняется программным интерфейсом сокетов. В этом случае поля в `sockadd_in` будут хранить значения с сетевым порядком следования байтов. В этом случае, если понадобится извлечь их и как-то обработать, используйте функции `ntohs` и `ntohl` для преобразования значений из формата с сетевым порядком следования байтов в формат с аппаратным порядком:

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

Эти две функции действуют подобно своим *host-to-network* (из аппаратного — в сетевой) аналогам.

В листинге 3.1 демонстрируется создание адреса сокета, представляющего порт `80` и IP-адрес `65.254.248.180`.

Листинг 3.1. Инициализация структуры 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;
```

ПРИМЕЧАНИЕ Некоторые платформы добавляют в структуру `sockaddr` дополнительное поле для хранения размера используемой структуры. Это дает возможность увеличить размер структуры `sockaddr` в будущем. На таких платформах достаточно просто записать в поле результат функции `sizeof`, возвращающей размер структуры. Например, инициализировать это поле в экземпляре структуры `sockaddr_in` с именем `myAddr` в Mac OS X можно с помощью инструкции `myAddr.sa_len = sizeof(sockaddr_in)`.

Безопасность типов

Поскольку при создании библиотеки поддержки сокетов безопасности типов почти не уделялось внимания, может пригодиться прием обертывания базовых типов данных сокетов и функций нестандартными объектно-ориентированными конструкциями, реализованными на уровне приложения. Это также помогает отделить программный интерфейс сокетов от игры и пригодится, если вдруг позднее вы решите заменить библиотеку сокетов альтернативной сетевой библиотекой. В этой книге мы создадим обертки для многих структур и функций, чтобы показать, как правильно использовать низкоуровневый API и обеспечить более надежное типизированное основание, которое вы сможете использовать в качестве фундамента для строительства своего кода. В листинге 3.2 представлена обертка для структуры `sockaddr`.

Листинг 3.2. Класс SocketAddress

```
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;
```

Класс `SocketAddress` имеет два конструктора. Первый принимает 4-байтный адрес IPv4 и порт и присваивает их значения полям внутренней структуры `sockaddr`. Конструктор автоматически выбирает семейство адресов `AF_INET`, потому что его параметры имеют смысл только для адресов IPv4. Для поддержки IPv6 можно дополнить этот класс еще одним конструктором.

Второй конструктор принимает готовую структуру `sockaddr` и копирует ее содержимое во внутреннее поле `mSockAddr`. Он может пригодиться для случаев, когда сетевой программный интерфейс возвращает структуру `sockaddr` и ее нужно завернуть в класс `SocketAddress`.

Вспомогательный метод `GetSize` класса `SocketAddress` обеспечивает читаемость кода при использовании функций, требующих размер структуры `sockaddr`.

Наконец, тип разделяемого указателя на адрес сокета гарантирует простоту передачи адреса без необходимости заботиться об освобождении памяти. На данный момент `SocketAddress` обладает незначительной дополнительной функциональностью, но он обеспечивает хорошую основу для дальнейшего ее наращивания по мере появления новых потребностей в будущих примерах.

Инициализация sockaddr из строки

Чтобы записать IP-адрес и номер порта в адрес сокета, требуется приложить некоторые усилия, особенно если информация об адресе передается программе в виде строки, прочитанной из конфигурационного файла, или из командной строки. Если имеется строка, которую требуется преобразовать в структуру `sockaddr`, эту

работу можно поручить функции `inet_pton` в POSIX-совместимых системах или `InetPton` — в Windows.

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

Обе функции принимают семейство адресов — `AF_INET` или `AF_INET6` — и преобразуют строковое представление IP-адреса в представление `in_addr`. Параметр `src` должен указывать на строку, завершающуюся нулевым символом, содержащую адрес в точечной нотации, а параметр `dst` — на поле `sin_addr` в требуемой структуре `sockaddr`. В случае успеха функция возвращает `1`; если исходная строка оформлена с ошибками, возвращается `0`; если возникла какая-то другая системная ошибка, возвращается `-1`. Листинг 3.3 демонстрирует, как инициализировать `sockaddr` с использованием одной из этих *presentation-to-network* (из строкового представления в сетевое) функций.

Листинг 3.3. Инициализация sockaddr с помощью InetPton

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

Чтобы `inet_pton` преобразовала строковое представление в двоичный IP-адрес, строка должна содержать именно IP-адрес. Строка не может быть доменным именем, так как функция не выполняет поиск в DNS. Если потребуется выполнить простой запрос к DNS, чтобы преобразовать доменное имя в IP-адрес, используйте `getaddrinfo`:

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

Параметр `hostname` должен быть строкой, завершающейся нулевым символом и содержащей искомое доменное имя. Например, `"live-shore-986.herokuapp.com"`.

Параметр `servname` должен быть строкой, завершающейся нулевым символом и содержащей номер порта или имя службы, отображаемое в номер порта. Например, в этом параметре можно передать `"80"` или `"http"`, чтобы получить структуру `sockaddr_in` с номером порта `80`.

Параметр `hints` должен являться указателем на структуру `addrinfo` с информацией о желаемом результате. Здесь можно указать желаемое семейство адресов или другие требования, можно также передать `nullptr`, чтобы получить все соответствующие результаты.

Наконец, параметр `res` должен быть указателем на переменную, в которую функция запишет указатель на начало связанного списка с вновь созданными структурами `addrinfo`. Каждая структура `addrinfo` в этом списке представляет раздел в ответе, полученном от сервера DNS:

```
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;
}
```

Поля `ai_flags`, `ai_socktype` и `ai_protocol` используются для запроса определенных результатов, когда `addrinfo` передается в `getaddrinfo` в параметре `hint`. В ответах эти поля можно игнорировать.

Поле `ai_family` сообщает семейство адресов, к которому принадлежит данная структура `addrinfo`. Значение `AF_INET` соответствует адресу IPv4, а значение `AF_INET6` — адресу IPv6.

Поле `ai_addrlen` сообщает размер структуры `sockaddr`, на которую указывает `ai_addr`.

Поле `ai_canonname` хранит каноническое имя опознанного узла, если в поле `ai_flags` структуры `addrinfo`, переданной в параметре `hints`, был установлен флаг `AI_CANONNAME`.

Поле `ai_addr` хранит указатель на структуру `sockaddr` для заданного семейства адресов, которая адресует узел с портом, указанные в параметрах `hostname` и `servname`.

Поле `ai_next` указывает на следующую структуру `addrinfo` в связанном списке. Так как доменное имя может отображаться в несколько адресов IPv4 и IPv6, нужно обойти связанный список, чтобы найти структуру `sockaddr`, соответствующую вашим потребностям. Как вариант, можно установить соответствующее значение в поле `ai_family` в структуре `addrinfo` и получать результаты только для желаемого семейства адресов. Последняя структура `addrinfo` в списке будет хранить `nullptr` в поле `ai_next`, показывая, что это последний элемент списка.

Поскольку `getaddrinfo` выделяет память для одной или нескольких структур `addrinfo`, после сохранения копии выбранной структуры `sockaddr` следует вызвать `freeaddrinfo` для освобождения памяти:

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

В `ai` должен передаваться указатель на самую первую структуру `addrinfo`, возвращаемый функцией `getaddrinfo`. Функция обойдет связанный список и освободит память, занимаемую всеми узлами `addrinfo` в списке и соответствующими им буферами.

Чтобы определить IP-адрес по имени узла, `getaddrinfo` создает пакет протокола DNS и посылает его по протоколу UDP или TCP одному из серверов DNS, настроенных в операционной системе. Затем она ждет ответа, анализирует его, конструирует связанный список структур `addrinfo` и возвращает его вызывающей

программе. Так как в процессе присутствуют операции отправки информации удаленному узлу и приема ответа от него, на его выполнение может потребоваться значительное время. Иногда ожидание может исчисляться миллисекундами, но чаще — секундами. `getaddrinfo` не имеет возможности использовать асинхронный режим работы, поэтому она будет блокировать работу вызывающего потока выполнения до получения ответа. Задержки могут вызывать неудовольствие у пользователей, поэтому, если потребуется получать IP-адреса по именам узлов в сети, попробуйте вынести вызов `getaddrinfo` в отдельный поток выполнения, отличный от основного потока выполнения игры. В Windows есть возможность использовать специализированную функцию `GetAddrInfoEx`, которая позволяет осуществлять асинхронные операции без создания отдельного потока выполнения.

Возможности функции `getaddrinfo` можно инкапсулировать в класс `SocketAddressFactory`, представленный в листинге 3.4.

Листинг 3.4. Разрешение имен с помощью 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;
      // использовать порт по умолчанию...
      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;
  }
};
```

Класс `SocketAddressFactory` имеет единственный статический метод для создания экземпляра `SocketAddress` из строкового представления имени узла и порта. Метод возвращает указатель `SocketAddressPtr`, то есть имеет возможность вернуть `nullptr` в случае неудачной попытки преобразовать имя. Это отличная альтернатива использованию конструктора `SocketAddress` для преобразования, так как, не требуя обрабатывать исключения, метод гарантирует, что никогда не появится неправильно инициализированный экземпляр `SocketAddress`: если `CreateIPv4FromString` вернет непустой указатель, значит, он будет указывать на допустимый экземпляр `SocketAddress`.

Метод сначала отделяет порт от имени, отыскивая двоеточие. Затем создает структуру-подсказку `addrinfo`, чтобы гарантировать получение только результатов с адресами IPv4. После этого он передает всю информацию в вызов `getaddrinfo` и выполняет обход списка с результатами, пока не найдет непустой адрес. Найденный адрес копируется в новый экземпляр `SocketAddress` с помощью соответствующего конструктора, и затем связанный список освобождается. Если что-то пойдет не так, возвращается `nullptr`.

Связывание сокета

Процедура извещения операционной системы о том, что сокет будет использоваться с определенным адресом и портом транспортного уровня, называется *связыванием* (binding). Чтобы вручную связать сокет с адресом и портом, можно использовать функцию `bind`:

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

`sock` — это связываемый сокет, прежде созданный функцией `socket`.

`address` — это адрес, с которым должен быть связан сокет. Обратите внимание, что этот адрес не имеет отношения к адресу, по которому сокет будет отправлять пакеты. Фактически этот параметр определяет адрес отправителя для любых пакетов, посылаемых сокетом. Сама необходимость определять адрес отправителя может показаться странной, потому что все пакеты, посылаемые узлом, с очевидностью исходят с адреса этого узла. Но не спешите с выводами, вспомните, что узел может иметь несколько сетевых интерфейсов и каждый из них может иметь собственный IP-адрес.

Передача определенного адреса в функцию `bind` позволяет указать, какой интерфейс должен использоваться сокетом. Это особенно полезно для узлов, действующих в роли маршрутизаторов или мостов между сетями, так как их интерфейсы могут быть подключены к разным группам компьютеров. В многопользовательских играх редко приходится указывать сетевой интерфейс и намного важнее привязать указанный порт ко всем доступным сетевым интерфейсам и IP-адресам, которые имеет узел. Для этого можно присвоить макроопределение `INADDR_ANY` полю `sin_addr` структуры `sockaddr_in`, передаваемой в вызов `bind`.

`address_len` должен содержать размер `sockaddr` в параметре `address`.

В случае успеха `bind` возвращает `0`, в случае ошибки `-1`.

Операция связывания сокета с `sockaddr` преследует две цели. Во-первых, она сообщает ОС, что этот сокет должен использоваться для приема любых входящих пакетов с адресом получателя, совпадающим с адресом и портом сокета. Во-вторых, определяет адрес и порт отправителя, которые библиотека поддержки сокетов должна использовать при создании заголовков сетевого и транспортного уровней для пакетов, посылаемых через сокет.

Обычно с заданным адресом и портом можно связать только один сокет. В ответ на попытку связать сокет с адресом и портом, которые уже используются, `bind` вернет признак ошибки. В этом случае можно повторять попытки связать сокет с другими портами, пока не будет найден свободный порт. Для автоматизации этого процесса в качестве порта можно указать число `0`. В этом случае библиотека сама найдет неиспользуемый порт и свяжет сокет с ним.

Сокет должен быть связан прежде, чем его можно будет использовать для передачи и приема данных. По этой причине, если процесс попытается послать данные, используя несвязанный сокет, библиотека автоматически свяжет его со свободным портом. То есть вызывать `bind` вручную имеет смысл, только если необходимо связать сокет с определенным адресом и портом. Это нужно серверам, которые должны принимать пакеты, прослушивая общеизвестный порт, но клиентам это обычно не требуется.

Клиент может использовать любой автоматически связанный порт: когда он посылает серверу первый пакет, этот пакет будет содержать автоматически выбранный адрес и порт отправителя, и сервер сможет использовать этот адрес, чтобы вернуть пакет с ответом.

Сокеты UDP

Через сокет UDP можно отправлять данные сразу после его создания. Если сокет не был предварительно связан, сетевой модуль найдет свободный порт в динамической области и автоматически привяжет его. Для отправки данных можно использовать функцию `sendto`:

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

`sock` — сокет, через который должна быть отправлена датаграмма. Если сокет не связан, библиотека автоматически свяжет его с доступным портом. Адрес и порт, связанные с сокетом, будут использоваться как адрес отправителя в заголовках исходящего пакета.

`buf` — указатель на начало буфера с данными. Буфер не обязательно должен иметь тип `char*`. Он может иметь любой тип при условии возможности приведения его к типу `char*`. По этой причине удобнее считать, что этот параметр имеет тип `void*`.

`len` — объем (длина буфера) посылаемых данных. Технически, размер датаграммы UDP, включая 8-байтный заголовок, не может превышать 65 535 байт, потому что поле «размер» в заголовке может хранить только 16 бит. Но не забывайте, что наибольший размер пакета, который может быть отправлен без фрагментации, определяется значением MTU канального уровня. Для Ethernet величина MTU составляет 1500 байт, но в этот объем входят не только фактические данные, посылаемые игрой, но и несколько заголовков и любые обертки пакетов. Как создатель игры старайтесь избегать фрагментации. Хорошее эмпирическое правило: не используйте датаграммы размером более 1300 байт.

`flags` — битовая коллекция флагов, объединенных поразрядной операцией ИЛИ (OR), управляющих передачей данных. Для большинства игр в этом параметре следует передавать `0`.

`to` — структура `sockaddr` с адресом получателя. Семейство адресов в этой структуре `sockaddr` должно совпадать с семейством, использовавшимся при создании сокета. Адрес и порт из параметра `to` будут скопированы в IP- и UDP-заголовки как IP-адрес и порт получателя.

`len` — размер структуры `sockaddr` в параметре `to`. Для IPv4 достаточно передать выражение `sizeof(sockaddr_in)`.

В случае успеха `sendto` вернет объем данных, добавленных в очередь для передачи. В противном случае она вернет `-1`. Обратите внимание: ненулевой положительный результат вовсе не означает, что датаграмма была отправлена, он лишь свидетельствует, что она добавлена в очередь для передачи.

Чтобы прочитать данные из сокета UDP, достаточно вызвать функцию `recvfrom`:

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

`sock` — сокет, откуда требуется прочитать данные. По умолчанию, если другая сторона ничего не посылала в сокет, поток выполнения заблокируется, пока сокетом не будет получена первая датаграмма.

`buf` — буфер для приема датаграммы. По умолчанию, скопировав датаграмму в буфер, указанный в вызове `recvfrom`, библиотека сокетов уничтожит свою копию.

`len` — должен определять максимальное число байтов, которое можно сохранить в буфер, указанный в параметре `buf`. Чтобы избежать ошибки переполнения буфера, `recvfrom` никогда не будет копировать данных больше, чем определено этим параметром. Любые байты, оставшиеся во входящей датаграмме, будут потеряны

навсегда, поэтому всегда используйте приемные буферы достаточного размера, чтобы вместить наибольшую из возможных датаграмм.

`flags` — битовая коллекция флагов, объединенных поразрядной операцией ИЛИ (OR) и управляющих приемом данных. В большинстве игр этот параметр должен быть нулем. Иногда может пригодиться флаг `MSG_PEEK`. При его наличии принятая датаграмма будет скопирована в буфер `buf`, и ее копия останется во входной очереди. То есть следующий вызов `recvfrom`, возможно с бóльшим буфером, сможет повторно извлечь ту же самую датаграмму.

`from` — должен быть указателем на структуру `sockaddr`, куда функция `recvfrom` запишет адрес и порт отправителя. Обратите внимание, что эту структуру не требуется инициализировать заранее какой-либо адресной информацией. Бытует ошибочное мнение, что ее (структуру) можно использовать, чтобы запросить пакет с указанного адреса, инициализировав данный параметр, но в действительности это невозможно. Датаграммы возвращаются функцией `recvfrom` в порядке их получения, и структура, на которую указывает `from`, заполняется соответствующим адресом отправителя датаграммы.

`fromlen` — должен указывать на целочисленную переменную, содержащую размер структуры `sockaddr`, на которую указывает `from`. Функция `recvfrom` может уменьшить это значение, если ей не потребуется все пространство для копирования адреса отправителя.

В случае успеха `recvfrom` вернет число байтов, скопированных в `buf`. В случае ошибки она вернет `-1`.

Безопасность типов сокетов UDP

В листинге 3.5 демонстрируется класс обертки `UDPSocket`, способный связывать сокет с адресом, а также отправлять и принимать датаграммы.

Листинг 3.5. Класс UDPSocket

```
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
  {
    // вернуть код ошибки как отрицательное число
    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);
}
```

Класс `UDPSocket` имеет три основных метода: `Bind`, `SendTo` и `ReceiveFrom`. Каждый использует класс `SocketAddress`, который был определен выше. Чтобы такое было возможно, класс `SocketAddress` должен объявить `UDPSocket` дружественным клас-

сом (friend class), иначе приватная переменная-член `sockaddr` будет недоступна его методам. Такое определение `SocketAddress` гарантирует, что никакой код за пределами модуля обертки сокета не сможет изменить `sockaddr` напрямую, что страхует от потенциальных ошибок.

Ценной особенностью объектно-ориентированной обертки является возможность задания деструкторов. В данном случае `~UDPSocket` автоматически закроет внутренний сокет и предотвратит утечку памяти.

Определение `UDPSocket` в листинге 3.5 выносит зависимость от механизма вывода сообщений об ошибках в класс `SocketUtil`. Отделение механизма вывода сообщений упрощает изменение обработки ошибок в будущем и скрывает тот факт, что для получения кода ошибки на одних платформах нужно вызвать `WSAGetLastError`, а на других — просто прочитать переменную `errno`.

Программный код не позволяет создать `UDPSocket` на пустом месте. Единственный конструктор `UDPSocket` объявлен приватным. По аналогии с классом `SocketAddressFactory` это решение не позволяет создать экземпляр `UDPSocket` с недопустимым значением `mSocket` внутри. Вместо конструктора предлагается использовать метод `SocketUtil::CreateUDPSocket`, представленный в листинге 3.6, который создаст экземпляр `UDPSocket` только после успешного вызова `socket`.

Листинг 3.6. Создание сокета UDP

```
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;
  }
}
```

Сокеты TCP

Протокол UDP не предполагает создания постоянного соединения и хранения информации о его состоянии. Это ненадежный протокол, и потому для отправки и приема датаграмм узлу достаточно иметь единственный сокет. Протокол TCP, напротив, является надежным протоколом и требует установки постоянного соединения между двумя узлами перед началом передачи данных. Кроме того, он

должен иметь информацию о состоянии соединения, чтобы при необходимости повторять передачу потерявшихся пакетов, и где-то хранить эту информацию. В программном интерфейсе сокетов Беркли информация о состоянии соединения хранится в самом сокете. Это означает, что для создания каждого TCP-соединения узел должен создать свой сокет.

Инициализация TCP-соединения между клиентом и сервером выполняется в три этапа. Чтобы выполнить первый этап, сервер должен сначала создать сокет, связать его с требуемым портом и затем прослушивать сокет в ожидании входящих запросов на соединение. После создания и связывания сокета с помощью `socket` и `bind` сервер может запустить прослушивание вызовом функции `listen`:

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

`sock` — сокет для перевода в режим прослушивания. Каждый раз, когда сокет, находящийся в режиме прослушивания, принимает пакет первого этапа установления TCP-соединения, он сохраняет запрос до момента, когда процесс, владеющий сокетом, примет запрос путем вызова функции `accept` и продолжит процедуру установления соединения.

`backlog` — максимальное число входящих запросов на соединение, ожидающих обработки в очереди. Как только это число будет превышено, любые последующие запросы начнут игнорироваться. Чтобы использовать значение по умолчанию, передайте в этом параметре значение `SOMAXCONN`.

В случае успеха функция вернет `0`, в случае ошибки — `-1`.

Чтобы принять входящий запрос и продолжить процедуру установки TCP-соединения, следует вызвать функцию `accept`:

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

`sock` — сокет в режиме прослушивания, для которого требуется принять запрос на соединение.

`addr` — указатель на структуру `sockaddr`, куда будет записан адрес удаленного узла, запросившего соединение. Подобно структуре адреса, передаваемой функции `recvfrom`, структура `sockaddr` в этом параметре не требует инициализации и не управляет приемом соединения. Она просто используется для сохранения адреса принятого соединения.

`addrlen` должен быть указателем на переменную с размером буфера `addr` в байтах. Функция `accept` может изменить эту переменную, записав в нее фактический размер сохраненной структуры с адресом.

В случае успеха `accept` создаст и вернет новый сокет, который можно использовать для взаимодействий с удаленным узлом. Этот новый сокет будет связан с тем же портом, что и сокет, находящийся в режиме прослушивания. Когда ОС примет пакет для данного порта, то по адресу и порту отправителя она определит, какому сокету передать пакет: не забывайте, что протокол TCP требует иметь уникальный сокет для каждого удаленного узла, с которым установлено соединение.

Новый сокет, возвращаемый функцией `accept`, связан с удаленным узлом, инициировавшим соединение. Он хранит адрес и порт удаленного узла, следит за всеми исходящими пакетами и автоматически повторяет их передачу в случае потери. Кроме того, это единственный сокет, который может взаимодействовать с данным удаленным узлом: процесс никогда не должен пытаться посылать данные удаленному узлу через сокет, инициировавший процедуру соединения и находящийся в режиме прослушивания. Эта попытка обречена на провал, так как прослушивающий сокет не соединен ни с каким узлом. Он просто выполняет функции диспетчера, помогая создать новый сокет в ответ на входящий запрос.

По умолчанию, в отсутствие ожидающих запросов на соединение функция `accept` заблокирует вызывающий поток выполнения, пока не поступит первый же запрос или пока не истечет предельное время ожидания.

Процесс, ожидающий запросов на соединение и принимающий их, является асимметричным. Сокет в режиме прослушивания нужен только пассивному серверу. Клиент, желающий установить соединение, напротив, должен создать сокет и использовать функцию `connect`, чтобы инициировать процедуру соединения с удаленным сервером:

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

`sock` — это сокет, через который устанавливается соединение.

`addr` — указатель на структуру с адресом удаленного узла.

`addrlen` — размер структуры в параметре `addr`.

В случае успеха `connect` вернет `0`, в случае ошибки — `-1`.

Вызов `connect` запускает процедуру установки TCP-соединения отправкой начального пакета с флагом `SYN` указанному узлу. Если на удаленном узле имеется сокет в режиме прослушивания, связанный с соответствующим портом, он обработает запрос вызовом функции `accept`. По умолчанию вызов функции `connect` блокирует вызывающий поток выполнения до тех пор, пока соединение не будет принято или пока не истечет предельное время ожидания.

Отправка и прием данных через подключенные сокеты

Подключенный сокет TCP хранит адрес удаленного узла. Поэтому процессу не требуется указывать адрес в каждой попытке отправить данные. При использовании сокетов TCP для отправки данных следует вызывать не функцию `sendto`, а функцию `send`:

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

`sock` — сокет, через который отправляются данные.

`buf` — буфер с данными для записи в поток. Обратите внимание, что в отличие от UDP здесь `buf` не является датаграммой и его отправка в виде единого блока не гарантируется. Данные просто добавляются в конец исходящего буфера сокета и передаются в некоторый момент в будущем, который определяется самой библиотекой сокетов. Если включен алгоритм Нейгла, описанный в главе 2, этот

момент наступит, только когда накопится объем данных, равный максимальному размеру сегмента (MSS).

`len` — число байтов для передачи. В отличие от UDP нет причин стараться удерживать это число ниже предполагаемой величины MTU канального уровня. Пока в исходящем буфере сокета есть место, сетевая библиотека будет добавлять данные и посылать их, когда их объем достигнет подходящей величины.

`flags` — битовая коллекция флагов, объединенных поразрядной операцией ИЛИ (OR), управляющих отправкой данных. Для большинства игр в этом параметре следует передавать `0`.

В случае успеха `send` вернет объем отправленных данных. Это значение может быть меньше параметра `len`, если места в исходящем буфере сокета было недостаточно, чтобы уместить в него все содержимое `buf`. Если в исходящем буфере вообще нет места, то вызывающий поток по умолчанию будет заблокирован, пока в буфере не освободится достаточно места или пока не истечет предельное время ожидания. В случае ошибки `send` вернет `-1`. Ненулевой положительный результат вовсе не означает, что какие-либо данные были отправлены, он лишь говорит о том, что данные добавлены в очередь для передачи.

Получить данные из подключенного сокета TCP можно вызовом `recv`:

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

`sock` — сокет, откуда требуется прочитать данные.

`buf` — буфер, куда должны быть скопированы принятые данные. Скопированные данные удаляются из приемного буфера сокета.

`len` — максимальный объем данных, который можно скопировать в `buf`.

`flags` — битовая коллекция флагов, объединенных поразрядной операцией ИЛИ (OR) и управляющих приемом данных. Любые флаги, которые можно использовать в вызове `recvfrom`, также можно использовать в вызове `recv`. Для большинства игр в этом параметре следует передавать `0`. В случае успеха `recv` вернет число принятых байтов. Оно может быть меньше значения параметра `len`. Нельзя предсказать, сколько байтов будет получено, опираясь на число вызовов `send`: сетевая библиотека на удаленном узле накапливает передаваемые данные и отправляет их, как только объем данных достигнет размера сегмента. Если `recv` вернет ноль при ненулевом значении `len`, это означает, что другая сторона соединения послала пакет `FIN` и не имеет данных для передачи. Если `recv` вернет ноль при нулевом значении `len`, это означает, что в сокете есть данные, готовые для чтения. При обслуживании множества сокетов это обстоятельство можно использовать для проверки наличия данных без необходимости выделять буфер для каждого из них. Как только `recv` сообщит, что в сокете есть данные для чтения, можно выделить память для буфера и вновь вызвать `recv`, передав ей этот буфер и ненулевое значение в `len`.

В случае ошибки `recv` вернет `-1`.

По умолчанию, если в приемном буфере сокета нет данных, `recv` заблокирует вызывающий поток выполнения, пока не будет получен следующий сегмент или пока не истечет предельное время ожидания.

ПРИМЕЧАНИЕ При желании для работы с подключенным сокетом можно использовать функции `sendto` и `recvfrom`. Однако в этом случае параметр с адресом будет игнорироваться, а сам программный код будет выглядеть запутанным. Аналогично, на некоторых платформах допускается вызов функции `connect` с сокетом UDP, чтобы сохранить адрес и порт удаленного узла в области данных о соединениях в сокете. Надежное соединение при этом не устанавливается, но такой прием позволяет использовать `send` для передачи данных по сохраненному адресу без необходимости каждый раз указывать адрес. Кроме того, такой сокет не будет принимать датаграммы с любых других адресов, отличающихся от сохраненного.

Безопасность типов сокетов TCP

Класс `TCPSocket` напоминает класс `UDPSocket`, но включает дополнительные методы поддержки постоянного соединения. Реализация класса приводится в листинге 3.7.

Листинг 3.7. Класс TCPSocket

```
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` обладает методами, поддерживающими специфику протокола TCP: `Send`, `Receive`, `Connect`, `Listen` и `Accept`. Метод `Bind` и деструктор не отличаются от аналогичных методов класса `UDPSocket`, поэтому здесь они не показаны. `Accept` возвращает `TCPSocketPtr`, гарантирующий автоматическое закрытие нового сокета при уничтожении ссылки на него. `Send` и `Receive` не требуют передачи адреса, потому что автоматически используют адрес, хранящийся в подключенном сокете.

Чтобы получить возможность создавать экземпляры `TCPSocket`, необходимо добавить функцию `CreateTCPSocket` в `SocketUtils`.

Блокирующий и неблокирующий ввод/вывод

Операция приема данных из сокета обычно выполняется в блокирующем режиме. Если в сокете нет данных, готовых к извлечению, поток выполнения блокируется до их появления. Такое поведение нежелательно, когда извлечение пакетов осуществляется в главном потоке выполнения. Операции отправки данных, приема и установки соединения также блокируют работу потока, если сокет не готов выполнить операцию. Это представляет проблему для приложений, действующих в режиме реального времени, таких как игры, которые должны иметь возможность проверить наличие входящих данных без снижения частоты кадров. Представьте, что игровой сервер поддерживает TCP-соединения с пятью клиентами. Если сервер вызовет `recv` с одним из сокетов, чтобы проверить получение данных от одного из своих клиентов, поток выполнения приостановится до тех пор, пока клиент не отправит какие-нибудь данные. Это помешает серверу проверить наличие данных в других сокетах, принять новые соединения и выполнить игровые операции. Очевидно, что игра не может работать таким образом. К счастью, существует три универсальных способа обойти описанную проблему: многопоточность, неблокирующий ввод/вывод и функция `select`.

Многопоточность

Один из способов обойти проблему блокирования ввода/вывода — производить блокирующие вызовы в отдельных потоках выполнения. В примере, описанном выше, серверу понадобилось бы запустить семь потоков выполнения: по одному для каждого клиента, один для сокета, принимающего запросы на соединение, и еще один или несколько для выполнения игровых операций. Организация такого процесса представлена на рис. 3.1.

На запуске поток выполнения, принимающий запросы на соединение, создает сокет, связывает его, вызывает `listen` и затем вызывает `accept`. Вызов `accept` блокируется, пока какой-нибудь клиент не попытается установить соединение. После этого вызов `accept` вернет новый сокет. Серверный процесс запустит для этого сокета новый поток выполнения, который в цикле будет вызывать `recv`. Вызов `recv` блокируется, пока клиент не пришлет данные. Когда это произойдет, `recv` разблокируется и поток выполнения, задействовав некоторый механизм обратных вызовов, передаст данные от клиента в основной поток выполнения, после чего замкнет цикл и вновь вызовет `recv`. Параллельно с этим поток, принимающий запросы на соединение, продолжит устанавливать новые соединения, а главный поток — выполнять игровые операции.

Это вполне работоспособное решение, но оно требует создания потока выполнения для каждого клиента, из-за чего плохо масштабируется с увеличением числа клиентов. Кроме того, оно может вызывать сложности с управлением, так как все потоки обслуживания клиентов будут принимать данные параллельно и придется использовать какой-то механизм, позволяющий обезопасить выполнение игровых операций. Наконец, если главный поток попытается послать данные в сокет в то же самое время, когда другой поток будет принимать данные из этого сокета, он

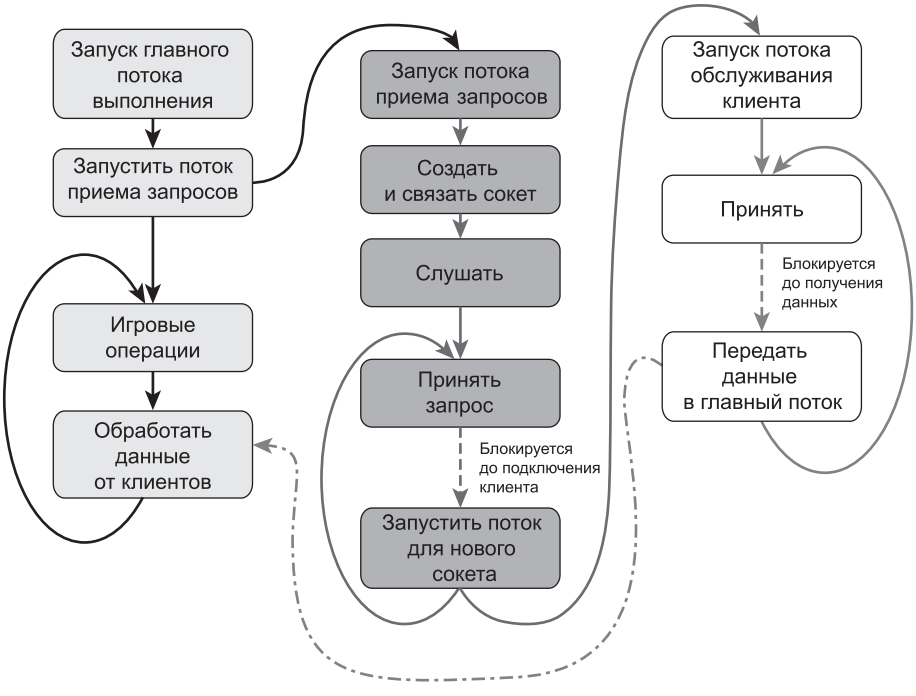


Рис. 3.1. Многопоточный процесс

окажется заблокированным и не сможет дальше эмулировать игру. Эти проблемы можно решить, но есть более простые варианты.

Неблокирующий ввод/вывод

Как упоминалось выше, по умолчанию сокеты действуют в блокирующем режиме. Но сокеты поддерживают также *неблокирующий* режим работы. Когда сокету, действующему в неблокирующем режиме, предлагается выполнить операцию, которая в ином случае заблокировалась бы, он сразу возвращает управление с результатом `-1`. При этом устанавливается код ошибки `EAGAIN` в `errno` или `WSAGetLastError` в `WSAEWOULDBLOCK`. Этот код сообщает, что предыдущая операция с сокетом могла бы заблокироваться, но вместо этого была прервана. Вызывающий процесс может теперь среагировать в соответствии с алгоритмом.

Перевести сокет в неблокирующий режим в Windows можно с помощью функции `ioctlsocket`:

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

`sock` — сокет для перевода в неблокирующий режим.

`cmd` — параметр сокета, который нужно изменить. В данном случае требуется передать значение `FIONBIO`.

argp — значение для параметра. Любое ненулевое значение включит неблокирующий режим, а нулевое — отключит.

В POSIX-совместимых операционных системах для этой цели используется функция fcntl:

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

sock — сокет для перевода в неблокирующий режим.

cmd — команда, которую нужно применить к сокету. В новейших POSIX-совместимых системах необходимо сначала выполнить команду F_GETFL, чтобы извлечь флаги, ассоциированные с сокетом в настоящий момент, объединить их при помощи операции ИЛИ (OR) с константой O_NONBLOCK и затем выполнить команду F_SETFL, чтобы изменить флаги в сокете. В листинге 3.8 демонстрируется метод для класса UDPSocket, включающий неблокирующий режим.

Листинг 3.8. Включение неблокирующего режима в сокете

```
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;
  }
}
```

Если сокет действует в неблокирующем режиме, можно без опаски вызывать любые блокирующие функции, которые немедленно будут возвращать управление при невозможности завершить операцию без блокировки. С использованием неблокирующих сокетов типичный игровой цикл мог бы выглядеть следующим образом (листинг 3.9).

Листинг 3.9. Игровой цикл с использованием неблокирующих сокетов

```
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();
  }
}
```

Переведя сокеты в неблокирующий режим, игра может выполнять проверку появления новых данных в каждом кадре. Если получены новые данные, игровой процесс сначала обрабатывает ожидающую датаграмму. Если данных нет, игра сразу переходит к выполнению операций в кадре. Если потребуется организовать обработку сразу нескольких датаграмм в одной итерации, можно добавить вложенный цикл, который будет извлекать ожидающие датаграммы, пока не прочитает их все или некоторое предельное количество. Важно ограничить число датаграмм, извлекаемых в одном кадре. Если этого не сделать, злонамеренный клиент сможет послать огромное количество однобайтных датаграмм быстрее, чем игра в состоянии будет их обработать, что, фактически, не позволит серверу продолжать выполнение игровых операций.

Функция select

Опрос неблокирующих сокетов в каждом кадре — простой и прямолинейный способ проверки появления новых входных данных без блокировки потока выполнения. Однако когда число опрашиваемых сокетов становится очень большим, такое решение перестает быть эффективным. В качестве альтернативы библиотека сокетов дает возможность проверить сразу множество сокетов и выполнить действие, как только один из них будет готов. Для этого используется функция `select`:

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

На POSIX-совместимых платформах параметр `nfds` определяет максимальное значение идентификатора сокета, подлежащего проверке. В POSIX каждый сокет представлен целым числом, поэтому данный параметр просто определяет максимальное число всех сокетов, передаваемых функции. В Windows, где сокеты представлены указателями, в этом параметре ничего передавать не нужно, и его можно просто игнорировать.

`readfds` — указатель на коллекцию сокетов, известную как `fd_set`, которая должна содержать сокеты для проверки на наличие в них данных, готовых к чтению. Как сформировать `fd_set`, описывается ниже. Когда любой сокет из множества `readfds` примет пакет, `select` вернет управление вызывающему потоку сразу, как только сможет. Но перед этим она сначала исключит из множества все сокеты, не имеющие информации для чтения. То есть когда `select` вернет управление, поток выполнения сможет выполнить чтение из любого сокета, оставшегося в `readfds`, без риска быть заблокированным. Если в `readfds` передать `nullptr`, проверка сокетов на доступность для чтения выполняться не будет.

`writefds` — указатель на коллекцию `fd_set` сокетов для проверки на готовность к записи. Когда `select` вернет управление, все сокеты, оставшиеся в коллекции `writefds`, гарантированно будут готовы к записи без блокировки вызывающего потока выполнения. Если в `writefds` передать `nullptr`, проверка сокетов на доступность для записи выполняться не будет. Обычно операция записи в сокет блокируется, только когда исходящий буфер сокета полностью заполнен.

`exceptfds` — указатель на коллекцию `fd_set` сокетов для проверки на ошибки. Когда `select` вернет управление, в наборе `exceptfds` останутся сокеты, в которых возникли ошибки. Если в `exceptfds` передать `nullptr`, проверка сокетов на наличие ошибок выполняться не будет.

`timeout` — указатель на переменную, определяющую предельное время ожидания любого из трех событий. Если время истекло до того, как любой сокет из `readfds` станет доступен для чтения, или любой сокет из `writefds` станет доступен для записи, или в любом сокете из `exceptfds` обнаружится ошибка, все три набора будут опустошены и `select` вернет управление вызвавшему потоку выполнения. Если в `timeout` передать `nullptr`, время ожидания не будет ограничено.

`select` возвращает число сокетов, оставшихся в `readfds`, `writefds` и `exceptfds`. В случае выхода по тайм-ауту она вернет `0`.

Чтобы инициализировать пустой набор `fd_set`, объявите его на стеке и обнулите макросом `FD_ZERO`:

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

Добавить сокет в набор можно макросом `FD_SET`:

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

Проверить присутствие сокета в наборе после возврата из `select` можно макросом `FD_ISSET`:

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

`select` оперирует не единственным сокетом, поэтому ее нельзя обернуть методом класса-обертки сокета. Правильнее ее обернуть методом во вспомогательном классе `SocketUtils`. В листинге 3.10 представлена функция `Select` для работы с экземплярами `TCPSocket`.

Листинг 3.10. Использование select с экземплярами 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)
{
  // сконструировать множества из векторов
  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;
}
```

Вспомогательные функции `FillSetFromVector` и `FillVectorFromSet` преобразуют вектор сокетов в `fd_set` и обратно, позволяют передавать `nullptr` вместо вектора, когда пользователь захочет передать `nullptr` вместо набора `fd_set`. Если такое решение и страдает некоторой неэффективностью, то уж точно не является проблемой в сравнении со временем, которое будет потрачено в случае блокировки операций с сокетами. Чтобы добиться лучшей производительности, можно обернуть `fd_set` типом данных на языке C++, поддерживающим оптимальный способ итераций через сокеты, оставшиеся после возврата из вызова `select`. Сохраните все соответствующие сокеты в экземпляре этого типа данных и не забудьте передать копию в вызов функции `select`, чтобы она не изменяла исходный набор.

Листинг 3.11 демонстрирует, как с помощью функции `Select` реализовать простой цикл работы TCP-сервера, устанавливающего соединения с новыми клиентами и одновременно принимающего данные от старых клиентов. Эту функцию можно было бы вызывать в главном или выделенном потоке выполнения.

Листинг 3.11. Цикл простого TCP-сервера

```
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))
    {
      // получен пакет — обойти сокеты...
      for(const TCPSocketPtr& socket : readableSockets)
      {
        if(socket == listenSocket)
        {
          // это сокет, принимающий запросы на соединение,
          // принять новое соединение
          SocketAddress newClientAddress;
```

```
          auto newSocket = listenSocket->Accept(newClientAddress);
          readBlockSockets.push_back(newSocket);
          ProcessNewClient(newSocket, newClientAddress);
        }
        else
        {
          // это обычный сокет – обработать данные...
          char segment[GOOD_SEGMENT_SIZE];
          int dataReceived =
            socket->Receive( segment, GOOD_SEGMENT_SIZE );
          if(dataReceived > 0)
          {
            ProcessDataFromClient( socket, segment,
                        dataReceived);
          }
        }
      }
    }
  }
}
```

Процедура начинается с создания сокета для приема запросов на соединение и добавления его в список сокетов, подлежащих проверке на доступность для чтения. Затем начинается цикл, который продолжается, пока приложение не потребует прервать его. Цикл использует функцию `Select`, которая блокирует работу, пока в один из сокетов в векторе `readBlockSockets` не поступит пакет. Когда это произойдет, `Select` гарантирует, что в `readableSockets` останутся только сокеты с доступными для чтения данными. Далее выполняется перебор сокетов, идентифицированных функцией `Select` как доступные для чтения. Если очередной сокет — это сокет, предназначенный для приема запросов на соединение, значит, удаленный узел вызвал `Connect`. Функция принимает соединение, добавляет новый сокет в `readBlockSockets` и извещает приложение вызовом `ProcessNewClient`. Но если очередной сокет — это обычный сокет, вызывается метод `Receive` для извлечения вновь поступивших данных, которые затем передаются приложению вызовом `ProcessDataFromClient`.

ПРИМЕЧАНИЕ Существуют другие способы обработки данных, поступающих из нескольких сокетов, но они зависят от типа платформы и не так широко распространены. Для поддержки тысяч одновременных соединений отличным выбором в Windows могут стать порты завершения ввода/вывода (I/O completion ports). Ссылки на более подробную информацию о портах завершения ввода/вывода можно найти в разделе «Для дополнительного чтения».

Дополнительные параметры сокетов

Поведением сокетов при приеме и передаче можно управлять с помощью различных параметров. Для установки значений этих параметров используется функция `setsockopt`:

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

`sock` — это настраиваемый сокет.

`level` и `optname` описывают устанавливаемый параметр. `level` — это целое число, идентифицирующее уровень, на котором определен данный параметр, а `optname` определяет сам параметр.

`optval` — указатель на значение для установки параметра.

`optlen` — объем данных. Например, если некоторый параметр принимает целое число, в `optlen` следует передать значение `4`.

`setsockopt` возвращает `0` в случае успеха и `-1` в случае ошибки.

В табл. 3.4 перечислены некоторые полезные параметры, доступные на уровне `SOL_SOCKET`.

Таблица 3.4. Параметры на уровне SOL_SOCKET

Макрос	Тип значения (Windows/ POSIX)	Описание
`SO_RCVBUF`	`int`	Определяет размер буфера, выделяемого этим сокетом для входящих пакетов. Входящие данные накапливаются в приемном буфере, пока процесс, владеющий сокетом, не вызовет `recv` или `recvfrom`, чтобы извлечь их. Не забывайте, что пропускная способность TCP ограничивается размером окна приема, который не может быть больше размера приемного буфера в сокете. То есть, управляя этим параметром, можно влиять на пропускную способность
`SO_REUSEADDR`	`Bool/int`	Указывает, должен ли сетевой уровень позволять данному сокету связывать IP-адрес и порт, уже связанные с другим сокетом. Это может пригодиться для отладки и в приложениях, перехватывающих трафик. Некоторые операционные системы требуют, чтобы вызывающий процесс обладал повышенными привилегиями
`SO_RECVTIMEO`	`DWORD/timeval`	Определяет время (в миллисекундах в Windows), по истечении которого заблокированная операция чтения должна прервать ожидание и вернуть управление
`SO_SNDBUF`	`int`	Определяет размер буфера, выделяемого этим сокетом для исходящих пакетов. Пропускная способность для исходящего трафика ограничивается сетевым уровнем. Если процесс посылает данные быстрее, чем сетевой уровень успевает их отправлять, эти данные будут накапливаться в исходящем буфере. Сокеты, использующие надежные протоколы, такие как TCP, хранят данные в исходящем буфере, пока принимающая сторона не подтвердит их получение. Когда исходящий буфер заполняется, функции `send` и `sendto` блокируются до освобождения необходимого объема в буфере

Макрос	Тип значения (Windows/ POSIX)	Описание
SO_SNDTIMEO	DWORD/timeval	Определяет время (в миллисекундах в Windows), по истечении которого заблокированная операция передачи должна прервать ожидание и вернуть управление
SO_KEEPALIVE	Bool/int	Применим только к сокетам, использующим протоколы, устанавливающие постоянное соединение, такие как TCP; этот параметр определяет необходимость автоматической передачи сокетом специальных пакетов, поддерживающих соединение открытым. Если другая сторона перестанет подтверждать получение этих пакетов, сокет перейдет в состояние ошибки, и когда в следующий раз процесс попытается послать данные через этот сокет, он будет извещен о потере соединения. Это может пригодиться не только для определения разрыва соединения, но также для поддержания соединений через брандмауэры и NAT, которые могут разрываться по истечении определенного времени

В табл. 3.5 описывается параметр TCP_NODELAY, доступный на уровне IPPROTO_TCP. Этот параметр можно установить только в сокетах TCP.

Таблица 3.5. Параметры на уровне IPPROTO_TCP

Макрос	Тип значения (Windows/ POSIX)	Описание
TCP_NODELAY	Bool/int	Указывает, должен ли игнорироваться алгоритм Нейгла для данного сокета. Истинное значение в этом параметре уменьшит задержки между запросом на передачу данных и фактической их передачей. Однако это может привести к образованию заторов в сети. Подробнее об алгоритме Нейгла рассказывается в главе 2 «Интернет»

В заключение

Сокеты Беркли являются самым распространенным механизмом передачи данных в Интернете. Несмотря на то что программный интерфейс библиотеки на разных платформах отличается, основные возможности остаются одинаковыми.

Базовым типом данных для адресов является sockaddr, и он может представлять адреса для разных протоколов сетевого уровня. Используйте его, если понадобится определить адрес отправителя или получателя.

Сокеты UDP не поддерживают постоянное соединение и не хранят информацию о своем состоянии. Создаются такие сокеты с помощью функции socket, а от-

правка данных через них осуществляется вызовом `sendto`. Чтобы организовать прием UDP-пакетов через сокет UDP, сначала вызовите функцию `bind`, чтобы зарезервировать порт в операционной системе, а затем — `recvfrom`, чтобы извлечь входящие данные.

Сокеты TCP поддерживают информацию о своем состоянии и требуют установить соединение перед передачей данных. Соединение устанавливается вызовом `connect`. Чтобы принять входящий запрос на соединение, вызывается функция `listen`. Когда в ожидающий сокет поступает запрос на соединение, следует вызвать `accept`, чтобы создать новый сокет, представляющий локальную конечную точку этого соединения. Передача данных через соединенные сокеты выполняется вызовом `send`, а прием — вызовом `recv`.

Операции с сокетами могут блокировать вызывающий поток выполнения и создавать проблемы для приложений, выполняющихся в режиме реального времени. Чтобы предотвратить их появление, вызовы, которые могут заблокироваться, следует выполнять в отдельных потоках выполнения, переводить сокеты в неблокирующий режим работы или использовать функцию `select`.

С помощью функции `setsockopt` настраивается поведение сокетов. После создания и настройки сокеты обеспечивают возможность взаимодействий между узлами в сетевой игре. В главе 4 «Сериализация объектов» мы начнем знакомиться с особенностями и оптимальными приемами использования этой возможности.

Вопросы для повторения

1. Назовите некоторые отличия POSIX-совместимых библиотек сокетов от реализации в Windows.
2. Какие два уровня TCP/IP отвечают за адресацию сокетов?
3. Объясните, как и почему сервер TCP создает уникальный сокет для соединения с каждым клиентом.
4. Объясните, как связать сокет с портом и что это означает.
5. Добавьте поддержку адресов IPv6 в `SocketAddress` и `SocketAddressFactory`.
6. Добавьте возможность создания сокетов TCP в `SocketUtils`.
7. Реализуйте чат-сервер, использующий протокол TCP для соединения с единственным узлом и передачи сообщений в обе стороны.
8. Добавьте поддержку нескольких клиентов в чат-сервер. Используйте неблокирующие сокеты на стороне клиента и `select` на стороне сервера.
9. Объясните, как изменить максимальный размер окна приема TCP.

Для дополнительного чтения

«I/O Completion Ports». Доступно по адресу: *https://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx*. Проверено 28 января 2016.

«Porting Socket Applications to WinSock». Доступно по адресу: *http://msdn.microsoft.com/en-us/library/ms740096.aspx*. Проверено 28 января 2016.

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

«WinSock2 Reference». Доступно по адресу: *http://msdn.microsoft.com/en-us/library/windows/desktop/ms740673%28v=vs.85%29.aspx*. Проверено 28 января 2016.

Институт информатики (Information Sciences Institute). (1981, сентябрь). «Transmission Control Protocol». Доступно по адресу: *http://www.ietf.org/rfc/rfc793.txt*.[2] Проверено 28 января 2016.

[1] *Уильям Ричард Стивенс, Билл Феннер, Эндрю М. Рудофф*. UNIX. Разработка сетевых приложений, СПб.: Питер, 2007. — *Примеч. пер.*

[2] Перевод на русский язык: *http://rfc.com.ru/rfc793.htm. — Примеч. пер.*

4 Сериализация объектов

Для передачи объектов между экземплярами сетевой многопользовательской игры код игры должен так преобразовать информацию об этих объектах, чтобы ее можно было послать по протоколу транспортного уровня. В этой главе обсуждается необходимость надежной системы *сериализации*. Здесь исследуются пути решения проблем, связанных с данными, ссылающимися на самих себя, со сжатием и простотой сопровождения кода при наличии жестких требований к производительности, предъявляемых режимом реального времени.

Необходимость сериализации

Под сериализацией понимается преобразование объекта из формата хранения в памяти с произвольным доступом в линейную последовательность битов. Эти биты могут сохраняться на диске или посылаться через сеть и затем преобразовываться в оригинальный формат. Предположим, что в игре «Robo Cat» игрок `RoboCat` представлен следующим классом:

```
class RoboCat: public GameObject
{
public:
 RoboCat(): mHealth(10), mMeowCount(3) {}
private:
 int32_t mHealth;
 int32_t mMeowCount;
};
```

Как упоминалось в главе 3 «Сокеты Беркли», программный интерфейс сокетов Беркли предлагает функции `send` и `sentdo` для передачи данных между узлами. Обе функции принимают указатель на данные для передачи. Соответственно, самый

простой способ передачи RoboCat между узлами без применения специализированного механизма сериализации мог бы выглядеть следующим образом:

```
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 приводит объект типа RoboCat к типу char*, чтобы передать его функции send. В качестве размера буфера указывается размер класса RoboCat, который в данном случае равен восьми. Функция приема, в свою очередь, снова приводит объект типа RoboCat к типу char*, на этот раз — чтобы сохранить принятые данные сразу в структуру. Допустим, что между узлами, использующими сокеты, уже установлено TCP-соединение, тогда процесс передачи состояния RoboCat между узлами будет выглядеть так:

1. Отправитель вызовет функцию NaivelySendRoboCat и передаст ей объект RoboCat для передачи.
2. Получатель создаст новый или найдет существующий объект RoboCat, в котором должно быть сохранено принятое состояние.
3. Получатель вызовет функцию NaivelyReceiveRoboCat и передаст ей указатель на объект RoboCat, выбранный на шаге 2.

В главе 5 «Репликация объектов» подробно рассказывается о шаге 2 и объясняется, как и когда следует искать или создавать принимающий объект RoboCat. А пока допустим, что система сама находит или создает объект RoboCat на стороне получателя.

По завершении процедуры передачи, если предположить, что оба узла действуют на идентичных аппаратных платформах, информация из исходного объекта RoboCat будет благополучно скопирована в целевой объект RoboCat. Размещение объекта RoboCat в памяти, представленное в табл. 4.1, демонстрирует, почему примитивные функции передачи/приема эффективно справляются со своей задачей.

Объект RoboCat на стороне получателя имел в поле mHealth значение 10 и в поле mMeowCount значение 3, установленные конструктором RoboCat. В ходе выполнения логики игры на стороне отправителя RoboCat потерял половину здоровья, значение которого теперь равно 5, и использовал одно «мяу». Так как поля mHealth и mMeowCount хранят значения элементарных типов, примитивная реализация передачи/приема работает правильно и объект RoboCat на стороне получателя получает правильные значения полей.

Таблица 4.1. Размещение объекта RoboCat в памяти

Адрес	Поле	Значение на стороне отправителя	Исходное значение на стороне получателя	Конечное значение на стороне получателя
Байты 0–3	mHealth	0x00000005	0x0000000A	0x00000005
Байты 4–7	mMeowCount	0x00000002	0x00000003	0x00000002

Однако объекты, являющиеся ключевыми элементами игры, редко бывают такими же простыми, как RoboCat из табл. 4.1. Более реалистичная версия RoboCat вызывает сложности, которые нарушают нормальную работу простой реализации передачи/приема и вынуждают задуматься о создании более надежной системы сериализации:

```
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;
};
```

Расширенная версия RoboCat создает сложности, которые необходимо учитывать при сериализации. В табл. 4.2 показано, как располагается этот объект в памяти до и после передачи.

Первые 4 байта в объекте RoboCat теперь занимает указатель на таблицу виртуальных методов. Здесь предполагается, что программа компилировалась в 32-разрядной архитектуре — в 64-разрядной системе этот указатель занял бы 8 байт. Теперь, когда класс RoboCat получил виртуальный метод RoboCat::Update(), каждый экземпляр RoboCat должен хранить указатель на таблицу с адресами реализаций виртуальных методов для RoboCat. Это порождает проблему для простой процедуры передачи/приема, потому что в разных процессах таблица виртуальных методов может иметь разные адреса. В данном случае при копировании полученного объекта RoboCat на стороне получателя правильный указатель 0x0B325080 на таблицу виртуальных методов заменяется ошибочным. После этого попытка вызвать метод Update принятой копии RoboCat на стороне получателя в лучшем случае вызовет ошибку доступа к памяти, а в худшем — запустит некий произвольный код.

Таблица 4.2. Размещение усложненной версии RoboCat в памяти

Адрес	Поле	Значение на стороне отправителя	Исходное значение на стороне получателя	Конечное значение на стороне получателя
Байты 0–3	`vTablePtr`	`0x0A131400`	`0x0B325080`	`0x0A131400`
Байты 4–7	`mHealth`	`0x00000005`	`0x0000000A`	`0x00000005`
Байты 8–11	`mMeowCount`	`0x00000002`	`0x00000003`	`0x00000002`
Байты 12–15	`mHomeBase`	`0x0D124008`	`0x00000000`	`0x0D124008`
Байты 16–143	`mName`	`"Fuzzy\0"`	`"\0"`	`"Fuzzy\0"`
Байты 144–167	`mMiceIndices`	`??????`	`??????`	`??????`

Указатель на таблицу виртуальных методов — не единственный указатель, который затирается в данном экземпляре. Копирование указателя `mHomeBase` из одного процесса в другой дает столь же бессмысленный результат. Указатели по своей природе ссылаются на области памяти в определенном адресном пространстве процесса. Поэтому нельзя слепо копировать поле указателя из одного процесса в другой и надеяться, что в памяти процесса получателя, на которую ссылается скопированный указатель, будут храниться соответствующие данные. Надежный механизм репликации должен либо скопировать данные, на которые ссылается указатель, и устанавливать ссылку на них в поле-указателе, либо найти имеющуюся версию данных в процессе-получателе и сохранить в указателе ее адрес. Обсуждение этих приемов мы продолжим в разделе «Ссылочные данные».

Еще одной проблемой простой реализации передачи/приема `RoboCat` является обязательное копирование всех 128 байт в поле `mName`. Несмотря на то что массив может хранить до 128 символов, иногда число действительно хранимых в нем символов намного меньше, как в примере объекта `RoboCat` с полем `mName`, хранящим строку «Fuzzy». Чтобы выполнить задачу по оптимизации сетевого трафика, хорошая система сериализации должна избегать сериализации ненужных данных. В данном случае система должна понять, что поле `mName` представляет строку, завершающуюся нулевым символом, и сериализовать символы до нулевого символа, включая и его. Это один из множества приемов сжатия данных в процессе сериализации, которые более подробно обсуждаются в разделе «Сжатие» далее в этой главе.

Заключительная проблема сериализации, из числа тех, которыми страдает новая версия `RoboCat`, возникает при копировании `std::vector<int32_t> mMiceIndices`. Внутренняя организация класса `vector` из библиотеки STL не регламентируется стандартом C++, потому неочевидно, будет ли правильно копироваться этот вектор простой реализацией передачи/приема из одного процесса в другой. Вероятно,

нет: в структуре вектора, скорее всего, имеется один или несколько указателей, ссылающихся на элементы вектора, и в векторе может иметься метод инициализации, который должен вызываться для настройки этих указателей. Практически наверняка простая реализация передачи/приема не сможет скопировать вектор без ошибок. Фактически можно смело предположить, что простая реализация потерпит неудачу при копировании любой структуры данных, используемой по принципу «черного ящика»: поскольку внутренняя организация структуры не определена явно, небезопасно осуществлять ее побитовое копирование. Правильная сериализация сложных структур данных рассматривается на протяжении этой главы.

Три проблемы, перечисленные выше, наглядно показывают, что экземпляр `RoboCat` не должен передаваться в сокет монолитным блоком — каждое поле должно сериализоваться отдельно, чтобы гарантировать правильность и эффективность работы программы. Для каждого поля можно создавать свой пакет и отправлять эти пакеты разными вызовами функции `send`, но это вызовет хаос в сетевом соединении и приведет к напрасной трате полосы пропускания на передачу ненужных заголовков пакетов. Вместо этого лучше собрать все необходимые данные в буфер и послать этот буфер как представление объекта. Чтобы упростить процесс, введем понятие *потока данных* (stream).

Потоки данных

В информатике под *потоком данных* (stream) понимается структура данных, представляющая упорядоченный набор элементов данных и позволяющая читать данные из этого набора или записывать в него.

Поток может быть *потоком вывода* (output stream), *потоком ввода* (input stream) или и тем и другим. Поток вывода действует как «слив» для пользовательских данных, дающий пользователю возможность последовательно добавлять в него элементы, но не читать их из него. Поток ввода, напротив, действует как источник данных, позволяющий последовательно извлекать элементы, но не предоставляет средств для записи их в поток. Когда поток одновременно является потоком ввода и вывода, он содержит методы как для добавления элементов данных, так и для их извлечения, возможно одновременного.

Часто поток данных является интерфейсом к некоторой другой структуре данных. Например, *поток вывода в файл* (file output stream) может быть оберткой вокруг файла, открытого для записи, поддерживая простой метод последовательной записи на диск данных разных типов. *Сетевой поток* (network stream) может быть оберткой для сокета, предоставляя методы-обертки вокруг функций `send()` и `recv()`, специализированные для работы с определенными типами данных.

Потоки данных в памяти

Поток данных в памяти (memory stream) обертывает буфер в памяти. Обычно такой буфер динамически размещается в *куче*. *Поток вывода в память* (output memory stream) имеет методы для последовательной записи данных в буфер,

а также метод доступа, обеспечивающий возможность чтения самого буфера. Вызвав метод доступа к буферу, пользователь может получить сразу все данные, записанные в поток, и передать их другой системе, такой как функция `send` сокета. В листинге 4.1 представлена реализация потока вывода в память.

Листинг 4.1. Поток вывода в память

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

 // возвращает указатель на данные в потоке
 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));
 // обработать ошибку вызова realloc
 // ...
 mCapacity = inNewLength;
}

void OutputMemoryStream::Write(const void* inData,
        size_t inByteCount)
{
 // проверить наличие места...
 uint32_t resultHead = mHead + static_cast<uint32_t>(inByteCount);
 if(resultHead > mCapacity)
 {
  ReallocBuffer(std::max( mCapacity * 2, resultHead));
 }

 // скопировать в начало свободной области в буфере
 std::memcpy(mBuffer + mHead, inData, inByteCount);

 // подготовить указатель на начало свободной области
 // для следующей операции записи
 mHead = resultHead;
}
```

Метод `Write(const void* inData, size_t inByteCount)` является основным способом записи данных в поток. Перегруженные версии метода `Write` принимают данные определенных типов, поэтому они не имеют дополнительного параметра для передачи числа байтов. Можно было бы сделать метод `Write` шаблонным, чтобы обеспечить поддержку любых типов данных, но для этого пришлось бы предусмотреть способ, препятствующий передаче данных сложных типов: не забывайте, что сложные типы требуют применения специальных подходов к сериализации. Использование статического утверждения с проверкой типа — один из способов обезопасить шаблонный метод `Write`:

```
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));
}
```

Использование вспомогательной функции автоматического определения количества байтов в специализированных методах помогает предотвратить ошибки из-за неверного числа байтов, переданного пользователем для выбранного им типа данных.

Всякий раз, когда в буфере `mBuffer` оказывается недостаточно места для записи новых данных, размер буфера автоматически увеличивается либо вдвое, либо до размера, достаточного для сохранения новых данных, в зависимости от того, что больше. Это типичный подход к увеличению размера буфера, а множитель можно скорректировать, приведя в соответствие с конкретными потребностями.

ВНИМАНИЕ Несмотря на то что `GetBufferPtr` возвращает указатель на внутренний буфер потока, доступный только для чтения, поток сохраняет монопольное владение буфером за собой. То есть указатель станет недействительным, как только поток освободит память, занимаемую буфером. Если необходимо, чтобы указатель, возвращаемый методом `GetBufferPtr`, оставался действительным после уничтожения объекта потока, буфер можно объявить как `std::shared_ptr<std::vector<uint8_t> >`, но оставим эту реализацию в качестве упражнения на конец главы.

Используя поток вывода в память, можно реализовать более надежные функции передачи `RoboCat`:

```
void RoboCat::Write(OutputMemoryStream& inStream) const
{
 inStream.Write(mHealth);
 inStream.Write(mMeowCount);
 // решение для mHomeBase пока отсутствует
 inStream.Write(mName, 128);
 // решение для mMiceIndices пока отсутствует
}

void SendRoboCat(int inSocket, const RoboCat* inRoboCat)
{
```

```
 OutputMemoryStream stream;
 inRoboCat->Write(stream);
 send(inSocket, stream.GetBufferPtr(),
   stream.GetLength(), 0);
}
```

Добавление метода `Write` в класс `RoboCat` открывает доступ к внутренним приватным полям и отделяет задачу сериализации от задачи отправки данных по сети. Это также дает вызывающей программе возможность записать экземпляр `RoboCat` в поток как один из набора элементов. Этот прием пригодится при копировании набора объектов, о чем рассказывается в главе 5.

Для приема экземпляра `RoboCat` на стороне получателя требуется реализовать соответствующий поток ввода из памяти и метод `RoboCat::Read`, как показано в листинге 4.2.

Листинг 4.2. Поток ввода из памяти

```
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);
 // решение для mHomeBase пока отсутствует
 inStream.Read(mName, 128);
 // решение для mMiceIndices пока отсутствует
}

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);
 }
}
```

После того как `ReceiveRoboCat` создаст временный буфер и заполнит его данными из сокета вызовом `recv`, этот буфер будет передан в монопольное владение потоку ввода из памяти. После этого пользователь сможет извлекать элементы данных в порядке их записи. Именно это делает метод `RoboCat::Read`, устанавливая соответствующие поля экземпляра `RoboCat`.

СОВЕТ При использовании этой парадигмы в готовой игре зачастую нежелательно выделять память для потока снова и снова, с получением каждого пакета, так как процедура выделения памяти может потребовать дополнительных затрат времени. Вместо этого лучше заранее выделить максимальный объем памяти для потока. В этом случае вновь поступивший пакет будет принят прямо в буфер потока, затем прочитан из потока, обработан, и в поле `mHead` запишется значение 0, чтобы подготовить поток к приему следующего пакета.

Также полезно дополнить `MemoryInputStream` возможностью управления собственной памятью. Конструктор, принимающий только максимальный размер, мог бы выделять память для буфера `mBuffer` потока, а метод доступа к буферу — возвращать `mBuffer` для передачи непосредственно в `recv`.

Данная реализация потока решает первую из проблем сериализации: она позволяет создать буфер, заполнить его значениями из отдельных полей исходного объекта, послать буфер удаленному узлу, извлечь значения полей в порядке их записи и записать их в соответствующие поля объекта-приемника. Кроме того, данный процесс не оказывает никакого влияния на поля в объекте-приемнике, которые не должны изменяться, такие как указатель на таблицу виртуальных методов.

Совместимость с порядком следования байтов

Не все процессоры хранят байты в многобайтных числах в одном и том же порядке. Порядок, в котором байты хранятся данной аппаратной платформой, так и называется: *порядок байтов* (endianness). Различают платформы с *прямым* порядком байтов (big-endian) и *обратным* (little-endian). Платформы с обратным порядком хранят младшие байты в многобайтных числах в младших адресах. Например, целое значение `0x12345678` по адресу `0x01000000` хранится в памяти на такой платформе, как показано на рис. 4.1.

Значение	0x78	0x56	0x34	0x12
Адрес	0x01000000	0x01000001	0x01000002	0x01000003

Рис. 4.1. Число 0x12345678 на платформе с обратным порядком байтов

Первым в памяти хранится самый младший байт, `0x78`. При чтении слева направо число оказывается перевернутым, именно поэтому такой порядок называют обратным. К платформам, использующим такой порядок, относятся: Intel x86, x64 и архитектура Apple iOS.

В платформах с прямым порядком, напротив, первым в памяти хранится самый старший байт. На рис. 4.2 показано, как хранится то же число по тому же адресу на платформе с прямым порядком байтов.

Значение	0x12	0x34	0x56	0x78
Адрес	0x01000000	0x01000001	0x01000002	0x01000003

Рис. 4.2. Число 0x12345678 на платформе с прямым порядком байтов

К платформам, использующим прямой порядок, относятся: Xbox 360, PlayStation 3 и архитектура IBM PowerPC.

СОВЕТ При разработке однопользовательских игр или игр для одной платформы обычно не приходится беспокоиться о порядке следования байтов, но когда возникает необходимость организовать передачу данных между платформами с разными порядками байтов, это превращается в проблему, на которую нельзя закрывать глаза. Лучшая стратегия решения этой проблемы при передаче данных с использованием потоков — выбрать определенный порядок байтов для самого потока, а затем менять порядок байтов в многобайтных числах на противоположный при записи в поток, если порядок байтов потока не совпадает с порядком байтов платформы, и аналогично менять порядок байтов на противоположный при чтении из потока, если порядок байтов платформы не совпадает с порядком байтов потока.

Большинство систем предоставляет эффективные алгоритмы изменения порядка байтов, а некоторые имеют даже встроенные функции или ассемблерные инструкции. Но если вам понадобится создать собственную реализацию, то в листинге 4.3 приводятся удобные функции изменения порядка байтов.

Листинг 4.3. Функции изменения порядка байтов

```
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);
}
```

Эти функции обрабатывают основные целочисленные значения без знака заданного размера, но они не поддерживают другие типы данных, такие как вещественные числа, вещественные числа двойной точности, целые со знаком, большие перечисления и другие, которые также требуют изменения порядка байтов. Чтобы реализовать такую поддержку, воспользуемся трюком с совмещением типов:

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

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

Этот класс имеет метод, принимающий данные одного типа, такого как `float`, и интерпретирующий их как данные другого типа, который поддерживается уже реализованными функциями изменения порядка байтов. Вспомогательные шаблонные функции, представленные в листинге 4.4, позволят затем изменить порядок байтов в значениях элементарных типов с использованием соответствующих функций.

Листинг 4.4. Шаблонные функции изменения порядка байтов

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

// специализация для 2-байтных значений...
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();
 }
};

// специализация для 4-байтных значений...
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();
 }
};

// специализация для 8-байтных значений...
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);
}
```

При вызове шаблонной функции `ByteSwap` она создаст экземпляр `ByteSwapper` с шаблоном, специализированным размером своего аргумента. Затем этот экземпляр вызовет соответствующую функцию `ByteSwap` при помощи `TypeAliaser`. В идеале компилятор оптимизирует промежуточные вызовы, оставив несколько операций, меняющих порядок байтов в регистрах.

ПРИМЕЧАНИЕ Не все данные требуют изменения порядка байтов из-за несовпадения порядка байтов платформы с порядком байтов в потоке. Например, строка однобайтных символов не требует изменения порядка байтов, потому что каждый отдельный символ представлен единственным байтом. Только данные простых типов должны подвергаться процедуре изменения порядка байтов, причем в соответствии с их размерами.

Теперь, используя `ByteSwapper`, обобщенные функции `Write` и `Read` смогут корректно поддерживать потоки с порядком байтов, отличающимся от порядка байтов платформы:

```
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));
 }
}
```

Потоки битов

Потоки в памяти, описанные в предыдущем разделе, имеют одно ограничение: они позволяют читать и записывать данные, состоящие только из целого числа байтов. В сетевом коде нередко бывает желательно представлять значения несколькими битами, что может потребовать интерпретации данных с точностью до одного бита. Для этого воспользуемся *потоком битов в памяти* (memory bit stream), способным сериализовать данные, состоящие из произвольного числа битов. Объявление такого *битового потока вывода в память* приводится в листинге 4.5.

Листинг 4.5. Объявление битового потока вывода в память

```
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;
};
```

Интерфейс потока битов напоминает интерфейс потока байтов, за исключением того, что вместо числа байтов предусматривает передачу числа битов, подлежащих записи. Операции создания, уничтожения и расширения буфера также схожи с предыдущими реализациями. Новая функциональность обеспечивается двумя методами `WriteBits`, представленными в листинге 4.6.

Листинг 4.6. Реализация битового потока вывода в память

```
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));
 }

 // вычислить смещение байта byteOffset в буфере,
 // разделив номер первого бита на 8,
 // и смещение бита bitOffset – в 3 младших битах номера
 uint32_t byteOffset = mBitHead >> 3;
 uint32_t bitOffset = mBitHead & 0x7;

 // вычислить маску для сохранения нужных битов в текущем байте
 uint8_t currentMask = ~(0xff << bitOffset);
 mBuffer[byteOffset] = (mBuffer[byteOffset] & currentMask)
       |(inData << bitOffset);

 // вычислить число неиспользованных битов
 // в целевом байте в буфере
 uint32_t bitsFreeThisByte = 8 – bitOffset;

 // если необходимо, перенести в следующий байт
 if(bitsFreeThisByte < inBitCount)
 {
  // нужно перенести в следующий байт
  mBuffer[byteOffset + 1] = inData >> bitsFreeThisByte;
 }

 mBitHead = nextBitHead;
}

void OutputMemoryBitStream::WriteBits(const void* inData, size_t inBitCount)
{
 const char* srcByte = static_cast<const char*>(inData);

 // записать все байты
 while(inBitCount > 8)
 {
  WriteBits(*srcByte, 8);
  ++srcByte;
  inBitCount -= 8;
 }

 // записать все, что осталось
```

```
 if(inBitCount > 0)
 {
  WriteBits(*srcByte, inBitCount);
 }
}
```

Внутренняя задача — запись битов в поток — решается методом `WriteBits(uint8_t inData, size_t inBitCount)`, который получает единственный байт и записывает заданное число битов из этого байта в поток. Чтобы было понятнее, как он работает, рассмотрим работу следующего фрагмента кода:

```
OutputMemoryBitStream mbs;

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

Он должен записать число 13, использовав 5 бит, и затем число 52, использовав следующие 6 бит. На рис. 4.3 показано, как выглядят эти числа в двоичном представлении.

Значение	1	0	0	0	1	1	0	1	0	0	0	0	0	1	1	0
Бит	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Байт	0								1							

Рис. 4.3. Числа 13 и 52 в двоичном представлении

То есть после выполнения этого фрагмента в памяти, на которую указывает `mbs.mBuffer`, должно быть сохранено два значения, как показано на рис. 4.4.

13:

Значение	0	0	0	0	1	1	0	1
Бит	7	6	5	4	3	2	1	0

52:

Значение	0	0	1	1	0	1	0	0
Бит	7	6	5	4	3	2	1	0

Рис. 4.4. Буфер потока с 5-битным числом 13 и 6-битным числом 52

Обратите внимание, что 5 бит числа 13 занимают первые 5 бит в байте с порядковым номером 0, а 6 бит числа 52 занимают последние 3 бита в байте 0 и первые 3 бита в байте 1.

Пройдясь по реализации метода, можно понять, как достигается такой результат. Представьте, что поток был только что создан, то есть `mBitCapacity` имеет значение `256`, `mBitHead` имеет значение `0`, и в буфере достаточно места, чтобы избежать его перераспределения в памяти. Сначала значение в поле `mBitHead`, представляющее индекс первого бита в потоке, доступного для записи, раскладывается на индекс

байта и индекс бита в этом байте. Так как байт состоит из 8 бит, индекс байта можно найти делением на 8, или, что то же самое, сдвигом вправо на 3 бита. Аналогично, индекс бита в этом байте можно получить, взяв младшие три бита, которые были сдвинуты за границы числа на предыдущем шаге. Так как число `0x7` в двоичном формате имеет вид `111`, поразрядная операция И (AND) значения `mBitHead` с числом `0x7` вернет как раз те самые 3 бита. В первом вызове, записывающем число `13`, поле `mBitHead` имеет значение `0`, соответственно, обе переменные, `byteOffset` и `bitOffset`, получают значение `0`.

Вычислив `byteOffset` и `bitOffset`, метод использует `byteOffset` как индекс в буфере `mBuffer`, чтобы найти целевой байт. Затем он сдвигает данные влево на значение `bitOffset` и выполняет поразрядную операцию ИЛИ (OR) с целевым байтом. В случае с записью числа `13` все выглядит довольно просто, потому что оба смещения равны 0. А теперь взгляните, как выглядит поток в начале вызова `WriteBits(52, 6)` (рис. 4.5).

Значение	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0
Бит	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Байт	0								1							

Рис. 4.5. Буфер потока непосредственно перед вторым вызовом WriteBits

В этот момент `mBitHead` имеет значение `5`. То есть `byteOffset` получит значение `0` и `bitOffset` — значение `5`.

Операция сдвига числа `52` влево на 5 бит вернет результат, изображенный на рис. 4.6.

Значение	1	0	0	0	0	0	0	0
Бит	7	6	5	4	3	2	1	0

Рис. 4.6. Число 52 в двоичном представлении после сдвига влево на 5 бит

Обратите внимание, что старшие биты вышли за границы байта, а младшие биты превратились в старшие. На рис. 4.7 изображен результат поразрядной операции ИЛИ (OR) этих битов с байтом 0 в буфере.

Значение	1	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0
Бит	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Байт	0								1							

Рис. 4.7. Буфер потока после сдвига числа 52 влево на 5 бит и выполнения операции ИЛИ (OR) с первым байтом в буфере

Байт 0 заполнен, но из-за переполнения, случившегося при сдвиге влево, в поток было записано только три бита из шести. Следующие строки в `WriteBits` обнаруживают и исправляют эту ситуацию. Метод вычисляет, сколько битов было свободно

в целевом байте, вычитая `bitOffset` из `8`. В данном случае результат равен 3 — это число битов, которые удалось уместить в байт. Если число свободных битов оказалось меньше числа битов, подлежащих записи, выполняется ветка обработки переполнения.

В этой ветке целевым выбирается следующий байт. Чтобы выделить биты для записи в следующий байт, метод сдвигает `inData` вправо на число свободных битов, имевшихся в предыдущем байте. На рис. 4.8 изображен результат сдвига числа `52` вправо на 3 бита.

Значение	0	0	0	0	0	1	1	0
Бит	7	6	5	4	3	2	1	0

Рис. 4.8. Число 52 после сдвига вправо на 3 бита

Старшие биты, вытесненные при сдвиге влево, теперь сдвигаются вправо и превращаются в младшие биты старшего байта. Когда метод выполнит поразрядное ИЛИ (OR) результата сдвига вправо с байтом `mBuffer[byteOffset + 1]`, поток окажется в финальном состоянии (рис. 4.9).

Значение	1	0	0	0	1	1	0	1	0	0	0	0	0	1	1	0
Бит	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Байт	0								1							

Рис. 4.9. Буфер потока в финальном состоянии

Вся основная работа выполняется методом `WriteBits(uint8_t inData, uint32_t inBitCount)`, а на долю `WriteBits(const void* inData, uint32_t inBitCount)` остается лишь разбить данные на байты и вызвать предыдущий метод `WriteBits` один раз для каждого байта.

Данный битовый поток вывода в память реализует все, что необходимо, но он далек от идеала, поскольку требует указывать число битов для каждого фрагмента, записываемого в поток. Однако часто верхняя граница числа битов зависит от типа записываемых данных. Лишь иногда требуется записать число битов, меньшее значения верхней границы. Поэтому для улучшения читаемости кода и упрощения его сопровождения можно добавить еще несколько методов, обрабатывающих основные типы данных:

```
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);}
```

Благодаря этим методам появляется возможность записывать значения простых типов, просто передавая их методу `Write`, а о передаче соответствующего числа битов позаботится параметр по умолчанию. На тот случай, если вызывающей программе потребуется указать меньшее число битов, методы готовы принять значение, переопределяющее значение параметра по умолчанию. Шаблонная функция с проверкой типа позволяет добиться большей обобщенности, чем набор перегруженных методов:

```
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);
}
```

Но даже при использовании шаблонного метода `Write` полезно реализовать специализированный метод для поддержки типа `bool`, потому что число битов для него должно быть равно 1, а не результату выражения `sizeof(bool) * 8`, равному 8.

ВНИМАНИЕ Данная реализация метода `Write` может использоваться только на платформах с обратным порядком следования байтов из-за особенностей адресации отдельных байтов. Если понадобится метод для работы на платформе с прямым порядком байтов, в программу следует добавить переупорядочение вызовом шаблонного метода `Write` перед вызовом `WriteBits` или реализовать адресацию, совместимую с прямым порядком следования байтов.

Битовый поток ввода из памяти, читающий биты из буфера, действует аналогично битовому потоку вывода в память. Его реализацию можно найти на сайте книги, но прежде чем заглянуть в нее, попробуйте реализовать этот поток самостоятельно.

Ссылочные данные

Код сериализации теперь способен обрабатывать данные элементарных типов и простые структуры, но пасует перед косвенными ссылками на данные, доступные через указатели или другие контейнеры. Вспомните определение класса `RoboCat` (приводится ниже):

```
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;
};
```

Здесь есть две сложные переменные, которые не поддерживаются текущей реализацией потока в памяти, — `mHomeBase` и `mMiceIndices`. Каждая из них требует своей стратегии сериализации, они обсуждаются в следующих разделах.

Встраивание или внедрение

Иногда требуется выполнить сериализацию переменных-членов, ссылающихся на данные, не используемые никакими другими объектами. Отличным примером может служить `mMiceIndices` в `RoboCat`. Это вектор целых чисел, хранящий индексы различных мышек, которыми заинтересовался наш `RoboCat`. Так как `std::vector<int>` — это «черный ящик», нельзя просто скопировать данные по адресу `std::vector<int>` в поток, используя стандартную функцию `OutputMemoryStream::Write`, потому что дело закончится сериализацией указателей в векторе `std::vector`, которые после десериализации на удаленном узле будут указывать на «мусор».

Вместо сериализации самого вектора функция должна записать в поток только данные, хранящиеся в векторе. Эти данные могут находиться в ОЗУ очень далеко от самого объекта `RoboCat`. Однако функция сериализации объекта должна записать эти данные в поток, внедрив их непосредственно в данные `RoboCat`. По этой причине данный процесс называют *встраиванием* (inlining) или *внедрением* (embedding). Например, функция сериализации `std::vector<int32_t>` могла бы выглядеть следующим образом:

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

Сначала функция сериализует размер вектора, а затем данные из него. Обратите внимание, что метод `Write` должен сериализовать размер вектора первым, чтобы соответствующий метод `Read` смог использовать его для создания вектора соответствующего размера перед десериализацией содержимого. Так как вектор хранит простые целые числа, метод сериализует их все сразу единственным вызовом `memcpy`. Для поддержки более сложных типов данных шаблонная версия метода `std::vector` `Write` сериализует элементы по одному:

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

Здесь после сериализации размера вектора метод по отдельности внедряет все элементы вектора. Такое решение обеспечивает поддержку векторов из векторов или векторов классов, содержащих векторы, и т. д. Для десериализации требуется реализовать аналогичную функцию `Read`:

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

Для поддержки контейнеров других типов и любых данных, доступных по указателям, можно реализовать дополнительные специализированные функции `Read` и `Write`, если эти данные находятся в монопольном владении сериализуемого объекта. Если данные совместно используются другими объектами или указывают на другие объекты, необходимо более сложное решение, известное как *связывание* (linking).

Связывание

Иногда на сериализуемые данные должны ссылаться сразу несколько указателей. Для примера рассмотрим поле `GameObject*` `mHomeBase` в `RoboCat`. Если два экземпляра `RoboCat` живут в одном доме, нет никакой возможности отразить этот факт текущим набором инструментов. Прием внедрения просто внедрит копию одного

и того же дома в каждый экземпляр RoboCat в процессе сериализации. В результате при десериализации будет создано два разных дома!

В некоторых случаях данные бывают структурированы таким способом, что внедрение просто невозможно. Взгляните на определение класса HomeBase:

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

Экземпляр HomeBase содержит список всех активных экземпляров RoboCat. А теперь представьте функцию сериализации объекта RoboCat, которая использует только прием внедрения. При сериализации объекта RoboCat функции потребуется внедрить его ссылку на HomeBase и затем, по цепочке, внедрить все активные экземпляры RoboCat, включая объект RoboCat, сериализация которого выполняется в данный момент. Это надежный рецепт организации переполнения стека из-за бесконечной рекурсии. Очевидно, что необходим какой-то другой инструмент.

Решение заключается в том, чтобы присвоить уникальный идентификатор каждому объекту, доступному по ссылке, и затем сериализовать ссылки на эти объекты как простые идентификаторы. После десериализации всех объектов на другом конце сети специальная подпрограмма могла бы отыскивать объекты по идентификаторам и сохранять указатели на них в соответствующие переменные-члены. Из-за этой особенности данный процесс называют *связыванием*.

Как присвоить уникальные идентификаторы всем объектам, посылаемым по сети, и как реализовать отображение этих идентификаторов в объекты, мы узнаем в главе 5, а пока предположим, что каждый поток имеет доступ к объекту LinkingContext (листинг 4.7), содержащему карту соответствий между сетевыми идентификаторами и игровыми объектами.

Листинг 4.7. Класс LinkingContext

```
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;
};
```

Класс `LinkingContext` реализует простую систему связывания для потоков в памяти:

```
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);
}
```

ПРИМЕЧАНИЕ Полностью реализованная система связывания и игровой код, использующий ее, должны быть готовы принять сетевые идентификаторы, которым в карте не соответствует ни один объект. Так как пакеты имеют свойство теряться, игра может получить объект с переменной-членом, ссылающейся на объект, который еще не был получен. Существует множество разных способов решения этой проблемы — игра может полностью игнорировать такие объекты или десериализовать их, связывая доступные ссылки, а недостающие инициализировать пустыми указателями. Более сложная система могла бы следить за переменными-членами с пустыми ссылками, чтобы при получении объекта с соответствующим сетевым идентификатором связать его. Конкретный выбор во многом зависит от специфики игры.

Сжатие

Имея инструменты сериализации всех типов данных, можно приступать к созданию кода, посылающего игровые объекты по сети. Однако такой код не всегда будет эффективно работать в условиях ограниченной пропускной способности самой сети. Раньше многопользовательские игры были вынуждены довольствоваться соединениями со скоростью передачи 2400 бит/с или даже меньше. В наше время создатели игр пользуются соединениями, скорость передачи в которых выше на

порядок, тем не менее они все еще должны заботиться о максимально эффективном использовании полосы пропускания.

Большой игровой мир может насчитывать сотни движущихся объектов, и передача полной информации о них в режиме реального времени сотням игроков легко может забить соединения даже с высочайшей пропускной способностью. В этой книге исследуется множество способов извлечь максимальную пользу из имеющейся полосы пропускания. В главе 9 «Масштабируемость» рассматриваются высокоуровневые алгоритмы, определяющие, кто и какие данные должен получать и какие свойства объектов нужно обновлять у того или иного клиента. В этом разделе, однако, мы начнем с самых основ и исследуем распространенные приемы сжатия данных на уровне битов и байтов, чтобы после выбора данных для передачи игра могла отправить их, используя как можно меньшее число битов.

Сжатие разреженного массива

Суть сжатия состоит в том, чтобы удалить всю информацию, которая не должна передаваться по сети. Такую ненужную информацию часто легко найти в разреженных или не полностью заполненных структурах данных. Рассмотрим поле `mName` в `RoboCat`. По какой-то причине создатель класса `RoboCat` решил, что имя экземпляра `RoboCat` лучше хранить в 128-байтном символьном массиве, в середине структуры данных. Метод потока `WriteBytes(const void* inData, uint32_t inByteCount)` уже способен внедрять символьные массивы, но если задуматься, он вполне мог бы сериализовать данные, не записывая в поток все 128 байт.

Основная стратегия сжатия сводится к анализу типичного случая и реализации алгоритмов, помогающих извлечь из него дополнительные выгоды. Именно этот подход мы и предпримем. Учитывая типичные имена, выбираемые пользователями, и архитектуру игры «Robo Cat», весьма велика вероятность, что пользователю не понадобятся все 128 символов для именования экземпляров `RoboCat`. То же можно сказать о любом массиве, независимо от его размера: только потому, что место в памяти выделяется для самого худшего случая, код сериализации не должен предполагать, что каждый случай будет худшим. В силу этого процедура сериализации может сэкономить память, просмотрев содержимое поля `mName` и подсчитав число символов, фактически используемых в имени. Если строка в `mName` завершается нулевым символом, задача становится тривиальной благодаря наличию функции `std::strlen`. Например, ниже показан оптимальный способ сериализации имени:

```
void RoboCat::Write(OutputMemoryStream& inStream) const
{
 ...// до этого момента выполняется сериализация других полей

 uint8_t nameLength =
  static_cast<uint8_t>(strlen(mName));
 inStream.Write(nameLength);
 inStream.Write(mName, nameLength);
 ...
}
```

Обратите внимание, что при сериализации вектора метод сначала записывает объем сериализованных данных и только потом сами данные. Это делается для того, чтобы принимающая сторона знала, какой объем данных читать из потока. Длина строки сериализуется этим методом в единственный байт. Это вполне безопасно, потому что массив может хранить не более 128 символов.

В действительности, если исходить из предположения, что доступ к имени осуществляется не так часто, как к остальной информации в экземпляре `RoboCat`, гораздо эффективнее с точки зрения кэширования представить имя объекта в виде `std::string`, что позволит уместить весь тип `RobotCat` в меньшее число строк кэша. В данном случае для сериализации имени можно было бы использовать метод, напоминающий метод сериализации векторов, реализованный в предыдущем разделе. Это делает пример с конкретным полем `mName` несколько искусственным, но суть от этого не меняется, и разреженные контейнеры все еще остаются отличной тактической целью сжатия.

Энтропийное кодирование

Энтропийное кодирование (Entropy encoding) — это понятие из теории информации, относящееся к проблеме сжатия данных, основанной на непредсказуемости появления отдельных элементов данных. Согласно теории, в пакете с ожидаемыми значениями меньше информации, чем в пакете с неожидаемыми. Поэтому для передачи ожидаемых значений должно требоваться меньше битов, чем для неожидаемых.

Часто важнее потратить такты процессора CPU на выполнение игровых действий, чем на вычисление точной величины энтропии пакета для достижения оптимального уровня сжатия. Однако существует очень простая и эффективная форма энтропийного кодирования. Она может пригодиться для сериализации переменных-членов, получающих некоторые определенные значения чаще других.

Например, рассмотрим поле `mPosition` класса `RoboCat`. Это значение типа `Vector3` с компонентами *X*, *Y* и *Z*. *X* и *Z* представляют координаты кота на поверхности земли, а *Y* — высоту кота над уровнем земли. Простейшая форма сериализации позиции могла бы выглядеть как-то так:

```
void OutputMemoryBitStream::Write(const Vector3& inVector)
{
 Write(inVector.mX);
 Write(inVector.mY);
 Write(inVector.mZ);
}
```

В этом случае для передачи `mPosition` по сети потребуется отправить $3 \times 4 = 12$ байт. Однако эта реализация не учитывает, что коты часто находятся точно на поверхности земли. То есть в большинстве векторов `mPosition` координата *Y* будет равна 0. Метод мог бы использовать всего один бит, чтобы показать, имеет ли компонент *Y* в `mPosition` типичное значение 0 или какое-то другое, менее типичное значение:

```
void OutputMemoryBitStream::WritePos(const Vector3& inVector)
{
 Write(inVector.mX);
 Write(inVector.mZ);
 if(inVector.mY == 0)
 {
  Write(true);
 }
 else
 {
  Write(false);
  Write(inVector.mY);
 }
}
```

После записи компонентов *X* и *Y* метод проверяет, равна ли нулю высота над поверхностью земли. Если она равна 0, записывается один бит с истинным значением, сообщающим: «да, объект находится на типичной высоте 0». Если компонент *Y* не равен 0, метод записывает один бит с ложным значением, сообщающим: «высота не равна 0, поэтому в следующих 32 битах находится фактическая высота». Обратите внимание, что в худшем случае для представления высоты потребуется 33 бита — один для флага, помогающего отличить типичное значение от нетипичного, и 32 бита для нетипичного значения. На первый взгляд такое решение может показаться неэффективным, так как сериализованное представление теперь занимает больше битов, чем прежде. Но для подсчета истинного числа битов, используемых в среднем, требуется точно определить, насколько типичным является случай, когда кот находится на поверхности земли.

Внедрение в игру механизма телеметрии поможет точно узнать, как часто кот находится на земле, — либо с привлечением тестировщиков, опробующих игру на сайте, либо от обычных пользователей, опробующих предварительную версию игры и присылающих аналитическую информацию через Интернет. Допустим, в ходе такого эксперимента выявилось, что в 90 % случаев кот находится на земле. Тогда ожидаемое число битов для представления высоты можно получить с помощью простого уравнения:

$$P_{НаЗемле} \times Bits_{НаЗемле} + P_{НаВысоте} \times Bits_{НаВысоте} = 0{,}9 \times 1 + 0{,}1 \times 33 = 4{,}2.$$

Ожидаемое число битов, необходимых для сериализации компонента *Y*, уменьшается с 32 до 4,2: экономия составляет более 3 байт на позицию. При наличии 32 игроков, обменивающихся позициями 30 раз в секунду, это решение может дать существенную экономию только на одной этой переменной.

Сжатие может быть еще более эффективным. Допустим, аналитические данные показывают, что когда кот не находится на уровне пола, он часто оказывается на уровне потолка, на высоте 100. В таком случае код мог бы обеспечить поддержку второго типичного значения — высоты на уровне потолка:

```
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);
 }
}
```

Метод все так же использует один бит, чтобы показать, что высота имеет типичное или нетипичное значение, но добавляет второй бит, сообщающий, какое из двух типичных значений использовать. Здесь типичные значения «зашиты» в код функции, но при наличии большего числа типичных значений подобная реализация будет выглядеть запутанной. В таких ситуациях можно использовать упрощенную реализацию алгоритма кодирования Хаффмана, выполняющего поиск по таблице типичных значений и замещающего типичные значения несколькими битами с индексами значений в таблице.

Однако вновь встает вопрос определения качества такой оптимизации: сам факт, что высота потолка является вторым типичным значением координаты *Y* кота, еще не означает, что данная оптимизация окажется эффективной, поэтому необходимо вновь обратиться к математическим расчетам. Допустим, аналитические данные показывают, что кот находится на уровне потолка в 7 % случаев. Тогда новое ожидаемое число битов для представления высоты можно получить с помощью следующего уравнения:

$$P_{НаЗемле} \times Bits_{НаЗемле} + P_{НаВысоте} \times Bits_{НаВысоте} + P_{НаПотолке} \times Bits_{НаПотолке} =$$
$$= 0{,}9 \times 2 + 0{,}07 \times 2 + 0{,}03 \times 33 = 2{,}93$$

Ожидаемое число битов теперь составляет 2,93, что на 1,3 бита меньше, чем при использовании одной только первой оптимизации. То есть оптимизация имеет смысл.

Существует множество форм энтропийного кодирования — от простых, как в примере, приведенном выше, до сложных и популярных, таких как кодирование Хаффмана, арифметическое кодирование, гамма-кодирование, кодирование длин серий (run length encoding) и др. Так же как и везде в разработке игр, перераспределение вычислительной мощности между энтропийным кодированием и игровыми операциями во многом является архитектурным решением. Ссылки на источники информации о других методах кодирования можно найти в разделе «Для дополнительного чтения».

Числа с фиксированной точкой

Молниеносная скорость вычислений с 32-разрядными вещественными числами является большим благом и показателем быстродействия в современную эпоху компьютерных вычислений. Однако необходимость выполнения в игре операций с вещественными числами еще не означает, что для передачи таких чисел по сети требуется использовать все 32 бита. Часто бывает полезно определить диапазон изменения чисел и требуемую точность представления значений и преобразовать их в формат с фиксированной точкой, чтобы при отправке данных можно было использовать как можно меньшее число битов. Для этого следует сесть за один стол с дизайнерами и проектировщиками игровой модели и точно определить потребности игры. Получив необходимые сведения, можно начинать строить систему, обеспечивающую максимальную эффективность.

Для примера вернемся вновь к полю `mLocation`. Процедура сериализации, сжимающая компонент *Y*, уже дает приличную экономию, однако она не затрагивает компоненты *X* и *Z*: для передачи каждого из них все еще используется 32 бита. Но в ходе нашего разговора с проектировщиками выяснилось, что размер игрового мира в игре «Robo Cat» составляет 4000 × 4000 единиц и центр игрового мира совпадает с началом системы координат. То есть минимальное значение компонентов *X* и *Z* равно –2000, а максимальное равно 2000. В ходе дальнейших дискуссий и тестирования игровой модели выяснилось, что для представления позиции на стороне клиента достаточно точности до 0,1 единицы. Это не говорит о том, что на стороне сервера позиция не должна вычисляться с более высокой точностью, просто для передачи значений клиенту достаточно указывать координаты с точностью до 0,1 единицы.

Эти пределы дают всю необходимую информацию, чтобы определить число битов, достаточное для сериализации этих значений. Следующая формула вычисляет общее число возможных значений компонента *X*:

$$(\textit{МаксимальноеЗначение} - \textit{МинимальноеЗначение})/\textit{Точность} + 1 = \\ = (2000 - (-2000))/0.1 + 1 = 40001.$$

То есть сериализуемый компонент теоретически может иметь 40001 различное значение. Если реализовать преобразование целых чисел, меньших 40001, в соответствующие вещественные значения, метод сможет сериализовать компоненты *X* и *Z*, просто замещая их соответствующими целочисленными значениями.

К счастью, это очень простая задача, которую иногда называют преобразованием в формат чисел с *фиксированной точкой*. Числа с фиксированной точкой — это числа, которые выглядят как целые, но в действительности представляют собой числа, равные отношению целого числа и некоторой константы. В данном случае константа равна требуемому уровню точности. На данный момент для сериализации методу требуется число битов, способное вместить любое целое число, меньшее 40001. Так как выражение $\log_2 40001$ дает 15,3, для сериализации каждого компонента *X* и *Z* достаточно всего 16 бит. Объединив все это, приходим к следующему коду:

```
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);
 ... // здесь выполняется запись компонента Y ...
}
```

Игра хранит компоненты вектора как полноценные вещественные числа, а при передаче их в сеть процедура сериализации преобразует их в числа с фиксированной точкой в диапазоне от 0 до 40 000, занимающие лишь 16 бит. Это решение экономит нам 32 бита на вектор, сокращая его ожидаемый размер с 96 до 35.

ПРИМЕЧАНИЕ На некоторых процессорах, таких как PowerPC в Xbox 360 и PS3, преобразование вещественных чисел в целые и обратно может оказаться довольно дорогостоящей операцией. Однако эти затраты окупаются экономией пропускной способности. Как это часто бывает, оптимизация — это компромисс, на который следует идти с учетом специфики разрабатываемой игры.

Сжатие геометрической информации

Сжатие на основе применения чисел с фиксированной точкой дает дополнительные преимущества при сериализации некоторых характерных игровых данных. Интересно отметить, что это всего лишь эффект применения теории информации: при наличии ограничений на возможные значения переменных для передачи такой информации требуется меньшее число битов. Этот прием можно использовать везде, где выполняется сериализация любых структур данных с известными ограничениями на их содержимое.

Многие геометрические типы данных как раз подпадают под это правило. В этом разделе обсуждаются кватернионы и матрицы преобразований. *Кватернион* (quaternion) — это структура данных, содержащая четыре вещественных числа и используемая для представления вращений в трехмерном пространстве. Обсуждение назначения кватернионов выходит далеко за рамки нашей книги, но в разделе «Для дополнительного чтения» вы найдете ссылки на дополнительные источники информации по этой теме. Для дальнейшего обсуждения важно знать, что для представления вращения выполняется нормализация кватернионов, в результате которой каждый компонент принимает значение в диапазоне от –1 до 1, а сум-

ма квадратов всех компонентов равна 1. Так как суммы квадратов компонентов имеют фиксированную величину, в сериализованное представление кватерниона достаточно включить лишь три компонента из четырех, а также единственный бит, представляющий знак четвертого компонента. Код десериализации сможет восстановить последний компонент, вычитая квадраты других компонентов из 1.

Кроме того, учитывая, что значения всех компонентов лежат между –1 и 1, сжатие компонентов с применением чисел с фиксированной точкой может дать еще больший эффект, если ограничение точности окажется приемлемым и не повлияет на работу игровой модели. Часто точности 16-битного целого числа, принимающего 65 535 возможных значений, вполне достаточно для представления чисел в диапазоне от –1 до 1. Это означает, что 4-компонентный кватернион, занимающий 128 бит памяти, можно сериализовать в 49-битное представление:

```
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;
 }
}
```

Прием сжатия геометрической информации можно также использовать для сериализации матриц *аффинных преобразований*. Матрица преобразований состоит из 16 вещественных чисел, но чтобы ее можно было назвать аффинной, она должна

разлагаться на перемещение в трех координатах, кватернион вращения и масштабирование в трех координатах, всего требуется 10 вещественных чисел. Применение энтропийного кодирования поможет еще больше сэкономить на пропускной способности при наличии дополнительных ограничений на типичные матрицы. Например, если типичная матрица не содержит информации о масштабировании, процедура сериализации отметит этот факт единственным битом. Если масштабирование носит однородный характер[1], процедура сообщит об этом еще одним битом и добавит только один компонент масштабного множителя вместо трех.

Простота сопровождения

Концентрация внимания исключительно на экономии пропускной способности может привести к созданию крайне запутанного программного кода. В некоторых случаях предпочтительнее иметь ясный и понятный код, простой в сопровождении, пусть и за счет потери эффективности.

Обобщение направления сериализации

Каждая новая структура данных или прием сжатия из обсуждавшихся в предыдущих разделах требовали реализации обоих методов — чтения и записи. Это означает необходимость не только реализации двух методов для каждого нового фрагмента функциональности, но и поддержания соответствия их друг другу: изменяя порядок записи переменной-члена, необходимо изменить и порядок ее чтения. Потребность в наличии двух таких тесно связанных методов для каждой структуры данных является надежным рецептом создания большой путаницы. Код был бы намного чище, если бы можно было оставить один метод для каждой структуры данных, обслуживающий сразу две операции — чтения и записи.

К счастью, механизм наследования и поддержка виртуальных методов предоставляют нам такую возможность. Один из способов заключается в том, чтобы определить классы `OutputMemoryStream` и `InputMemoryStream`, наследующие общий базовый класс `MemoryStream` с методом `Serialize`:

```
class MemoryStream
{
 virtual void Serialize(void* ioData,
       uint32_t inByteCount) = 0;
 virtual bool IsInput() const = 0;
};

class InputMemoryStream: public MemoryStream
{
 ...// выше находятся другие методы
 virtual void Serialize(void* ioData, uint32_t inByteCount)
 {
  Read(ioData, inByteCount);
```

[1] С одинаковым множителем по всем трем измерениям. — *Примеч. пер.*

```
 }

 virtual bool IsInput() const {return true;}
};

class OutputMemoryStream: public MemoryStream
{
 ...// выше находятся другие методы
 virtual void Serialize(void* ioData, uint32_t inByteCount)
 {
  Write(ioData, inByteCount);
 }

 virtual bool IsInput() const {return false;}
}
```

Методы `Serialize` в двух дочерних классах могли бы принимать указатель на данные и их объем и выполнять соответствующую операцию: чтение или запись. Используя метод `IsInput`, функция могла бы различать потоки ввода и вывода. Базовый класс `MemoryStream` мог бы реализовать шаблонный метод `Serialize`, предполагая наличие нешаблонных версий в подклассах:

```
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));
  }
 }
}
```

Шаблонный метод `Serialize` принимает обобщенный параметр с данными и выполняет операцию чтения или записи в зависимости от того, что делает нешаблонный метод `Serialize` дочернего класса. Такой подход упрощает замену каждой пары методов `Read` и `Write` соответствующим методом `Serialize`. Нестандартный метод `Serialize` должен принимать только параметр типа `MemoryStream` и выполнять чтение или запись, используя виртуальный метод `Serialize` потока. Таким образом,

единственный метод может обслуживать чтение и запись нестандартных классов, гарантируя, что реализации ввода и вывода всегда будут оставаться в полном соответствии друг с другом.

ВНИМАНИЕ Такая реализация менее эффективна, чем предыдущие, потому что выполняет вызовы виртуальных функций. Данную систему можно реализовать с применением шаблонов вместо виртуальных функций, чтобы вернуть часть утраченной эффективности, но мы оставим ее вам в качестве самостоятельного упражнения.

Сериализация, управляемая данными

Большая часть объектного кода, осуществляющего сериализацию, следует одному и тому же шаблону: сериализация выполняется для каждой переменной-члена класса объекта в отдельности. Иногда могут встречаться какие-то оптимизации, но общая структура кода обычно сохраняется без изменений. В действительности структура настолько унифицирована, что если бы во время выполнения имелась информация о переменных-членах в объектах, большую часть потребностей сериализации можно было бы удовлетворить единственным методом.

В некоторых языках, таких как C# и Java, имеются встроенные системы рефлексии (reflection), или отражения, позволяющие во время выполнения получать информацию о структуре класса. В C++, однако, для выявления членов классов во время выполнения требуется создавать собственную систему. К счастью, создать простейшую систему рефлексии не очень сложно (листинг 4.8).

Листинг 4.8. Простая система рефлексии

```
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` представляет элементарный тип переменной-члена. Данная система поддерживает только типы `int`, `float` и `string`, но ее легко расширить на любые другие элементарные типы.

Класс `MemberVariable` представляет единственную переменную-член в составном типе данных. Он хранит имя переменной-члена (для нужд отладки), ее тип и смещение в памяти относительно начала родительского типа данных. Информация о смещении играет важную роль: код сериализации сможет находить в памяти значение переменной-члена, складывая смещение с базовым адресом указанного объекта. Именно так выполняются операции чтения и записи данных в переменных-членах.

Наконец, класс `DataType` хранит все переменные-члены определенного класса. Для каждого класса, поддерживающего сериализацию, управляемую данными, создается свой экземпляр `DataType`. Опираясь на инфраструктуру поддержки рефлексии, следующий код загружает информацию об организации класса примера:

```
#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))
  });
 }
};
```

Здесь в качестве примера используется класс, хранящий информацию об объекте `RoboMouse`. В какой-то момент должна быть вызвана статическая функция `InitDataType`, чтобы инициализировать переменную-член `sDataType`. Эта функция создает экземпляр `DataType`, представляющий класс `MouseStatus`, и заполняет его вектор `mMemberVariables`. Обратите внимание на использование нестандартного макроса `OffsetOf` для вычисления смещения каждой переменной-члена. Встроенный в C++ макрос `offsetof` не предназначен для работы с классами, не являющимися простыми структурами данных. Из-за этого некоторые компиляторы сообщают об ошибке, встретив попытку применить `offsetof` к классам с виртуальными функциями или к другим типам, не являющимся структурами данных. Если класс не переопределяет унарный оператор `&`, не имеет виртуальных функций в иерархии и любых ссылочных переменных-членов, нестандартный макрос будет работать. В идеале вместо заполнения данных с информацией о типе вручную хорошо было бы иметь инструмент, анализирующий заголовочные файлы C++ и автоматически генерирующий эту информацию.

После этого реализация простой функции сериализации сводится к обходу переменных-членов в типе данных:

```
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;
  }
 }
}
```

Метод `GetOffset` каждой переменной-члена возвращает указатель на данные этого члена в экземпляре. Затем инструкция `switch` анализирует результат `GetPrimitiveType` и вызывает типизированную функцию `Serialize`, выполняющую фактическую сериализацию.

Этот прием можно сделать еще более эффективным, сохранив дополнительную информацию в классе `MemberVariable`. Например, для каждой переменной можно было бы сохранять число битов в сериализованном представлении для автоматического сжатия. Также можно было бы хранить типичные значения для поддержки процедурной реализации энтропийного кодирования.

В целом этот подход жертвует производительностью ради простоты: реализация имеет множество ветвлений, которые могут вызвать сброс конвейера[1], но требует писать меньше кода, а значит, ниже вероятность ошибки. Кроме того, система рефлексии может пригодиться для решения самых разных задач, не только для сериализации данных с целью передачи их по сети. Ее можно использовать для сериализации на диск, сборки мусора, создания редактора объектов с графическим интерфейсом и решения множества других проблем.

В заключение

Сериализация — это процесс преобразования сложных структур данных в линейную последовательность байтов, которую можно отправить другому узлу в сети. Самое простое решение — копирование структуры в буфер байтов с помощью `memcpy` — часто оказывается неработоспособным. Поток данных, основная рабочая лошадь сериализации, позволяет сериализовать сложные структуры данных, которые могут включать ссылки на другие структуры данных, а затем восстанавливать ссылки после десериализации.

Существует несколько приемов повышения эффективности сериализации. Разреженные структуры данных можно преобразовать в более компактную форму. Типовые значения в переменных-членах можно сжать без потери точности, используя энтропийное кодирование. Структуры данных с геометрической или другой подобной информацией также можно сжать без потери точности за счет выявления ограничений и передачи только компонентов, необходимых для воссоздания структуры. Если допустима некоторая потеря точности, вещественные числа можно преобразовывать в числа с фиксированной точкой исходя из известного диапазона значений и необходимой точности.

Эффективность часто достигается за счет простоты сопровождения, и иногда имеет смысл поступиться некоторой долей производительности, чтобы упростить сопровождение системы сериализации. Методы `Read` и `Write` для структур данных можно свернуть в единственный метод `Serialize`, выполняющий чтение или запись в зависимости от направления потока, которым он оперирует. А сериализацию можно сделать управляемой данными, используя для этого метаданные, сгенерированные вручную или автоматически, без необходимости писать специализированные функции чтения и записи для каждой структуры.

Теперь у вас есть все, что необходимо для упаковки и отправки объектов удаленному узлу. Следующая глава повествует о том, как конструировать данные, чтобы удаленный узел смог создать или найти соответствующий объект для приема данных, и как эффективно осуществлять частичную сериализацию в том случае, когда достаточно передать только часть данных из объекта.

[1] Имеется в виду конвейер команд в микропроцессоре. — *Примеч. пер.*

Вопросы для повторения

1. Почему небезопасно просто скопировать объект в буфер и послать его удаленному узлу?
2. Что такое «порядок следования байтов»? Почему его необходимо учитывать при сериализации данных? Расскажите, как решаются проблемы порядка следования байтов при сериализации данных.
3. Опишите, как эффективно сжать разреженную структуру данных.
4. Опишите два способа сериализации объектов с указателями в них. Приведите примеры, когда эти способы можно использовать.
5. Что такое «энтропийное кодирование»? Приведите простой пример его использования.
6. Расскажите, как сэкономить пропускную способность, сериализуя вещественные числа в числа с фиксированной точкой.
7. Объясните подробно, почему реализация функции `WriteBits`, представленная в этой главе, правильно работает только на платформах с обратным порядком байтов. Реализуйте решение, которое будет работать также на платформах с прямым порядком байтов.
8. Реализуйте метод `MemoryOutputStream::Write(const unordered_map<int, int>&)`, записывающий в поток значения из карты соответствий между целыми числами.
9. Напишите парный ему метод `MemoryOutputStream::Read(unordered_map<int, int>&)`.
10. Создайте шаблонную реализацию метода `MemoryOutputStream::Read` из предыдущего задания, так чтобы он правильно обрабатывал шаблон `<tKey, tValue> unordered_map<tKey, tValue>`.
11. Реализуйте эффективные функции `Read` и `Write` для матриц аффинных преобразований, используя тот факт, что масштаб обычно равен 1, а когда он не равен 1, масштабирование выполняется однородно.
12. Реализуйте модуль сериализации с обобщенным методом, опирающимся на шаблоны вместо виртуальных функций.

Для дополнительного чтения

Bloom, Charles. (1996, август 1). «Compression: Algorithms: Statistical Coders». Доступно по адресу: *http://www.cbloom.com/algs/statisti.html*. Проверено 28 января 2016.

Blow, Jonathan. (2004, январь 17). «Hacking Quaternions». Доступно по адресу: *http://number-none.com/product/Hacking%20Quaternions/*. Проверено 28 января 2016.

Ivancescu, Gabriel. (2007, декабрь 21). «Fixed Point Arithmetic Tricks». Доступно по адресу: *http://x86asm.net/articles/fixed-point-arithmetic-and-tricks/*. Проверено 28 января 2016.

5 Репликация объектов

Сериализация данных в объектах — это лишь первый шаг в передаче информации между узлами. Эта глава посвящена обобщенной схеме репликации, используемой для поддержки *синхронизации миров* и объектов между удаленными процессами.

Состояние мира

Успешная многопользовательская игра должна давать игрокам ощущение, что они действуют в одном игровом мире. Когда один игрок открывает дверь или уничтожает зомби, все игроки поблизости должны увидеть открытую дверь или убитого зомби. Многопользовательские игры позволяют ощутить общность за счет определения *состояния мира* на всех узлах и обмена любой информацией, необходимой для поддержания соответствия этого состояния между узлами.

Существует несколько способов создания и поддержания единства состояний миров между удаленными узлами, эти способы зависят от топологии сети, о чем подробнее рассказывается в главе 6 «Топологии сетей и примеры игр». Один из них заключается в организации передачи сервером состояния мира всем подключенным клиентам. Клиенты принимают эту информацию и изменяют состояние своего мира, приводя его в соответствие с полученной информацией. То есть все игроки на клиентских узлах в конечном счете наблюдают одно и то же состояние игрового мира.

Допустим, что некая объектно-ориентированная игра имеет объектную игровую модель, тогда состояние игрового мира можно определить как совокупность состояний всех игровых объектов в этом мире. То есть задачу передачи состояния мира можно разложить на множество задач передачи состояний всех объектов мира.

В этой главе рассматривается задача передачи состояния объекта между узлами с целью обеспечения согласования состояния мира на множестве удаленных узлов.

Репликация объекта

Акт передачи состояния объекта от одного узла к другому называют *репликацией* (replication). Репликация включает в себя не только сериализацию, обсуждавшуюся в главе 4 «Сериализация объектов». Для успешной репликации объекта узел должен выполнить три подготовительных шага перед сериализацией его внутреннего состояния:

1. Отметить пакет как пакет, содержащий состояние объекта.
2. Уникально идентифицировать реплицируемый объект.
3. Указать класс реплицируемого объекта.

Сначала узел-отправитель отмечает пакет как содержащий состояние объекта. Узлы могут обмениваться информацией не только с целью репликации объектов, поэтому было бы небезопасно предполагать, что каждая входящая датаграмма содержит данные для репликации объекта. Также полезно определить перечисление `PacketType` для идентификации типов пакетов. Пример такого перечисления приводится в листинге 5.1.

Листинг 5.1. Перечисление PacketType

```
enum PacketType
{
  PT_Hello,
  PT_ReplicationData,
  PT_Disconnect,
  PT_MAX
};
```

Для каждого отправляемого пакета узел должен сначала сериализовать соответствующее значение типа `PacketType` в поток `MemoryStream` пакета. В этом случае принимающий узел сможет сразу же прочитать тип пакета из входящей датаграммы и определить, как она должна обрабатываться. Традиционно первый пакет в сеансе взаимодействий между узлами отмечен как пакет «приветствия» и используется для инициации взаимодействия, распределения памяти для состояния и, возможно, начинает процедуру аутентификации. Эти пакеты отличаются значением `PT_Hello` в первом байте входящей датаграммы. Аналогично, значение `PT_Disconnect` в первом байте указывает, что пакет является запросом, начинающим процедуру отключения. Значение `PT_MAX` используется кодом, чтобы узнать максимальное число элементов в перечислении `PacketType`. Для репликации объекта узел-отправитель сериализует значение `PT_ReplicationData`.

Далее, узел-отправитель должен идентифицировать сериализованный объект. Благодаря этому принимающий узел сможет определить, имеется ли у него копия полученного объекта. Если имеется, он сможет записать сериализованное состояние в готовый объект, не создавая новый. Как вы наверняка помните, класс `LinkingContext`, описанный в главе 4, уже опирается на наличие в объектах уникального идентификатора. Эти идентификаторы можно использовать для поиска объектов с целью репликации состояния. Следует отметить, что класс

LinkingContext можно расширить, как показано в листинге 5.2, реализовав присвоение уникальных сетевых идентификаторов объектам, которые еще не имеют их.

Листинг 5.2. Расширенный класс 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);
  }

  // остался без изменений ...
  GameObject* GetGameObject(uint32_t inNetworkId);

private:
  std::unordered_map<uint32_t, GameObject*> mNetworkIdToGameObjectMap;
  std:: unordered_map<const GameObject*, uint32_t>
      mGameObjectToNetworkIdMap;

  uint32_t mNextNetworkId;
}
```

Новая переменная-член `mNextNetworkId` хранит следующий неиспользуемый сетевой идентификатор и увеличивается всякий раз, когда он задействуется. Так как переменная хранит 4-байтное целое число без знака, можно предположить, что она никогда не переполнится: в случаях, когда в процессе игры может понадобиться реплицировать более 4 миллиардов объектов, необходимо реализовать более сложную систему. А пока предположим, что простой наращиваемый счетчик надежно гарантирует уникальность сетевых идентификаторов.

Когда узел будет готов записать идентификатор из `inGameObject` в пакет с состоянием объекта, он вызовет `mLinkingContext->GetNetworkId(inGameObject, true)`, сообщив экземпляру `LinkingContext`, что тот должен сгенерировать сетевой идентификатор, если это необходимо. Затем он запишет идентификатор в пакет вслед за байтом `PacketType`. Когда удаленный узел получит пакет, он прочитает идентификатор и с помощью своего экземпляра `LinkingContext` попытается найти требуемый объект. Если объект будет найден, узел сможет десериализовать данные непосредственно в него. В противном случае будет создан новый объект.

Чтобы создать новый объект, удаленный узел должен иметь информацию о классе объекта. Узел-отправитель предоставляет эту информацию, сериализуя некоторый идентификатор класса вслед за идентификатором объекта. Одно из решений «в лоб» состоит в том, чтобы выбрать идентификатор класса из множества, используя механизм динамической идентификации типа, как показано в листинге 5.3. После этого принимающий узел мог бы использовать инструкцию `switch`, как показано в листинге 5.4, чтобы создать экземпляр нужного класса по идентификатору класса.

Листинг 5.3. Жестко определенная, тесно связанная идентификация класса

```
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'));
  }
}
```

Листинг 5.4. Жестко определенное, тесно связанное создание экземпляров

```
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;
}
```

Несмотря на работоспособность, это решение является очень несовершенным по нескольким причинам. Во-первых, здесь используется оператор `dynamic_cast`, требующий включения встроенной поддержки RTTI[1] в C++. Механизм RTTI часто отключается в играх, так как требует дополнительной памяти для хранения информации о каждом классе полиморфного типа. Кроме того, данное решение образует тесную связь между системой игровых объектов и системой репликации. Каждый раз, когда возникнет необходимость добавить в игровую модель новый класс и обеспечить возможность его репликации, вам придется внести изменения в обе функции — `WriteClassType` и `CreateGameObjectFromStream` — в сетевом коде. Об этом легко забыть и в результате получить несогласованный программный код. Кроме того, если появится желание повторно использовать систему репликации в новой игре, для этого понадобится полностью переписать эти функции, которые ссылаются на игровой код в прежней игре. Наконец, тесная взаимозависимость программных компонентов существенно усложняет модульное тестирование, так как тесты не смогут загрузить сетевой модуль, не загрузив игровой. В том, что игровой код зависит от сетевого, нет ничего криминального, но сетевой код никогда не должен зависеть от игрового.

Ослабить связь между игровым и сетевым кодом можно, обобщив идентификацию объектов и создав в системе процедуры, использующие реестр создаваемых объектов.

Реестр создаваемых объектов

Реестр создаваемых объектов отображает идентификатор класса в функцию, создающую объект данного класса. С помощью реестра сетевой код может отыскивать функцию по идентификатору и вызывать ее для создания требуемого объекта. Если игра включает систему рефлексии, вероятно, у вас уже реализована система идентификации, если нет — ее несложно создать.

Каждый реплицируемый класс должен быть подготовлен к сохранению в реестре. Для начала каждому классу должен быть присвоен уникальный идентификатор и сохранен в статической константе с именем `kClassId`. Для идентификации клас-

[1] RTTI (Run-Time Type Identification — информация о типе времени выполнения) — механизм, позволяющий определять типы объектов во время выполнения. — *Примеч. пер.*

сов можно было бы использовать глобально уникальные идентификаторы GUID, чтобы исключить возможное их совпадение, однако 128-битные идентификаторы могут оказаться слишком обременительными для небольшого подмножества классов, подлежащих репликации. Неплохой альтернативой могут оказаться четырехсимвольные литералы на основе имени класса с последующей проверкой конфликтов при записи классов в реестр. Последняя альтернатива — создавать идентификаторы классов во время компиляции с использованием инструмента сборки, автоматически генерирующего уникальные коды.

ВНИМАНИЕ Поддержка четырехсимвольных литералов зависит от реализации. Определение 32-разрядных значений в виде четырехсимвольных литералов, таких как `'DXT5'` или `'GOBJ'`, дает простую и понятную систему идентификации. Такой способ хорош еще тем, что идентификаторы ясно различимы в дампе памяти пакета. По этой причине многие сторонние движки, от «Unreal» до «C4», используют такие значения, как маркеры и идентификаторы. К сожалению, подобный подход определяется стандартом C++ как зависящий от реализации, то есть не все компиляторы одинаково преобразуют строковые литералы в целочисленное представление. Большинство компиляторов, включая GCC и Visual Studio, используют одинаковые соглашения, но если вы собираетесь использовать многосимвольные литералы для взаимодействий между процессами, скомпилированными разными компиляторами, убедитесь сначала, что оба компилятора преобразуют литералы одинаково.

После присвоения каждому классу уникального идентификатора добавьте в `GameObject` виртуальную функцию `GetClassId`. Переопределите эту функцию в каждом дочернем классе, наследующем `GameObject`, чтобы она возвращала идентификатор своего класса. Наконец, добавьте в каждый дочерний класс статическую функцию, создающую и возвращающую экземпляр класса. В листинге 5.5 показаны `GameObject` и два дочерних класса, подготовленные к записи в реестр.

Листинг 5.5. Классы, подготовленные к записи в реестр создаваемых объектов

```
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();}
  //...
};
```

Обратите внимание, что каждый дочерний класс должен реализовывать виртуальную функцию `GetClassId`. Даже при том, что код выглядит идентичным, возвращаемые значения отличаются, потому что отличаются константы `kClassId`. Так как во всех классах используется один и тот же код, некоторые разработчики предпочитают генерировать его с помощью макроса препроцессора. Вообще говоря, сложные макросы препроцессора лучше не использовать, так как современные отладчики не очень хорошо работают с ними, но они помогают уменьшить вероятность ошибок, которые могут возникать при многократном копировании фрагментов кода. Кроме того, если скопированный код впоследствии потребуется изменить, при использовании макроса достаточно будет изменить сам макрос, чтобы распространить изменения на все классы. Листинг 5.6 демонстрирует, как использовать макросы в таких случаях.

Листинг 5.6. Классы, подготовленные к записи в реестр создаваемых объектов с использованием макроса

```
#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)
  //...
};
```

Символы \ в конце каждой строки в определении макроса сообщают компилятору, что определение продолжается на следующей строке.

Покончив с системой идентификации классов, создадим `ObjectCreationRegistry` для хранения карты соответствий идентификаторов и функций создания классов. Игровой код, полностью независимый от системы репликации, может заполнить эту карту информацией о классах, поддерживающих репликацию, как показано в листинге 5.7. С технической точки зрения класс `ObjectCreationRegistry` не обязательно должен быть *синглтоном* (singleton), как показано в листинге 5.7, достаточно сделать его доступным для игрового и сетевого кода.

Листинг 5.7. Класс ObjectCreationRegistry, синглтон и карта

```
typedef GameObject* (*GameObjectCreationFunc)();

class ObjectCreationRegistry
{
public:
  static ObjectCreationRegistry& Get()
  {
    static ObjectCreationRegistry sInstance;
    return sInstance;
  }

  template<class T>
  void RegisterCreationFunction()
  {
    // проверить отсутствие дубликата идентификатора класса
    assert(mNameToGameObjectCreationFunctionMap.find(T::kClassId) ==
        mNameToGameObjectCreationFunctionMap.end());
    mNameToGameObjectCreationFunctionMap[T::kClassId] =
      T::CreateInstance;
  }

  GameObject* CreateGameObject(uint32_t inClassId)
  {
    // добавьте проверку на наличие ошибки — в настоящее время
    // происходит аварийное завершение, если функция не найдена
    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>();
}
```

Тип `GameObjectCreationFunc` — это указатель на функцию с сигнатурой, соответствующей сигнатуре статического метода `CreateInstance` в каждом классе. `RegisterCreationFunction` — это шаблон, используемый для предотвращения несовпадения идентификатора класса с функцией создания экземпляра. В код, осуществляющий инициализацию игры, добавьте вызовы `RegisterObjectCreation` для заполнения реестра идентификаторами классов и ссылками на функции создания экземпляров.

Теперь, когда узлу-отправителю понадобится записать в пакет идентификатор класса, наследующего `GameObject`, он сможет просто вызвать его метод `GetClassId`. Когда принимающему узлу понадобится создать экземпляр указанного класса, он просто вызовет метод `CreateGameObject` объекта реестра и передаст ему идентификатор класса.

В действительности данная система является нестандартной версией системы RTTI, встроенной в язык C++. Будучи написанной вручную, она дает более полный контроль над ресурсами памяти, размером типа идентификатора и совместимостью с разными компиляторами, чем стандартный оператор `typeid` в C++.

СОВЕТ Если ваша игра использует систему рефлексии, подобную той, что описана в разделе «Сериализация, управляемая данными» в главе 4, можете воспользоваться этой системой вместо описанной здесь. Просто в каждый класс, наследующий `GameObject`, добавьте виртуальную функцию `GetDataType`, которая возвращала бы `DataType` объекта вместо идентификатора класса. Затем добавьте уникальный идентификатор в каждый `DataType` и функцию создания экземпляра. При такой организации реестр будет отображать не идентификаторы классов в функции, а идентификаторы типов данных в `DataType`. Чтобы выполнить репликацию в этом случае, получите `DataType` вызовом метода `GetDataType` и сериализуйте идентификатор `DataType`. Чтобы создать экземпляр, найдите в реестре `DataType` по идентификатору и используйте функцию создания экземпляра из `DataType`. Это решение поможет сделать `DataType` доступным для обобщенной сериализации с принимающей стороны репликации.

Несколько объектов в пакете

Как уже отмечалось ранее, эффективность использования полосы пропускания тем выше, чем ближе размер пакета к величине MTU. Но не все объекты имеют такой большой размер, поэтому для повышения эффективности можно организовать отправку в одном пакете сразу нескольких объектов. Для этого узел, отметивший пакет признаком `PT_ReplicationData`, просто должен повторить следующие операции с каждым объектом:

1. Записать сетевой идентификатор объекта.
2. Записать идентификатор класса объекта.
3. Записать сериализованное представление объекта.

Когда принимающий узел завершит десериализацию объекта, все неиспользованные данные в пакете должны интерпретироваться им как еще один объект. То есть узел должен повторять процедуру извлечения объекта из пакета, пока в нем не останется неиспользованных байтов.

Простая репликация состояния мира

Имея законченную систему репликации объектов, несложно организовать репликацию состояния всего мира путем репликации всех объектов в этом мире. Если игровой мир относительно невелик, как в оригинальной игре «Quake», тогда состояние всего мира можно уместить в один пакет. В листинге 5.8 приводится исходный код диспетчера репликации, который передает состояние всего мира именно таким способом.

Листинг 5.8. Репликация состояния мира

```
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)
{
  // записать идентификатор игрового объекта
  inStream.Write(mLinkingContext->GetNetworkId(inGameObject, true));

  // записать идентификатор класса игрового объекта
  inStream.Write(inGameObject->GetClassId());

  // записать данные из игрового объекта
  inGameObject->Write(inStream);
}

void ReplicationManager::ReplicateWorldState(
  OutputMemoryBitStream& inStream,
  const vector<GameObject*>& inAllObjects)
{
  // отметить пакет как содержащий данные для репликации
  inStream.WriteBits(PT_ReplicationData, GetRequiredBits<PT_MAX>::Value );

  // записать все объекты
  for(GameObject* go: inAllObjects)
  {
    ReplicateIntoStream(inStream, go);
  }
}
```

`ReplicateWorldState` — это общедоступная (public) функция, которую вызывающий код может использовать для записи данных из коллекции объектов в исходящий

поток. Сначала она записывает метку, сообщающую, что данные предназначены для репликации, и затем с помощью приватного метода `ReplicateIntoStream` записывает все объекты по отдельности. С помощью объекта `LinkingContext` `ReplicateIntoStream` записывается сетевой идентификатор каждого объекта и с помощью виртуального метода `GetClassId` — идентификатор класса объекта. Затем вызывается виртуальная функция `Write` игрового объекта для сериализации фактических данных.

ПОЛУЧЕНИЕ ЧИСЛА БИТОВ, НЕОБХОДИМЫХ ДЛЯ СЕРИАЛИЗАЦИИ ЗНАЧЕНИЯ

Как вы наверняка помните, битовый поток позволяет сериализовать значения полей в произвольное число битов. Число битов должно быть достаточно большим, чтобы вместить максимально возможное значение поля. При сериализации перечислений компилятор может вычислить наиболее подходящее число битов на этапе компиляции, устраняя вероятность ошибки при добавлении или удалении элементов перечислений. Хитрость заключается в том, чтобы снабдить последний элемент перечисления окончанием `_MAX`. Например, в перечислении `PacketType` последний элемент называется `PT_MAX`. В этом случае порядковый номер элемента `_MAX` автоматически будет увеличиваться или уменьшаться при добавлении или удалении элементов, и вы легко сможете определить максимальное значение в перечислении.

Далее метод `ReplicateWorldState` сможет передать значение этого последнего элемента в шаблонном аргументе методу `GetRequiredBits` (см. далее) для вычисления числа битов, необходимых для представления максимального значения типа пакета. Для большей эффективности, чтобы все вычисления выполнялись на этапе компиляции, здесь используется прием, известный как *шаблонное метапрограммирование* (template metaprogramming), который многие считают темной стороной искусства программирования на C++. Язык шаблонов C++ настолько сложен, что фактически реализует вселенную Тьюринга и позволяет компилятору вычислить результат произвольной функции — при условии, что входные ее параметры известны на этапе компиляции. В данном случае код вычисления числа битов, необходимых для представления максимального значения, имеет следующий вид:

```
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};
};
```

Шаблонное метапрограммирование не имеет аналогов операторам циклов, поэтому вместо итераций должна использоваться рекурсия. То есть `GetRequiredBits` полагается на рекурсивный вызов `GetRequiredBitsHelper`, выполняющий поиск наибольшего установленного бита в значении аргумента и тем самым вычисляющий число битов, необходимых для представления значения. Это делается путем увеличения аргумента `tBits` на единицу с каждым сдвигом аргумента `tValue` на один бит вправо. Когда значение `tValue` достигнет `0`, будет вызвана специализация базового случая, которая просто вернет накопленное значение `tBits`.

С выходом стандарта C++11 появилась возможность использовать ключевое слово `constexpr`, предоставляющее те же возможности, что и шаблонное метапрограммирование, но с меньшими сложностями. Однако к моменту написания этих строк данный стандарт поддерживался не всеми современными компиляторами (например, Visual Studio 2013), поэтому для совместимости безопаснее пока использовать прием на основе шаблонного метапрограммирования.

Получив пакет с информацией для репликации состояния, принимающий узел передаст его диспетчеру репликации, который выполнит обход всех сериализованных объектов в пакете. Если требуемый игровой объект отсутствует, клиент создаст его и десериализует состояние в него. Если игровой объект существует, клиент обнаружит его и десериализует состояние в существующий объект. Закончив обработку пакета, он уничтожит все локальные игровые объекты, для которых не нашлось данных в пакете, так как отсутствие данных говорит о том, что соответствующий игровой объект отсутствует в игровом мире узла-отправителя. В листинге 5.9 приводится дополнительная часть диспетчера репликации, отвечающая за обработку входящего пакета, отмеченного как пакет с данными для репликации.

Листинг 5.9. Репликация состояния мира

```
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);
  }

  // теперь выполнить обход mObjectsReplicatedToMe.
```

```
  // если в пакете отсутствует информация для объекта,
  // уничтожить его
  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);
  }

  // прочитать изменения
  go->Read(inStream);

  // вернуть игровой объект, чтобы запомнить его
  // как принятый в пакете
  return go;
}
```

Когда код, принимающий пакет, прочитает его тип и определит, что пакет содержит данные для репликации, он сможет передать поток в `ReceiveWorld`. Метод `ReceiveWorld` вызывает `ReceiveReplicatedObject`, чтобы извлечь объект, и запоминает каждый полученный объект в наборе. После извлечения всех объектов он проверяет, имеются ли объекты, принятые в прошлый раз и отсутствующие в этот, и уничтожает их, чтобы добиться полной синхронизации миров. Передача и прием состояния мира таким способом реализуются просто, но при этом требуется, чтобы все состояние мира умещалось в одном пакете. Для поддержки более обширных миров необходим альтернативный способ репликации состояния.

Изменения в состоянии мира

Поскольку каждый узел хранит собственную копию состояния мира, нет необходимости пытаться уместить его целиком в один пакет. Вместо этого отправитель может создавать пакеты, представляющие изменения в состоянии мира, а получа-

тель — применять эти изменения к собственной копии. В этом случае отправитель может использовать множество пакетов для синхронизации состояния огромного мира с удаленным узлом.

Когда репликация выполняется таким способом, каждый пакет должен сообщать, что он содержит *различие в состоянии мира* (world state delta). Так как состояние мира складывается из состояний объектов, различие в состоянии мира складывается из *различий в состояниях объектов* (object state delta), по одному для каждого изменившегося объекта. Каждое различие в состоянии объекта представляет одно из трех действий репликации:

1. Создание игрового объекта.
2. Изменение игрового объекта.
3. Уничтожение игрового объекта.

Репликация различия в состоянии объекта похожа на репликацию полного состояния объекта, за исключением того, что отправитель должен указать в пакете, какое действие над объектом требуется выполнить. На данном этапе набор флагов и признаков, предшествующих сериализованным данным, становится настолько сложным, что имеет смысл создать *заголовок репликации* (replication header), включающий сетевой идентификатор объекта, действие репликации и класс, если необходимо. В листинге 5.10 показана возможная реализация такого заголовка.

Листинг 5.10. Заголовок репликации

```
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);
  }
};
```

Методы `Read` и `Write` помогают организовать сериализацию заголовка в поток пакета перед данными объекта. Обратите внимание, что нет необходимости сериализовать идентификатор класса объекта в случае уничтожения этого объекта.

Когда отправителю потребуется выполнить репликацию набора различий в состояниях объектов, он создаст поток в памяти, отметит его как пакет `PT_ReplicationData`, а затем для каждого изменения сериализует `ReplicationHeader` и соответствующий объект. Класс `ReplicationManager` должен иметь три дополнительных метода для создания, изменения и удаления реплицируемых объектов, как показано в листинге 5.11. Они берут на себя обязанность создания экземпляра `ReplicationHeader`, благодаря чему он остается недоступным за пределами `ReplicationManager`.

Листинг 5.11. Репликация различий в состояния объектов

```
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);
}
```

Теперь, обрабатывая пакет, получатель должен уметь применять все три операции. В листинге 5.12 демонстрируется, как это реализовано.

Листинг 5.12. Обработка действий репликации

```
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);

      // может так сложиться, что пакет с операцией
      // создания еще не был принят, поэтому создать
      // заготовку и сериализовать в нее, чтобы продолжить чтение
      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:
      // в данном случае ничего не делается
      break;
  }
}
```

После идентификации пакета как содержащего состояние объекта получатель перебирает заголовки и извлекает сериализованные данные. Если заголовок требует создать объект, получатель убеждается, что такого объекта еще не существует. Если объект действительно отсутствует, он создает его, заполняя его сериализованными данными.

Если заголовок требует изменить объект, получатель находит требуемый объект и извлекает данные в него. По разным причинам, в том числе и из-за ненадежной работы сети, может так сложиться, что получатель не обнаружит целевой игровой объект. В этом случае он все равно должен обработать остальные объекты в пакете, то есть продвинуться вперед в потоке данных, прочитав данные, соответствующие заголовку. Для этого создается временный объект, данные из потока извлекаются в этот объект, и затем он удаляется. Если такой подход окажется слишком неэффективным или невозможным из-за особенностей конструирования объекта, в заголовок сериализации объекта можно добавить поле, определяющее объем сериализованных данных. В этом случае получатель сможет определить объем данных для отсутствующего объекта и пропустить их, прочитав из потока требуемый объем.

ВНИМАНИЕ Репликация части состояния мира или объекта возможна, только если отправитель имеет точную информацию о текущем состоянии мира получателя. Обладая этой информацией, отправитель сможет определить, какие изменения следует реплицировать. Так как Интернет по своей природе ненадежен, нельзя просто предположить, что состояние мира на стороне получателя соответствует последним пакетам, переданным отправителем. Либо узлы должны посылать пакеты с использованием протокола TCP, гарантирующего надежность, либо, для обеспечения надежности, они должны использовать протокол прикладного уровня, основанный на UDP. Эта тема обсуждается в главе 7 «Задержки, флуктуации и надежность».

Репликация части состояния объекта

При передаче изменений не всегда требуется отправлять все свойства объекта. Для отправителя может быть желательно сериализовать только часть свойств, изменившихся с момента последнего обновления. Для поддержки этой возможности можно использовать битовое поле, представляющее сериализованные свойства. Каждый бит может представлять свойство или группу свойств. Например, класс

MouseStatus из главы 4 мог бы использовать перечисление из листинга 5.13, чтобы связать свойства с определенными битами.

Листинг 5.13. Перечисление свойств MouseStatus

```
enum MouseStatusProperties
{
  MSP_Name   = 1 << 0,
  MSP_LegCount = 1 << 1,
  MSP_HeadCount = 1 << 2,
  MSP_Health  = 1 << 3,
  MSP_MAX
};
```

Значения этого перечисления можно объединять с помощью поразрядной операции ИЛИ (OR), чтобы представить несколько свойств. Например, различие в состоянии объекта, содержащее значения для свойств mHealth и mLegCount, могло содержать в битовом поле значение MSP_Health | MSP_LegCount. Обратите внимание, что битовое поле с единицами во всех битах указывает на необходимость сериализации всех свойств.

Метод Write класса должен извлекать битовое поле и на его основе определять, какие свойства требуется сериализовать в поток. В листинге 5.14 приводится пример реализации этого метода для класса MouseStatus.

Листинг 5.14. Определение свойств для записи с помощью битового поля

```
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);
  }
}
```

Перед записью свойств метод записывает в поток значение inProperties, чтобы процедура десериализации смогла прочитать только записанные свойства. Затем он проверяет отдельные биты в поле и записывает необходимые свойства. Процедура десериализации демонстрируется в листинге 5.15.

Листинг 5.15. Десериализация части состояния объекта

```
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);
  }
}
```

Метод `Read` сначала читает поле `writtenProperties`, чтобы потом, руководствуясь полученной информацией, выполнить десериализацию нужных свойств.

Передачу части состояния объекта на основе битового поля можно также использовать в более обобщенных, двунаправленных и управляемых данными процедурах сериализации, рассматривавшихся в главе 4. В листинге 5.16 приводится реализация `Serialize`, дополненная поддержкой битового поля с целью репликации части состояния объекта.

Листинг 5.16. Двунаправленная, управляемая данными сериализация части состояния объекта

```
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;
      }
    }
  }
}
```

Вместо определения значений битов с использованием перечисления, созданного вручную, реализация, управляемая данными, интерпретирует индексы битов как индексы переменных-членов. Обратите внимание, что здесь сразу же вызывается метод `Serialize` со значением `inProperties`. При работе с потоком вывода этот вызов запишет битовое поле в поток, а при работе с потоком ввода прочитает индексы свойств в переменную, затерев все, что было передано в ней при вызове. Это вполне нормальное поведение, так как операция ввода должна использовать сериализованное битовое поле, единичные разряды в котором соответствуют сериализованным свойствам. Если потребуется организовать сериализацию более 32 свойств, определите битовое поле с типом `uint64_t`. Если число свойств может превысить 64, подумайте о том, чтобы поставить в соответствие отдельным битам сразу несколько свойств, или разбейте класс.

Удаленный вызов процедур в виде сериализованных объектов

В сложной многопользовательской игре узлу может понадобиться передать какую-то другую информацию, не являющуюся объектом. Представьте ситуацию, когда один узел должен передать другому узлу звук взрыва или отобразить вспышку на экране другого узла. Действия, подобные этим, лучше передавать с использованием *удаленных вызовов процедур* (Remote Procedure Call, RPC). Удаленный вызов процедуры — это такое действие на одном узле, которое вынуждает один или несколько других удаленных узлов выполнить некоторую процедуру. С этой целью было разработано множество протоколов прикладного уровня, от текстовых, таких как XML-RPC, до двоичных, таких как ONC-RPC. Однако если игра уже включает систему репликации объектов, описанную в этой главе, удаленный вызов процедур можно реализовать поверх нее.

Каждый вызов процедуры можно представить как уникальный объект с переменными-членами для передачи параметров. Чтобы вызвать процедуру на удаленном узле, вызывающий узел должен послать объект соответствующего типа с заполненными переменными-членами. Например, для функции `PlaySound`,

```
void PlaySound(const string& inSoundName, const Vector3& inLocation,
        float inVolume);
```

можно определить структуру `PlaySoundRPCParams` с тремя переменными-членами:

```
struct PlaySoundRPCParams
{
  string mSoundName;
  Vector3 mLocation;
  float mVolume;
};
```

Чтобы вызвать `PlaySound` на удаленном узле, вызывающий узел должен создать объект `PlayerSoundRPCParams`, присвоить значения переменным-членам и сериализовать объект в пакет с состоянием объекта. Однако такой подход может стать причиной спагетти-подобного кода при использовании большого числа удаленных вызовов, а также необходимости просмотра большого числа сетевых идентификаторов объектов, что в действительности является излишним, потому что объекты удаленных вызовов процедур не нуждаются в уникальной идентификации.

Более простое решение заключается в создании модульной обертки вокруг системы RPC и ее интеграции с системой репликации. Для этого сначала добавьте дополнительный тип действия репликации — `RA_RPC`. Этот признак позволит идентифицировать сериализованные данные как удаленный вызов процедуры и предоставит удаленному узлу возможность сразу передать данные в специализированный модуль обработки RPC. Он также сообщит функции сериализации `ReplicationHeader`, что действие не нуждается в сетевом идентификаторе и его не требуется сериализовать. Когда метод `ProcessReplicationAction` в классе `ReplicationManager` идентифицирует операцию как `RA_RPC`, он должен будет передать пакет в модуль RPC для дальнейшей обработки.

Модуль RPC должен хранить структуру данных, отображающую каждый идентификатор RPC в функцию, которая выполнит десериализацию параметров и вызовет соответствующую процедуру. В листинге 5.17 показан пример реализации такого класса `RPCManager`.

Листинг 5.17. Пример реализации 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;
};
```

В этом примере каждый удаленный вызов идентифицируется четырехсимвольным кодом в виде целого числа без знака. При необходимости RPCManager мог бы использовать полноценные строки: хотя строки допускают большее число возможных комбинаций, они занимают большую долю полосы пропускания, чем целые числа. Обратите внимание на сходство с реестром объектов. Регистрация функций в ассоциативном массиве — распространенный способ ослабить связь между взаимозависимыми системами.

Когда ReplicationManager идентифицирует действие как RA_RPC, он передаст полученный поток модулю RPC, который затем развернет его и вызовет нужную функцию. Для поддержки этой возможности игровой код должен зарегистрировать функцию для каждого удаленного вызова. В листинге 5.18 показано, как регистрируется функция PlaySound.

Листинг 5.18. Регистрация удаленного вызова процедуры

```
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 — это связывающая функция, выполняющая десериализацию параметров и передающая их в вызов PlaySound. Игровой код должен вызвать функцию RegisterRPCs и передать ей соответствующий экземпляр RPCManager. С помощью RegisterRPCs можно также зарегистрировать другие удаленные вызовы процедур. Здесь предполагается, что функция PlaySound реализована где-то в другом месте.

Наконец, чтобы удаленно вызвать процедуру, в вызывающем коде должна иметься функция, которая запишет соответствующий заголовок ObjectReplicationHeader с параметрами в исходящий пакет. В зависимости от реализации она может создавать пакет и отправлять его с помощью игрового кода или сетевого модуля проверять, не существует ли уже готового пакета, ожидающего отправки удаленному узлу. В листинге 5.19 приводится пример функции, записывающей вызов удаленной процедуры в исходящий пакет.

Листинг 5.19. Запись PlaySoundRPC в пакет, ожидающий отправки

```
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);
}
```

Создание функций-оберток, осуществляющих сериализацию и десериализацию вызовов удаленных процедур вручную, регистрация их в `RPCManager` и поддержание соответствия между их параметрами и функциями, выполняющими фактические операции, может потребовать немало усилий. По этой причине во многих движках, поддерживающих RPC, используются специализированные инструменты сборки для автоматического создания функций-оберток и их регистрации в модуле RPC.

ПРИМЕЧАНИЕ Иногда узлу может понадобиться вызвать не простую функцию, а метод определенного объекта. Несмотря на внешнее сходство с удаленным вызовом процедур, для этой цели разработан другой прием, известный как *вызов удаленных методов* (Remote Method Invocation, RMI). Игра, поддерживающая RMI, могла бы использовать сетевой идентификатор в `ObjectReplicationHeader` для идентификации целевого объекта. Идентификатор с нулевым значением мог бы соответствовать простой функции RPC, а с ненулевым — методу определенного игрового объекта. Как вариант для экономии полосы пропускания за счет увеличения объема программного кода можно определить новое действие репликации — `RA_RMI`, подразумевающее наличие поля сетевого идентификатора, а реализация действия `RA_RPC` могла бы продолжать игнорировать его.

Нестандартные решения

Неважно, сколько универсальных инструментов репликации объектов или средств удаленного вызова процедур включают движки, — некоторые игры все еще требуют нестандартных решений репликации и обмена сообщениями. Возможно, в имеющейся реализации отсутствуют необходимые возможности или для передачи быстро изменяющихся значений имеющаяся обобщенная инфраструктура репликации объектов оказывается слишком громоздкой и неэффективно расходует пропускную способность. В таких ситуациях всегда можно добавить собственные действия репликации, расширив перечисление `ReplicationAction` и дополнив инструкцию `switch` в `ProcessReplicationFunction` новыми вариантами. В отдельных случаях при сериализации заголовка `ReplicationHeader` для некоторых объектов можно включать или опускать соответствующий сетевой идентификатор или идентификатор класса.

Если требуемые изменения находятся полностью за пределами `ReplicationManager`, попробуйте расширить перечисление `PacketType` и определить совершенно новые типы пакетов и диспетчеры для их обработки. Следуя шаблону проектирования на основе реестра, использованному для реализации `ObjectCreationRegistry` и `RPCManager`, легко можно внедрить высокоуровневый код для обработки таких нестандартных пакетов, не загромождая низкоуровневую сетевую систему.

В заключение

Под репликацией объектов подразумевается не только передача сериализованных данных между узлами. Прежде всего, протокол прикладного уровня должен определять все возможные типы пакетов, а сетевой модуль должен соответствующим образом маркировать пакеты, содержащие данные из объектов. Каждый объект должен иметь уникальный идентификатор, чтобы получатель смог сохранить принятые данные в соответствующем объекте. Наконец, каждый класс объектов также должен иметь уникальный идентификатор, чтобы получатель смог создать объект требуемого класса, если он еще не был создан до этого. Сетевой код не должен зависеть от классов игровых объектов, поэтому используйте в нем карту какого-либо вида для регистрации реплицируемых классов и функций создания экземпляров.

Небольшие игры могут создавать общие миры между узлами, пересылая все объекты в одном пакете. Большие игры, состояние мира в которых не умещается в один пакет, должны использовать протокол, поддерживающий передачу различий в состоянии мира. Каждое различие может включать действия репликации по созданию, изменению и удалению объекта. Для эффективности изменения объекта можно пересылать только часть свойств объекта. Выбор подмножества зависит от топологии сети и надежности протокола прикладного уровня.

Иногда играм требуется передавать между узлами не только данные о состоянии объектов. Часто возникает необходимость удаленного вызова процедуры. Простой способ реализовать возможность удаленного вызова — ввести еще одно действие репликации и оформлять данные, необходимые для вызова процедуры, в виде пакетов репликации. Модуль RPC может осуществлять регистрацию процедур, осуществляющих обертывание удаленных вызовов, их интерпретацию и вызов требуемых функций, а диспетчер репликации может передавать все входящие запросы RPC этому модулю.

Репликация объектов — это один из ключевых инструментов низкоуровневого программирования многопользовательских игр и критически важный ингредиент поддержки некоторых высокоуровневых топологий сетей, описываемых в главе 6.

Вопросы для повторения

1. Какие три ключевых значения должны присутствовать в пакете репликации объекта, кроме самого содержимого этого объекта?
2. Почему нежелательна зависимость сетевого кода от реализации игры?
3. Объясните, как реализовать поддержку создания реплицируемых объектов на стороне получателя без создания сетевого кода, зависимого от игровых классов.
4. Реализуйте простую игру с пятью движущимися объектами в ней. Осуществите репликацию этих объектов на удаленный узел, посылая пакеты с состоянием мира 15 раз в секунду.
5. В продолжение вопроса 4: какие проблемы возникнут с увеличением числа игровых объектов? Как решаются эти проблемы?

6. Реализуйте систему, поддерживающую отправку обновлений только нескольких свойств объектов.
7. Что такое RPC? Что такое RMI? Чем они отличаются?
8. Используя инфраструктуру, разработанную в этой главе, реализуйте удаленный вызов процедуры `SetPlayerName(const string& inName)`, сообщающий другим узлам имя локального игрока.
9. Реализуйте нестандартный тип пакетов для передачи информации о клавишах на клавиатуре, удерживаемых игроком в нажатом состоянии, эффективно используя полосу пропускания. Объясните, как интегрировать эту реализацию с инфраструктурой репликации, разработанной в этой главе.

Для дополнительного чтения

Carmack, J. (1996, август). «Here Is the New Plan». Доступно по адресу: *http://fabiensanglard.net/quakeSource/johnc-log.aug.htm*. Проверено 28 января 2016.

Srinivasan, R. (1995, август). «RPC: Remote Procedure Call Protocol Specification Version 2». Доступно по адресу: *http://tools.ietf.org/html/rfc1831*. Проверено 28 января 2016.

Van Waveren, J. M. P. (2006, март). «The DOOM III Network Architecture». Доступно по адресу: *http://mrelusive.com/publications/papers/The-DOOM-III-Network-Architecture.pdf*. Проверено 28 января 2016.

Winer, Dave (1999, июнь). «XML-RPC Specification». Доступно по адресу: *http://xmlrpc.scripting.com/spec.html*.[1] Проверено 28 января 2016.

[1] Перевод на русский язык: *http://allxml.h1.ru/articles/XML-RPC%20Specification.htm. — Примеч. пер.*

6 Топологии сетей и примеры игр

В первой части этой главы рассматриваются две основные конфигурации организации взаимодействия множества компьютеров: «клиент-сервер» и «точка-точка». Во второй части главы мы приступим к созданию двух примеров игр, объединяющих все рассматривавшиеся до сих пор темы.

Топологии сетей

По большому счету, в главах с первой по пятую в основном рассматривались проблемы, касающиеся взаимодействий между двумя компьютерами через Интернет и обмена информацией способом, характерным для сетевых игр. Хотя игры, рассчитанные только на двух игроков, и существуют, все же подавляющее число популярных игр поддерживают большее число игроков. Однако даже в случае с двумя игроками возникает множество вопросов. Как игроки будут посылать друг другу обновления в игровом мире? Будет ли выполняться репликация объектов, как описано в главе 5, или реплицироваться будут только действия игроков? Что случится, если компьютеры обнаружат неразрешимые противоречия в состояниях игры? Все это — важные вопросы, ответы на которые должны быть сформулированы в любой сетевой многопользовательской игре.

Топология сети (network topology) — это конфигурация графа, вершинам которого соответствуют конечные узлы сети, а ребрам — связи между ними. В контексте игры топология определяет способ организации участвующих в игре компьютеров, гарантирующий, что у всех игроков будет информация об актуальном состоянии игры. Так же как выбор сетевых протоколов, выбор топологии сопряжен с поиском компромисса между достоинствами и недостатками. В этом разделе исследуются две основные топологии, используемые в играх: *«клиент-сервер»* и *«точка-точка»*, а также некоторые их разновидности.

«Клиент-сервер»

В топологии «*клиент-сервер*» один экземпляр игры действует как выделенный сервер, а все остальные экземпляры — как клиенты. Каждый клиент взаимодействует только с сервером, тогда как сервер взаимодействует со всеми клиентами. Эта топология изображена на рис. 6.1.

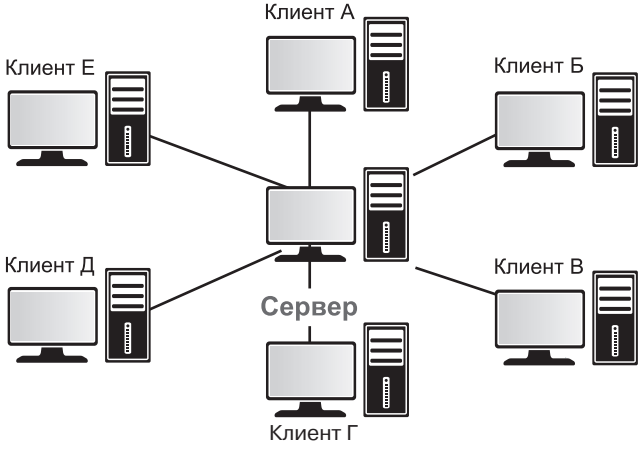


Рис. 6.1. Топология «клиент-сервер»

В топологии «клиент-сервер» с n клиентами имеется всего $O(n)$ соединений. Однако число соединений между сервером и клиентами распределяется несимметрично: сервер будет иметь $O(n)$ соединений (по одному с каждым клиентом), тогда как каждый клиент будет иметь только одно соединение — соединение с сервером. В отношении полосы пропускания: если имеется n клиентов, каждый из которых посылает b байтов данных в секунду, сервер должен обладать пропускной способностью, позволяющей обрабатывать $b \times n$ входящих байтов в секунду. Аналогично, если сервер должен посылать каждому клиенту c байтов данных в секунду, он должен иметь возможность обрабатывать $c \times n$ исходящих байтов в секунду. Но каждый клиент должен иметь входящую пропускную способность только c байтов в секунду и исходящую пропускную способность b байтов в секунду. Это означает, что с увеличением числа клиентов линейно возрастают требования к пропускной способности сервера. В теории требования к пропускной способности клиентов не зависят от их количества, но на практике увеличение числа поддерживаемых клиентов ведет к увеличению количества объектов, подлежащих репликации, что, в свою очередь, влечет некоторое увеличение трафика каждого клиента.

Хотя это не единственный подход к реализации топологии «клиент-сервер», тем не менее в большинстве игр, построенных на этой топологии, используется *уполномоченный* (authoritative) сервер. То есть считается, что любые игровые действия, выполняемые сервером, являются правильными. Если клиент обнаружит у себя расхождения с сервером, он должен обновить состояние игры, опираясь на информацию сервера. Например, в игре «Robo Cat Action», обсуждаемой далее в этой

главе, каждый кот, управляемый игроком, может бросить клубок пряжи. Но в модели с уполномоченным сервером клиенту не позволено определять, поразил ли он клубком другого игрока. Вместо этого клиент должен информировать сервер, что хочет выполнить бросок. А уже сервер решает, может ли клиент метнуть клубок, и если может, — поразил ли он другого игрока.

Использование схемы с уполномоченным сервером означает наличие некоторой задержки в ответной реакции на действия клиента. Тема задержек подробно обсуждается в главе 7 «Задержки, флуктуации и надежность», однако она заслуживает того, чтобы кратко рассмотреть ее здесь. Когда происходит бросок клубка, только сервер может принять решение, что должно произойти. Разумеется, какое-то время потребуется, чтобы отправить серверу запрос на выполнение броска, сервер, в свою очередь, потратит какое-то время на принятие решения и рассылку ответа всем клиентам. Одним из факторов, вносящих вклад в эту задержку, является *время передачи/подтверждения* (Round Trip Time, RTT) — время (обычно измеряется в миллисекундах), необходимое для передачи пакетов запроса и ответа для определенного компьютера в сети. В идеальном случае величина RTT составляет 100 мс или меньше, однако даже в современном Интернете существует множество факторов, препятствующих уменьшению величины RTT до такого низкого значения.

Представьте игру с сервером и двумя клиентами — А и Б. Так как сервер посылает все игровые данные каждому клиенту, это означает, что если клиент А бросит клубок, пакет с запросом на бросок должен сначала дойти до сервера. Затем сервер обработает факт броска и отправит результат обратно клиентам А и Б. В этом сценарии худшая сетевая задержка, испытываемая клиентом Б, могла бы составить 1/2 величины RTT для клиента А плюс время, необходимое серверу на обработку, плюс 1/2 величины RTT для клиента Б. В быстрой сети это может не ощущаться, но в реальности большинство игр вынуждены использовать разнообразные приемы сокрытия задержек. Об этом подробно рассказывается в главе 8 «Улучшенная обработка задержек».

Существует дополнительная классификация типов серверов. Некоторые серверы являются *выделенными* (dedicated) в том смысле, что их функция заключается только в обслуживании состояния игры и взаимодействии со всеми клиентами. Процесс выделенного сервера полностью отделен от любых клиентских процессов, выполняющих игру. Обычно выделенный сервер не имеет монитора и не отображает графику. Серверы такого типа часто используются в высокобюджетных играх, таких как «Battlefield», что позволяет разработчикам запускать множество процессов выделенных серверов на одной мощной машине.

Альтернативой выделенному серверу является *прослушивающий,* или *локальный, сервер* (listen server). В этой схеме сервер сам является активным участником игры. Одно из преимуществ схемы с локальным сервером — возможность уменьшить стоимость развертывания, потому что нет необходимости арендовать сервер в вычислительном центре: вместо этого один из игроков может использовать свой компьютер одновременно в роли сервера и клиента. Однако схема с локальным сервером имеет большой недостаток — компьютер с локальным сервером должен быть достаточно мощным и иметь высокоскоростное подключение к Интернету, чтобы справляться с повышенной нагрузкой. Подход с использованием локаль-

ного сервера иногда ошибочно называют соединением «точка-точка», однако его правильнее было бы называть схемой с *ведущим узлом* (peer hosted). В игре все еще присутствует сервер, но он, по стечению обстоятельств, выполняется на компьютере игрока, участвующего в игре.

Важно отметить, что если локальный сервер наделить функциями уполномоченного сервера, он будет иметь полную картину состояния игры. То есть игрок, на чьем компьютере выполняется локальный сервер, теоретическим получает возможность использовать это обстоятельство для обмана других. Кроме того, в модели «клиент-сервер» обычно только сервер знает сетевые адреса всех активных клиентов. Это может стать большой проблемой в случае отключения сервера из-за перебоев связи или из-за того, что рассердившийся игрок решит покинуть игру. Некоторые локальные серверы реализуют понятие *передачи управления*, когда, в случае утраты соединения с локальным сервером, сервером автоматически становится один из клиентов. Однако для этого требуется пересылать некоторую дополнительную информацию между клиентами. Это означает, что для передачи управления требуется реализация гибридной модели, в которой одновременно используются две топологии: «клиент-сервер» и «точка-точка».

«Точка-точка»

В топологии «*точка-точка*» (peer-to-peer) каждый участник связан со всеми остальными участниками. Как показано на рис. 6.2, это приводит к увеличению трафика между клиентами. Число соединений есть квадратичная функция от числа клиентов, иначе говоря, для сети из n узлов, где каждый узел должен иметь $O(n)$ соединений, общее число соединений составит $O(n^2)$. Это также означает ужесточение

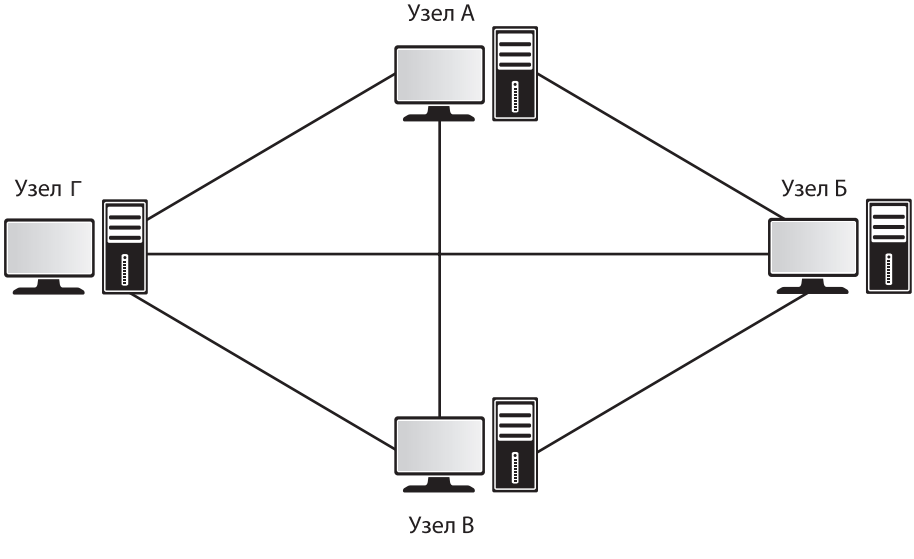


Рис. 6.2. Топология точка-точка

требований к пропускной способности каждого узла с увеличением числа узлов, участвующих в игре. Однако, в отличие от топологии «клиент-сервер», требования к пропускной способности являются симметричными, то есть всем узлам предъявляются одинаковые требования к пропускной способности.

В игре, основанной на топологии «точка-точка», понятие уполномоченности намного более размыто. Одно из возможных решений — распределить ответственность между узлами за разные части игры, но на практике такие системы сложны в реализации. Чаще в играх с топологией «точка-точка» используется прием рассылки всех действий каждому узлу, и каждый узел выполняет эти действия. Данную модель иногда называют *моделью разделяемого ввода* (input sharing model).

Одним из аспектов топологии «точка-точка», который делает модель разделяемого ввода более жизнеспособной, является тот факт, что задержки играют менее важную роль. В противоположность модели «клиент-сервер» с промежуточным звеном между клиентами в модели «точка-точка» все узлы имеют непосредственные соединения друг с другом. Это означает, что в худшем случае задержка между узлами составит 1/2 RTT. Однако в игре с топологией «точка-точка» имеется еще один источник задержек, связанный со сложностью обеспечения синхронизации всех узлов друг с другом.

Вспомним обсуждение модели детерминированного соответствия в главе 1. В игре «Age of Empires» все игровые действия разделены на «ходы», выполняющиеся в 200-миллисекундные интервалы. В течение 200 мс все команды помещаются в очередь, а когда время истекает, команды рассылаются всем узлам. Кроме того, выполняется задержка на один шаг, то есть когда каждый узел отображает результаты хода с порядковым номером 1, он накапливает команды, которые будут выполнены во время отображения хода с порядковым номером 3. Несмотря на кажущуюся простоту такой пошаговой синхронизации, фактическая реализация может оказаться весьма сложной. Пример игры «Robo Cat RTS», обсуждаемый далее в этой главе, реализует самую простую версию такой модели.

Не менее важно гарантировать согласованность состояния игры между всеми узлами. Это означает, что реализация игры должна быть полностью детерминированной. То есть заданный набор входных команд всегда должен приводить к одним и тем же результатам. В число важных аспектов, связанных с такими гарантиями, входят использование контрольных сумм для проверки согласованности состояния игры между узлами и синхронизация генераторов случайных чисел между всеми узлами, — обе темы подробно будут рассматриваться далее в этой главе.

Еще одна проблема модели «точка-точка» связана с подключением новых игроков. Так как каждый узел должен знать адреса всех остальных узлов, теоретически новый игрок мог бы подключиться к любому узлу. Однако координационные службы со списками доступных игр принимают только один адрес — в данном случае один узел выбирается как «главный узел» — единственный принимающий соединения от новых игроков.

Наконец, проблема отключения сервера, являющаяся насущной для модели «клиент-сервер», фактически отсутствует в модели «точка-точка». Обычно в случае потери соединения с узлом игра может приостановиться на несколько секунд, после

чего узел будет исключен из игры. После отключения одного узла все остальные могут продолжать действовать в соответствии с игровой моделью.

Реализация модели «клиент-сервер»

Объединив все идеи, которые мы рассмотрели в этой книге к данному моменту, можно приступать к созданию начальной версии сетевой игры. В этом разделе обсуждается одна из таких игр. Игра «Robo Cat Action» — это игра с «видом сверху», в которой коты стараются собрать как можно больше мышей и при этом могут кидать друг в друга клубки пряжи. На рис. 6.3 показана сцена из этой игры. Исходный код первой версии игры можно найти в каталоге `Chapter6/RoboCatAction` репозитория.

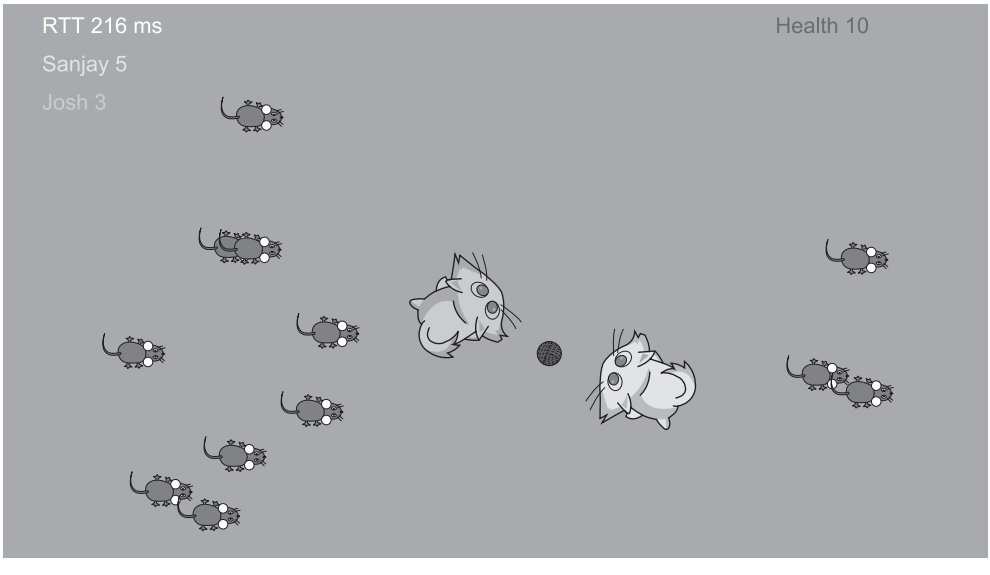


Рис. 6.3. Начальная версия игры «Robo Cat Action»

Управлять игрой «Robo Cat Action» несложно. Клавиши D и A управляют поворотом кота по часовой и против часовой стрелки соответственно. Клавиши W и S — движением вперед и назад. Клавиша K выполняет бросок клубка, который может попасть в другого кота. Кот также может собирать мышей, наезжая на них.

Эта первая версия игры опирается на существенные допущения: задержки в сети отсутствуют или незначительны и все пакеты достигают своих адресатов. Совершенно ясно, что эти допущения далеки от реальности для любой сетевой игры, поэтому в последующих главах, и особенно в главе 7 «Задержки, флуктуации и надежность», мы обсудим варианты ликвидации этих допущений. А пока займемся обсуждением основ разработки игры в модели «клиент-сервер», не отвлекаясь на дополнительные сложности, связанные с задержками и потерями пакетов.

Разделение серверного и клиентского кода

Одной из ключевых характеристик модели «клиент-сервер» с уполномоченным сервером является отличие кода, выполняемого сервером и каждым клиентом. Возьмем для примера главный персонаж, робокота (Robo Cat). Одно из свойств кота — переменная `mHealth` — хранит уровень его здоровья. Сервер должен знать о состоянии здоровья кота, потому что когда его уровень достигает нуля, кот должен перейти в состояние перерождения (в конце концов, кошки имеют девять жизней). Для клиента также важна информация о состоянии здоровья кота, поэтому его уровень должен отображаться в правом верхнем углу экрана. Даже при условии, что истинным считается значение серверного экземпляра переменной `mHealth`, клиент должен кэшировать эту переменную у себя, чтобы отображать его в пользовательском интерфейсе.

То же можно сказать о функциях. Класс `RoboCat` может иметь несколько функций-членов, необходимых только серверу, только клиенту и обоим. Для решения этой проблемы в «Robo Cat Action» используются механизмы наследования и виртуальных функций. То есть существует базовый класс `RoboCat` и два дочерних класса — `RoboCatServer` и `RoboCatClient`, — каждый из которых переопределяет имеющиеся и реализует новые функции-члены. С точки зрения производительности использование виртуальных функций не позволяет достичь наивысшей возможной скорости, но с точки зрения простоты организации кода иерархическая модель выглядит проще.

Идею деления кода на разные классы можно развить еще дальше: исследовав исходный код, легко заметить, что он делится на три отдельные цели. Первая цель — библиотека RoboCat, содержащая общий код, используемый сервером и клиентом. Она включает такие классы, как `UDPSocket`, реализованный в главе 3, и класс `OutputMemoryBitSteam`, реализованный в главе 4. Другие две цели — исполняемые файлы `RoboCatServer` для сервера и `RoboCatClient` для клиента.

ПРИМЕЧАНИЕ Из-за наличия двух разных исполняемых файлов для сервера и клиента, чтобы опробовать игру «Robo Cat Action», нужно запустить их оба. Сервер принимает один параметр командной строки, определяющий номер порта для приема соединений. Например:

```
RoboCatServer 45000
```

Данная команда указывает, что сервер должен принимать соединения от клиентов на порту с номером 45000.

Клиент принимает два параметра командной строки: полный адрес сервера (включая порт) и имя клиента. Например:

```
RoboCatClient 127.0.0.1:45000 John
```

Данная команда указывает, что клиент должен подключиться к серверу, принимающему соединение на порту 45000, по адресу `localhost`, с именем игрока «John». Естественно, к одному серверу могут подключиться несколько клиентов, а так как игра потребляет не очень много ресурсов, для тестирования на одном компьютере можно запустить несколько экземпляров игры.

Иерархия класса `RoboCat` в данном примере включает три разных класса в трех разных целях: базовый класс `RoboCat` находится в разделяемой библиотеке, а классы `RoboCatServer` и `RoboCatClient`, что неудивительно, — в соответствующих исполняемых файлах. Такой подход дает ясное разделение программного кода

и помогает различать серверный и клиентский код. Чтобы получить более полное представление о данном подходе, взгляните на иерархию класса `GameObject` в игре «Robo Cat Action» (рис. 6.4).

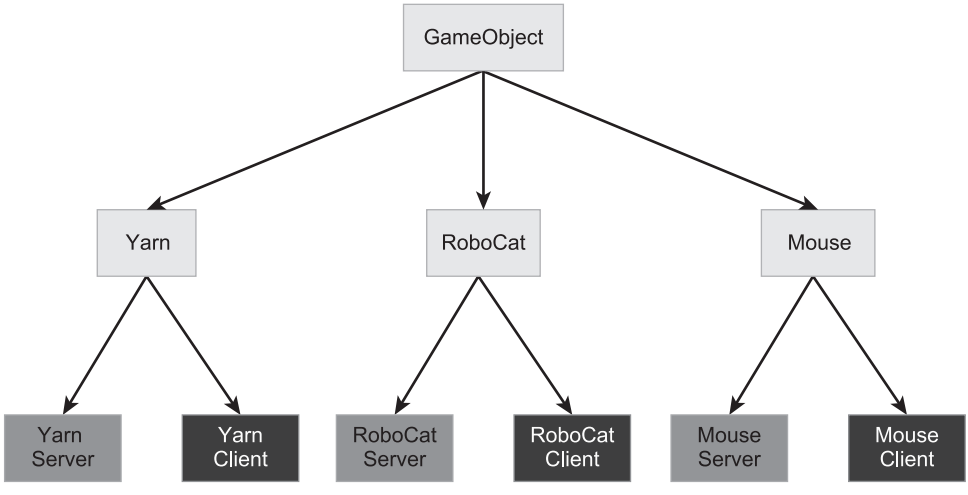


Рис. 6.4. Иерархия класса GameObject в игре «Robo Cat Action» (элементы светло-серого цвета принадлежат разделяемой библиотеке, черного — выполняемому файлу клиента и темно-серого — выполняемому файлу сервера)

Диспетчер сети и приветствие новых клиентов

Класс `NetworkManager` и наследующие его классы `NetworkManagerClient` и `NetworkManagerServer` реализуют основные операции, связанные со взаимодействием по сети. Например, весь код, принимающий пакеты и помещающий их в очередь для последующей обработки, находится в базовом классе `NetworkManager`. Реализация обработки пакетов очень похожа на то, что разбиралось в главе 3 «Сокеты Беркли», поэтому она не будет описываться повторно.

Класс `NetworkManager` отвечает также за присоединение новых клиентов к игре. Игра «Robo Cat Action» допускает свободный вход в игру и выход из нее, поэтому новый клиент может в любой момент присоединиться к ней. Как можно догадаться, приветствие новых клиентов по-разному реализовано на сервере и клиенте, поэтому данная функциональность разделена между `NetworkManagerClient` и `NetworkManagerServer`.

Прежде чем погрузиться в программный код, стоит рассмотреть в общих чертах сам процесс подключения. Процедуру подключения можно условно разделить на четыре шага:

1. Чтобы подключиться к игре, клиент отправляет серверу пакет «приветствия». Этот пакет содержит только литерал `"HELO"` (чтобы идентифицировать тип пакета) и сериализованную строку с именем игрока. Клиент продолжает посылать такие пакеты до тех пор, пока не получит подтверждение от сервера.

2. Получив пакет приветствия, сервер присваивает новому игроку идентификационный номер и выполняет некоторые операции по инициализации, такие как связывание адреса отправителя `SocketAddress` с идентификационным номером игрока. Затем сервер посылает клиенту свой пакет приветствия. Этот пакет содержит литерал `"WLCM"` и идентификационный номер, присвоенный игроку.
3. Получив пакет приветствия от сервера, клиент сохраняет свой идентификационный номер и начинает отправлять и принимать информацию об объектах.
4. В некоторый момент времени сервер отправит информацию обо всех объектах, порожденных для нового клиента, как самому новому клиенту, так и всем остальным клиентам.

В данном конкретном примере потеря пакетов просто компенсируется их избыточностью. Если клиент не получит пакет `"WLCM"`, он продолжит посылать серверу пакеты `"HELO"`. Если сервер получит пакет `"HELO"` от клиента с уже зарегистрированным адресом `SocketAddress`, он просто повторит отправку пакета `"WLCM"`.

Заглянув в код, можно увидеть два литерала, используемые для идентификации пакетов, оформленные как константы в базовом классе `NetworkManager`:

```
static const uint32_t kHelloCC = 'HELO';
static const uint32_t kWelcomeCC = 'WLCM';
```

На стороне клиента класс `NetworkManagerClient` определяет перечисление с возможными состояниями клиента:

```
enum NetworkClientState
{
  NCS_Uninitialized,
  NCS_SayingHello,
  NCS_Welcomed
};
```

После инициализации экземпляр `NetworkManagerClient` присваивает своей переменной-члену `mState` значение `NCS_SayingHello`. Пока клиент находится в этом состоянии, он будет продолжать посылать пакеты `"HELO"` серверу. Получив пакет `"WLCM"`, клиент должен начать передавать серверу обновления. В этом случае обновления являются входящими пакетами, о которых рассказывается далее.

Кроме того, клиент распознает пакеты по присвоенным им четырехсимвольным литералам. В частности, в игре «Robo Cat Action» существует всего два типа пакетов, которые может принимать клиент: пакет приветствия `"WLCM"` и пакет с информацией о состоянии. Реализация отправки и приема пакетов напоминает конечный автомат и представлена в листинге 6.1.

Листинг 6.1. Отправка и прием пакетов на стороне клиента

```
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;
  }
}
```

Единственная проблема, связанная с отправкой пакетов `"HELO"`, — клиент не должен посылать их слишком часто. Он делает это, проверяя время, прошедшее с момента отправки последнего такого пакета. Сам пакет устроен очень просто, так как клиенту нужно лишь записать в него литерал `"HELO"` и свое имя. Аналогично, пакет `"WLCM"` содержит только идентификационный номер клиента, который клиент должен сохранить. Реализация обработки этих пакетов показана в листинге 6.2. Обратите внимание, что `HandleWelcomePacket` проверяет состояние клиента после получения пакета `"WLCM"`. Эта проверка помогает избежать ошибок, если пакет `"WLCM"` будет принят клиентом повторно. Похожая проверка выполняется в `HandleStatePacket`.

Листинг 6.2. Передача пакетов "HELO" и прием пакетов "WLCM" на стороне клиента

```
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)
  {
    // если принят идентификационный номер, сервер ответил на приветствие!
    int playerId;
    inInputStream.Read(playerId);
    mPlayerId = playerId;
    mState = NCS_Welcomed;
    LOG("'%s' was welcomed on client as player %d",
      mName.c_str(), mPlayerId);
  }
}
```

На стороне сервера процедура выглядит немного сложнее. Для начала, на сервере имеется ассоциативный массив адресов с именем `mAddressToClientMap`, в котором регистрируются все известные клиенты. Ключом в этом ассоциативном массиве служит `SocketAddress`, а значением — указатель на экземпляр `ClientProxy`. Прокси-объекты, представляющие клиентов, мы обсудим далее в этой главе, а пока просто считайте их объектами, при помощи которых сервер может следить за состоянием клиентов. Имейте в виду: из-за того, что здесь непосредственно используются адреса сокетов, могут возникнуть проблемы с передачей пакетов через NAT, как уже обсуждалось в главе 2. В игре «Robo Cat» мы не будем заниматься решением этой проблемы.

Когда сервер впервые получит пакет, он выполнит поиск в карте адресов, чтобы определить, зарегистрирован ли уже отправитель. Если отправитель не зарегистрирован, сервер проверяет тип пакета. Если получен пакет, отличный от пакета `"HELO"`, он просто игнорируется.

В противном случае сервер создает прокси-объект для нового клиента и посылает пакет `"WLCM"`. Реализация этого процесса представлена в листинге 6.3, где, впрочем, отсутствует код отправки пакета `"WLCM"`, так как он так же прост, как код отправки пакета `"HELO"`.

Листинг 6.3. Обработка новых клиентов на сервере

```
void NetworkManagerServer::ProcessPacket
(
  InputMemoryBitStream& inInputStream,
  const SocketAddress& inFromAddress
)
{
  // клиент зарегистрирован?
```

```
  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);

    // создать прокси-объект
    // ...
    // и поприветствовать его клиента...
    SendWelcomePacket(newClientProxy);

    // инициализировать диспетчера репликации для данного клиента
    // ...
  }
  else
  {
    LOG("Bad incoming packet from unknown client at socket %s",
      inFromAddress.ToString().c_str());
}
```

Разделяемый ввод и прокси-объекты

Реализация репликации игровых объектов в «Robo Cat Action» очень напоминает подход, обсуждавшийся в главе 5 «Репликация объектов». В игре используется три команды репликации: создать, изменить и удалить. Кроме того, для уменьшения объема информации, передаваемой в пакетах, используется прием частичной репликации объектов. Поскольку в игре используется модель с уполномоченным сервером, репликация объектов осуществляется только в одном направлении — от сервера к клиенту, то есть за отправку информации об изменившихся объектах отвечает сервер (в виде пакетов с литералом `'STAT'`), а клиент, в свою очередь, отвечает за обработку команд репликации. Сервер должен выполнить некоторую работу, чтобы отправить соответствующие команды каждому клиенту, о чем мы поговорим далее в этом разделе.

А пока рассмотрим те пакеты, которые клиент должен отправлять серверу. Так как сервер является уполномоченным, клиент не должен посылать ему никаких команд, связанных с репликацией объектов. Однако чтобы точно имитировать действия каждого клиента, он должен знать, что именно каждый клиент пытается сделать. Это подводит нас к идее *пакета ввода*. В каждом кадре клиент обрабатывает события ввода. Если какое-либо из этих событий сопровождается информацией, которая должна обрабатываться на сервере, например перемещение кота или бросок клубка, клиент посылает такие события серверу. Сервер принимает пакет ввода и сохраняет его в *прокси-объекте*, который используется для слежения за состоянием конкретного клиента. Наконец, когда сервер приступит к обработке игровой модели, он учтет все события ввода, хранящиеся в прокси-объекте клиента.

Класс `InputState` хранит мгновенный снимок состояния ввода клиента в определенном кадре. В каждом кадре объект `InputManager` обновляет `InputState`, опираясь на события ввода клиента. Информация, хранимая в `InputState`, будет различаться для разных игр. В данном конкретном случае хранится только смещение в каждом из четырех направлений, описывающее перемещение кота, и признак нажатия клавиши, отвечающей за бросок клубка пряжи. В результате получается класс с очень небольшим числом членов, как показано в листинге 6.4.

Листинг 6.4. Объявление класса InputState

```
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;
};
```

`GetDesiredHorizontalDelta` и `GetDesiredVerticalDelta` — это вспомогательные функции, определяющие общее смещение по каждой оси. Например, если игрок

удерживает нажатыми обе клавиши, A и D, общее смещение по горизонтали должно быть равно нулю. Реализация функций `Read` и `Write` не включена в листинг 6.4 — эти функции просто читают и записывают значения переменных-членов в указанный битовый поток.

Обратите внимание, что `InputState` обновляется диспетчером `InputManager` в каждом кадре. Для большинства игр не имеет смысла так часто передавать на сервер `InputState`. В идеале следовало бы объединить в одном пакете объекты `InputState` из нескольких кадров, идущих подряд, и передать их все сразу. Но чтобы не усложнять реализацию, в «Robo Cat Action» такое объединение не выполняется. Вместо этого текущее состояние `InputState` извлекается каждые *x* секунд и сохраняется в виде объекта `Move`.

Класс `Move` фактически является оберткой вокруг `InputState` с двумя дополнительными вещественными значениями: одно хранит отметку времени создания объекта `Move`, а другое — разницу во времени между текущим и предыдущим шагом. Определение этого класса приводится в листинге 6.5.

Листинг 6.5. Класс Move

```
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;
};
```

Функции `Read` и `Write` читают и записывают состояние ввода и отметку времени из/в указанный поток.

ПРИМЕЧАНИЕ Несмотря на то что класс `Move` является всего лишь тонкой оберткой вокруг `InputState` с двумя дополнительными переменными, хранящими значения времени, он был создан с целью подчеркнуть ориентированность кода на покадровую обработку. Диспетчер `InputManager` опрашивает состояние клавиатуры в каждом кадре и сохраняет данные в `InputState`. Только когда клиенту действительно понадобится создать экземпляр `Move`, отметка времени начинает приобретать смысл.

Далее последовательность шагов сохраняется в списке `MoveList`. Этот класс хранит, как можно догадаться, список шагов, а также отметку времени последнего шага в списке. Когда клиент обнаруживает, что должен сохранить очередной шаг, он добавляет его в список. Затем, когда подойдет срок, `NetworkManagerClient` записывает последовательность шагов в пакет. Обратите внимание, что при записи последовательности шагов выполняется оптимизация числа записываемых битов счетчика, основанная на предположении, что в один пакет никогда не будет записываться больше трех шагов. Это предположение опирается на константные множители, определяющие частоту шагов и передачи пакетов ввода. Клиентский код, имеющий отношение к списку шагов, показан в листинге 6.6.

Листинг 6.6. Клиентский код обслуживания списка шагов

```
const Move& MoveList::AddMove(const InputState& inInputState,
               float inTimestamp)
{
  // первый шаг имеет разность во времени 0
  float deltaTime = mLastMoveTimestamp >= 0.f ?
           inTimestamp – mLastMoveTimestamp: 0.f;

  mMoves.emplace_back(inInputState, inTimestamp, deltaTime);
  mLastMoveTimestamp = inTimestamp;
  return mMoves.back();
}

void NetworkManagerClient::SendInputPacket()
{
  // посылать, только если есть пакет ввода для отправки!
  MoveList& moveList = InputManager::sInstance->GetMoveList();

  if(moveList.HasMoves())
  {
    OutputMemoryBitStream inputPacket;
    inputPacket.Write(kInputCC);

    // послать только последние три шага
    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();
  }
}
```

Обратите внимание, что в `SendInputPacket` для доступа к элементам `MoveList` используется оператор индексирования массива. Внутри `MoveList` использует структуру данных `deque`, поэтому данный оператор имеет постоянное время выполнения. В плане избыточности реализация `SendInputPacket` не отличается

высокой надежностью. Клиент посылает шаги только один раз. Поэтому, например, если пакет с командой ввода «бросок» не достигнет сервера, будет считаться, что клиент не выполнял никакого броска. Очевидно, что это нежелательная ситуация в многопользовательской игре.

В главе 7 «Задержки, флуктуации и надежность» вы узнаете, как добавить некоторую избыточность в пакеты ввода. В частности, каждый шаг можно было бы отправлять трижды, чтобы предоставить серверу три возможности обработать шаг. Это несколько усложнит реализацию, потому что серверу придется определять, обрабатывался ли тот или иной шаг.

Как упоминалось выше, для хранения информации о каждом клиенте сервер использует прокси-объекты. Среди прочих, одной из важнейших обязанностей прокси-объекта является хранение отдельного диспетчера репликации для каждого клиента. Это позволяет серверу иметь полное представление о том, какую информацию он посылал или не посылал каждому клиенту. Так как для сервера важно, чтобы каждый пакет репликации не посылался в каждом кадре каждому клиенту, для каждого клиента необходимо иметь отдельный диспетчер репликации. Это станет особенно важно после добавления избыточности, потому что поможет серверу узнать, какие именно переменные требуется переслать конкретному клиенту.

Каждый прокси-объект хранит также адрес сокета, имя и идентификационный номер игрока. Кроме того, именно в прокси-объекте сохраняется информация о шагах каждого клиента. Когда сервер принимает пакет ввода, он добавляет все шаги, выполненные клиентом, в экземпляр `ClientProxy`, представляющий этого клиента. Часть объявления класса `ClientProxy` находится в листинге 6.7.

Листинг 6.7. Неполное объявление класса ClientProxy

```
class ClientProxy
{
public:
  ClientProxy(const SocketAddress& inSocketAddress, const string& inName,
        int inPlayerId);
  // Функции опущены
  // ...
  MoveList& GetUnprocessedMoveList() {return mUnprocessedMoveList;}
private:
  ReplicationManagerServer mReplicationManagerServer;
  // Переменные опущены
  // ...
  MoveList mUnprocessedMoveList;
  bool mIsLastMoveTimestampDirty;
};
```

Наконец, класс `RoboCatServer` выполняет обработку шагов в своей функции `Update`, представленной в листинге 6.8. Важно отметить, что разность во времени, передаваемая в каждый вызов `ProcessInput` и `SimulateMovement`, — это разность во времени между шагами, а не разность во времени между кадрами на сервере. Именно так сервер старается гарантировать максимальную близость игровых событий к действиям клиента, даже если в одном пакете принимаются несколько шагов сразу.

Такой подход позволяет также иметь разную частоту кадров на стороне сервера и клиента. Это может несколько усложнить моделирование физики объектов, которое должно выполняться на протяжении нескольких шагов. Если это важно для вашей игры, организуйте хранение частоты кадров моделирования физических процессов отдельно от других частот.

Листинг 6.8. Обновление в классе RoboCatServer

```
void RoboCatServer::Update()
{
  RoboCat::Update();
  // Часть кода опущена
  // ...

  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();

  // Часть кода опущена
  // ...
}
```

Реализация модели «точка-точка»

Игра «Robo Cat RTS» — это стратегия в реальном масштабе времени, поддерживающая до четырех игроков. Каждому игроку даются три кота. Управление осуществляется так: сначала нужно выбрать кота щелчком мыши, а затем щелкнуть правой кнопкой на цели. Если целью является просто некоторая точка на игровом поле, кот переместится в эту точку. Если целью является кот противника, ваш кот сначала приблизится к цели, а затем атакует ее. Так же как в сюжетной игре «Robo Cat Action», коты атакуют друг друга, бросая клубки пряжи. На рис. 6.5 показана игра «Robo Cat RTS» в действии. Реализация начальной версии игры находится в каталоге `Chapter6/RoboCatRTS`.

Хотя в обеих версиях игры используется протокол UDP, сетевая модель в «Robo Cat RTS» существенно отличается от сетевой модели в «Robo Cat Action». Так же как сюжетная игра, начальная версия RTS, обсуждаемая ниже, предполагает, что пакеты не будут теряться. Однако из-за особенностей модели детерминированного соответствия игра будет функционировать с некоторым запаздыванием, при этом, чем больше будут задержки, тем хуже будет качество восприятия игры.

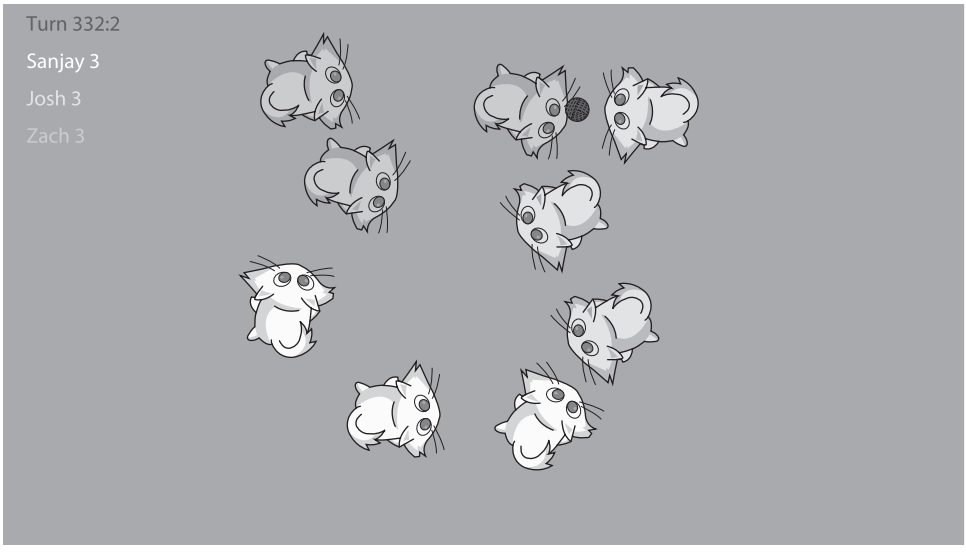


Рис. 6.5. Игра «Robo Cat RTS» в действии

Так как в «Robo Cat RTS» используется модель «точка-точка», нет необходимости делить код на несколько проектов. Все узлы будут выполнять один и тот же код. Это несколько уменьшает число файлов, а также означает, что все игроки будут запускать один и тот же исполняемый файл.

ПРИМЕЧАНИЕ Существует два различных способа запуска игры «Robo Cat RTS», при этом в обоих случаях запускается один и тот же исполняемый файл. Чтобы запустить экземпляр игры в роли ведущего узла, укажите номер порта и имя игрока:

```
RoboCatRTS 45000 John
```

Чтобы запустить обычный экземпляр игры, укажите полный адрес ведущего узла (включая номер порта), а также имя игрока:

```
RoboCatRTS 127.0.0.1:45000 Jane
```

Обратите внимание, что, если указать адрес обычного, рядового узла, игрок все равно будет подключен к игре, но процедура подключения будет выполнена быстрее, если указать адрес ведущего узла.

При этом в игре «Robo Cat RTS» используется идея *ведущего узла* (master peer). Основная роль ведущего узла заключается в том, чтобы сообщить IP-адреса узлов, участвующих в игре. Такая организация особенно хорошо согласуется при использовании координационной службы, поддерживающей список известных доступных игр. Кроме того, ведущий узел — единственный, кому позволено присваивать идентификационные номера новым игрокам. Сделано это в первую очередь для того, чтобы избежать конкуренции, которая может возникнуть, если два новых игрока одновременно попытаются связаться с разными узлами. Кроме этого особого случая, ведущий узел действует точно так же, как все остальные узлы. Так как

каждый узел поддерживает собственное, независимое состояние игры, игра может продолжаться после отключения ведущего узла.

Подключение новых узлов и запуск игры

Процесс подключения к игре в модели «точка-точка» выглядит несколько сложнее, чем в модели «клиент-сервер». Так же как в игре «Robo Cat Action», новый узел сначала посылает пакет `"HELO"` с именем игрока. Но на этот раз в ответ на этот пакет может быть получен один из трех разных пакетов:

1. Пакет **«Добро пожаловать»** (`"WLCM"`) — означает, что пакет `"HELO"` был получен ведущим узлом и новый узел приглашается к участию в игре. Пакет `"WLCM"` содержит идентификационный номер нового игрока, идентификационный номер игрока для ведущего узла и число участников в игре (не включая нового игрока). Кроме того, в пакете передаются имена и IP-адреса всех узлов.
2. Пакет **«В подключении отказано»** (`"NOJN"`) — означает, что игра уже началась или число участников достигло максимума. Получив такой пакет, новый узел завершает игру.
3. Пакет **«Узел не является ведущим»** (`"NOMP"`) — такой пакет возвращается, если пакет приветствия `"HELO"` был отправлен узлу, не являющемуся ведущим. В этом случае в пакете будет возвращен адрес ведущего узла, чтобы новый узел мог послать ему пакет `"HELO"`.

Однако с получением пакета `"WLCM"` процедура подключения не заканчивается. Новый узел теперь должен послать всем узлам пакет включения в игру `'INTR'`. Этот пакет должен содержать идентификационный номер игрока и его имя. Благодаря этому каждый узел, участвующий в игре, гарантированно получит информацию о новом узле и сохранит ее в своих структурах данных для дальнейшего слежения за ним.

Поскольку каждый узел сохраняет адреса, полученные во входящих пакетах, теоретически может возникнуть проблема при подключении одного или нескольких узлов, находящихся в локальной сети. Например, пусть узел А — ведущий узел, а узел Б находится в той же локальной сети, что и узел А. Это означает, что карта узлов на узле А будет включать адрес узла Б в локальной сети. Теперь представьте, что новый узел В попытался соединиться с узлом А через его внешний IP-адрес. Узел А пригласит узел В в игру и передаст ему адрес узла Б. Но узел Б окажется недостижимым для узла В, потому что узел В находится за пределами локальной сети с узлами А и Б. Соответственно, узел В потерпит неудачу при попытке связаться с узлом Б и не сможет правильно вступить в игру. Эта проблема демонстрируется на рис. 6.6, *а*.

В главе 2 «Интернет» описывалось одно из решений этой проблемы, помогающее преодолеть барьер NAT. Возможны также другие решения, вовлекающие внешний сервер. В одном из таких решений установить начальное соединение между узлами помогает внешний сервер, который иногда называют *контактным сервером* (rendezvous server). Такой подход гарантирует, что любой узел сможет соединиться с любым узлом через внешний IP-адрес. Использование контактного сервера иллюстрируется на рис. 6.6, *б*.

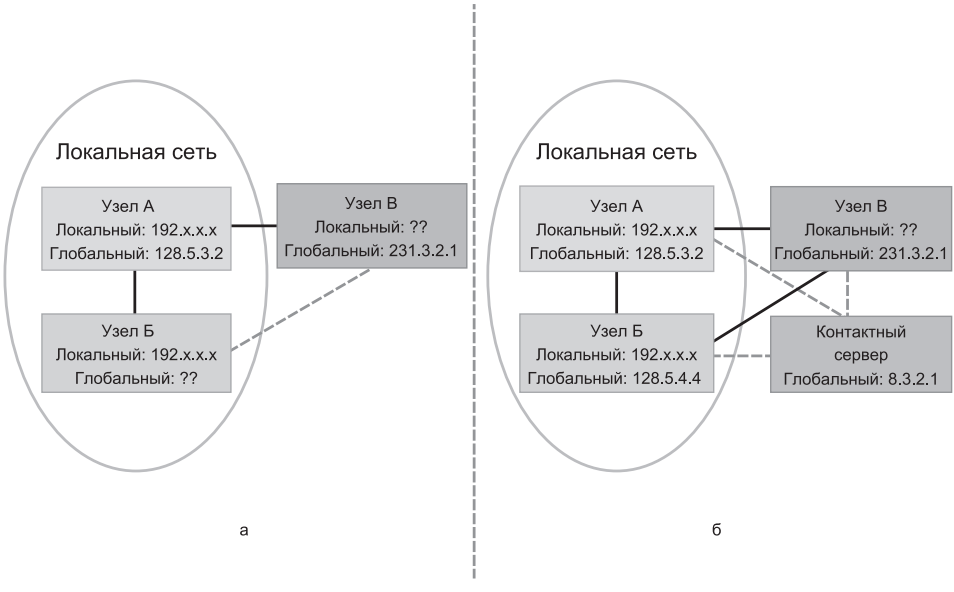


Рис. 6.6. Узел В не может подключиться к узлу Б (а); контактный сервер помогает установить начальное соединение между узлами (б)

Другой подход, используемый некоторыми игровыми службами, заключается в развертывании центрального сервера, осуществляющего маршрутизацию всех пакетов между узлами. Это означает, что весь сетевой трафик в игре сначала направляется на центральный сервер, а затем переправляется соответствующим узлам. Хотя в этом втором случае требуется намного более мощный сервер, он гарантирует, что никакой узел не узнает общедоступные IP-адреса других узлов. Это решение выглядит очень привлекательным с точки зрения безопасности, так как, например, способно предотвратить попытки разорвать связь между узлами посредством атаки вида «отказ в обслуживании».

С другой стороны, подумайте о том, как поступить, если узел не сможет соединиться с некоторыми узлами, участвующими в игре. Такое может случиться даже при использовании контактного сервера и центрального сервера, осуществляющего передачу пакетов. Простейшее решение — запретить такому узлу участвовать в игре, но чтобы отследить подобную ситуацию, придется писать дополнительный код. Так как в этой главе предполагается отсутствие каких-либо проблем с подключением, здесь не будут демонстрироваться возможные решения этой проблемы. Однако в коммерческих играх обработка такой ситуации обязательно должна быть предусмотрена.

После присоединения к игре диспетчер `NetworkManager` каждого узла переходит в состояние ожидания. Когда на ведущем узле будет нажата клавиша Enter, он отправит всем рядовым узлам пакет запуска игры (`"STRT"`). Этот пакет послужит сигналом трехсекундной готовности. Как только трехсекундный интервал истечет, игра считается начавшейся.

Обратите внимание, что прием с запускающим пакетом слишком примитивен, так как трехсекундный таймер не компенсирует сетевые задержки между ведущим узлом и всеми остальными. То есть ведущий узел всегда начинает игру раньше других узлов. Это не оказывает влияния на синхронизацию в игре благодаря модели детерминированного соответствия, но это может означать, что на ведущем узле будет наблюдаться временная пауза, дающая возможность другим узлам включиться в игру. Одно из решений проблемы заключается в том, чтобы на каждом узле вычитать 1/2 времени RTT из времени трехсекундного таймера. То есть если допустить, что величина RTT между ведущим узлом и узлом A составляет 100 мс, узел A мог бы вычесть 50 мс из общего времени ожидания перед началом, что помогло бы улучшить синхронизацию.

Обмен командами и согласование ходов

Чтобы упростить реализацию, «Robo Cat RTS» выполняется с фиксированной частотой 30 кадров в секунду, с фиксированной длительностью кадра ~33 мс. Это означает, что даже если какому-то узлу потребуется более 33 мс для отображения кадра, выполнение игровой модели все равно будет протекать так, как если бы длительность кадра составляла 33 мс. В терминах игры «Robo Cat RTS» каждый из этих 33-миллисекундных отрезков называется *этапом* (subturn). Три этапа составляют шаг игры, или ход. То есть шаг имеет продолжительность 100 мс, или, иначе говоря, каждую секунду выполняется 10 шагов. В идеале продолжительность этапов и шагов могла бы изменяться в зависимости от задержек в сети и производительности узла. Фактически это одна из тем, обсуждаемых в презентации Биттнера и Террано «Age of Empires». Однако чтобы не усложнять реализацию, в игре «Robo Cat RTS» продолжительность этапов и шагов никогда не изменяется.

Теперь что касается репликации. Каждый узел осуществляет полное моделирование игрового мира. Это означает, что игровые объекты вообще никогда не передаются между узлами. Вместо этого в ходе игры передаются только пакеты с информацией о шагах. Эти пакеты содержат список команд, отдаваемых каждым узлом на определенном шаге, вместе с парой фрагментов других ключевых данных.

Следует отметить, что в данном случае между «командами» и вводом проводится четкая грань. Например, щелчок левой кнопкой мыши на фигурке кота выбирает этого кота. Но так как выбор кота никак не влияет на состояние игры, этот щелчок не генерирует никаких команд. С другой стороны, если после выбора кота игрок щелкнул правой кнопкой мыши, это значит, что он хочет, чтобы его кот переместился в указанную точку или напал на другого кота. Так как оба этих действия влияют на состояние игры, в обоих случаях генерируется команда.

Кроме того, никакие команды не выполняются немедленно. Вместо этого каждый узел накапливает команды, отданные в течение определенного шага. В конце шага каждый узел посылает свой список команд всем остальным узлам. Этот список команд планируется для выполнения на *будущем* шаге. В частности, команда, отданная узлом на шаге x, не будет выполнена до шага $x + 2$. Здесь учитывается примерно 100 мс, необходимых для приема и обработки пакетов всеми узлами. То есть с момента отдачи команды до ее выполнения в нормальных условиях проходит

до 200 мс. Однако поскольку задержки являются естественным состоянием игры, они не оказывают отрицательного воздействия на восприятие игры участником, по крайней мере в стратегиях реального времени.

Понятие команд естественно укладывается в иерархию наследования. В частности, существует базовый класс `Command`, объявление которого приводится в листинге 6.9.

Листинг 6.9. Объявление класса Command

```
class Command
{
public:
  enum ECommandType
  {
    CM_INVALID,
    CM_ATTACK,
    CM_MOVE
  };

  Command():
  mCommandType(CM_INVALID),
  mNetworkId(0),
  mPlayerId(0)
  {}

  // на основе полученного буфера конструирует экземпляр
  // соответствующего подкласса команды
  static shared_ptr<Command> StaticReadAndCreate(
                 InputMemoryBitStream& inInputStream);

  // методы доступа к свойствам
  // ...

  virtual void Write(OutputMemoryBitStream& inOutputStream);
  virtual void ProcessCommand() = 0;

protected:
  virtual void Read(InputMemoryBitStream& inInputStream) = 0;

  ECommandType mCommandType;
  uint32_t mNetworkId;
  uint32_t mPlayerId;
};
```

Реализация класса `Command` практически не требует пояснений. Здесь определяется поле с типом команды и целочисленное поле для хранения сетевого идентификатора модуля, отдавшего команду. Виртуальная функция `ProcessCommand` вызывается, когда необходимо фактически выполнить команду. Функции `Read` и `Write` используются для чтения/записи команды в битовый поток. Функция `StaticReadAndCreate` сначала читает из битового потока тип команды, затем, опираясь на значение типа, создает экземпляр соответствующего подкласса и вызывает функцию `Read` этого подкласса.

В данном примере определено только два подкласса. Команда «переместить» (`CM_MOVE`) перемещает кота в указанную точку. Команда «напасть» (`CM_ATTACK`) сообщает коту, что тот должен напасть на кота противника. Команда перемещения имеет одну дополнительную переменную-член типа `Vector3` с координатами конечной точки перемещения. Каждый из подклассов команд имеет также собственную функцию `StaticCreate`, используемую для создания экземпляра команды и указателя `shared_ptr` на него. Реализации функций `StaticCreate` и `ProcessCommand` для команды перемещения показаны в листинге 6.10.

Листинг 6.10. Выбранные функции из MoveCommand

```
MoveCommandPtr MoveCommand::StaticCreate(uint32_t inNetworkId,
                    const Vector3& inTarget)
{
  MoveCommandPtr retVal;
  GameObjectPtr go =
    NetworkManager::sInstance->GetGameObject(inNetworkId);
  uint32_t playerId = NetworkManager::sInstance->GetMyPlayerId();

  // команда данному персонажу может быть отдана, только если
  // игрок владеет им и этот персонаж — кот
  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);
  }
}
```

Функция `StaticCreate` принимает сетевой идентификатор кота, которому отдается команда, а также координаты цели. Она также выполняет некоторые проверки, чтобы убедиться, что игровой объект существует, этот объект является котом и управляется узлом, отдавшим команду. Функция `ProcessCommand` выполняет ряд простейших проверок, чтобы убедиться, что полученный ею сетевой идентификатор совпадает с сетевым идентификатором кота и идентификатор игрока, отдавшего команду, соответствует игроку, управляющему котом. Вызов `EnterMovingState`

просто сообщает коту, что тот должен начать перемещение, что произойдет в одном из ближайших этапов. Перемещение кота реализуется точно так же, как в однопользовательских играх, поэтому здесь эта реализация не рассматривается.

Команды хранятся в списке `CommandList`. Подобно классу `MoveList` в сюжетной версии игры, `CommandList` — это лишь обертка для очереди `deque` команд. Он имеет функцию `ProcessCommands`, которая вызывает функцию `ProcessCommand` каждой команды в списке.

Диспетчер ввода каждого узла имеет экземпляр `CommandList`. Когда локальный игрок отдаст команду с помощью клавиатуры или мыши, диспетчер ввода добавит ее в список. Для инкапсуляции списка команд, а также данных, связанных с синхронизацией, для каждого законченного 100-миллисекундного шага используется класс `TurnData`. Диспетчер сети имеет поле-вектор, индекс которого соответствует номерам шагов. В каждом элементе этого вектора диспетчер сети сохраняет ассоциативный массив, ключом в котором является идентификационный номер игрока, а значением — экземпляр `TurnData` для этого игрока. Благодаря такой организации данные о шаге каждого игрока хранятся отдельно. Это помогает диспетчеру сети убедиться, что данные были приняты от всех узлов.

Закончив этап, каждый узел проверит, закончился ли шаг. Если шаг закончился, узел подготовит пакеты с информацией о выполненном шаге и отправит их всем остальным узлам. Эта функция достаточно сложна, поэтому она приводится в листинге 6.11.

Листинг 6.11. Отправка пакетов с информацией о шаге всем узлам

```
void NetworkManager::UpdateSendTurnPacket()
{
  mSubTurnNumber++;
  if (mSubTurnNumber == kSubTurnsPerTurn)
  {
    // создать экземпляр с информацией о шаге
    TurnData data(mPlayerId,
           RandGen::sInstance->GetRandomUInt32(0, UINT32_MAX),
           ComputeGlobalCRC(),
           InputManager::sInstance->GetCommandList());

    // пакет требуется отправить всем узлам
    OutputMemoryBitStream packet;
    packet.Write(kTurnCC);

    // отправляются данные для шага, отстоящего от текущего на 2 шага
    packet.Write(mTurnNumber + 2);
    packet.Write(mPlayerId);
    data.Write(packet);

    for (auto &iter: mPlayerToSocketMap)
    {
      SendPacket(packet, iter.second);
    }

    // сохранить данные для шага + 2
    mTurnData[mTurnNumber + 2].emplace(mPlayerId, data);
```

```
    InputManager::sInstance->ClearCommandList();

    if (mTurnNumber >= 0)
    {
      TryAdvanceTurn();
    }
    else
    {
      // в шаге с отрицательным номером команд не может быть
      mTurnNumber++;
      mSubTurnNumber = 0;
    }
  }
}
```

Конструктору `TurnData` передаются два параметра — случайное значение и контрольная сумма CRC, обсуждаемые в следующем разделе. А пока обратите внимание, что узел подготавливает пакет, включающий список всех команд для выполнения через два шага от текущего. Затем этот пакет отправляется всем узлам. Кроме того, узел сначала сохраняет у себя копию информации о шаге и только потом очищает список команд в диспетчере ввода.

Наконец, обратите внимание на проверку номера шага на отрицательное значение. В самом начале игры первому шагу присваивается порядковый номер –2. То есть команды, отданные на шаге с номером –2, будут запланированы для выполнения на шаге с номером 0. Это означает, что в первые 200 мс нет никаких команд для выполнения, но это не избавляет от начальной задержки — таково свойство механизма детерминированного соответствия.

Функция `TryAdvanceTurn`, представленная в листинге 6.12, получила такое имя, поскольку не гарантирует переход к следующему шагу. Это объясняется необходимостью обеспечить детерминированное соответствие шагов. По сути, если текущим является шаг x, `TryAdvanceTurn` выполнит переход к шагу $x + 1$, только если все данные для шага $x + 1$ были приняты к этому моменту. Если какие-то данные отсутствуют, диспетчер сети перейдет в режим ожидания.

Листинг 6.12. Функция TryAdvanceTurn

```
void NetworkManager::TryAdvanceTurn()
{
  // перейти к следующему шагу,
  // ТОЛЬКО ЕСЛИ получены данные от всех участников
  if (mTurnData[ mTurnNumber + 1].size() == mPlayerCount)
  {
    if (mState == NMS_Delay)
    {
      // отбросить любой ввод, полученный в период ожидания
      InputManager::sInstance->ClearCommandList();
      mState = NMS_Playing;

      // ждать 100 мс, чтобы дать медленным узлам шанс
      // наверстать упущенное
      SDL_Delay(100);
```

```
    }

    mTurnNumber++;
    mSubTurnNumber = 0;

    if (CheckSync(mTurnData[mTurnNumber]))
    {
      // обработать все команды для этого шага
      for (auto& iter: mTurnData[mTurnNumber])
      {
        iter.second.GetCommandList().ProcessCommands(iter.first);
      }
    }
    else
    {
      // для простоты в случае рассинхронизации просто завершить игру
      Engine::sInstance->SetShouldKeepRunning(false);
    }
  }
  else
  {
    // данные получены не от всех игроков — выполнить задержку :(
    mState = NMS_Delay;
  }
}
```

В состоянии задержки никакие игровые объекты не обновляются. Вместо этого диспетчер сети ждет получения недостающих пакетов с информацией о шаге. Каждый раз, когда во время задержки принимается новый пакет, диспетчер сети вновь вызывает `TryAdvanceTurn` в надежде, что новый пакет был последним из недостающих. Этот процесс продолжается до тех пор, пока не будут приняты все необходимые данные. Аналогично, если во время ожидания соединение с одним из узлов будет разорвано, этот узел будет исключен из игры, а все остальные предпримут попытку продолжить игру.

Не забывайте: эта первая версия «Robo Cat RTS» опирается на предположение, что рано или поздно будут приняты все пакеты. Чтобы исправить ситуацию с потерей пакета, состояние задержки можно дополнить определением узла, чьи команды отсутствуют, и отправкой ему запроса на повторную пересылку данных. Тогда если несколько таких запросов будут проигнорированы, замолчавший узел будет исключен из игры. Кроме того, данные для предыдущего шага можно включать в пакеты для следующего шага, чтобы в случае потери предыдущего пакета утраченные данные можно было бы найти во входящем пакете для следующего шага.

Синхронизация

Одной из самых больших сложностей в разработке игр, использующих модель «точка-точка», когда каждый узел выполняет игровую модель независимо от других, является поддержка экземпляров игры в синхронизированном состоянии. Даже незначительные несоответствия, такие как несогласованность позиций, в будущем могут перерасти в более серьезные проблемы. Если несоответствия

сохранятся, состояния игровых моделей с течением времени будут различаться все больше и больше. В какой-то момент накопившиеся различия окажутся настолько большими, что появится ощущение, будто узлы играют в разные игры! Очевидно, что подобные несоответствия недопустимы, поэтому очень важно гарантировать и проверять синхронизацию.

Синхронизация генераторов псевдослучайных чисел

Некоторые причины рассинхронизации являются более очевидными, чем другие. Например, использование *генератора псевдослучайных чисел* (Pseudo-Random Number Generator, PRNG) — единственный способ для компьютера получить числа, кажущиеся случайными. Создание случайных элементов — чрезвычайно важная особенность для многих игр, поэтому полностью избавиться от случайных чисел обычно не представляется возможным. Однако в играх, основанных на модели «точка-точка», необходимо гарантировать, что в любом шаге два узла всегда получат от генератора случайных чисел одни и те же результаты.

Если прежде вам доводилось использовать генератор случайных чисел в программах на C/C++, вы наверняка знакомы с функциями `rand` и `srand`. Функция `rand` генерирует псевдослучайное число, а функция `srand` инициализирует PRNG *начальным числом*. Для данного конкретного начального значения PRNG гарантированно будет производить одну и ту же последовательность чисел. Часто в качестве начального числа функции `srand` передается текущее значение времени. На практике это означает, что каждый раз будут генерироваться разные последовательности чисел.

С точки зрения задачи синхронизации это означает, что каждый узел должен выполнить два важных условия, чтобы обеспечить воспроизведение одних и тех же случайных чисел:

- Генератор случайных чисел на каждом узле должен инициализироваться одним и тем же начальным значением. В игре «Robo Cat RTS» ведущий узел выбирает начальное число и рассылает его в пакете запуска (`"STRT"`), чтобы каждый узел знал, какое начальное число использовать.
- Обращения к PRNG на одном и том же шаге на всех узлах всегда должны выполняться в одной и той же последовательности и в одних и тех же местах в программном коде, и число этих обращений должно совпадать для всех узлов. Это означает, что в игре не могут участвовать разные версии, использующие PRNG чаще или реже, то же относится к разным аппаратным платформам, если игра является кроссплатформенной.

Однако существует еще одна проблема, не такая очевидная на первый взгляд. Как оказывается, функции `rand` и `srand` плохо подходят для нужд синхронизации. Стандарт языка C не определяет алгоритм работы PRNG. То есть разные реализации стандартной библиотеки C на разных платформах (или даже просто скомпилированные разными компиляторами) не гарантируют использование одного и того же алгоритма PRNG. В этом случае совершенно не важно, будут ли генераторы инициированы одним и тем же начальным числом, — разные алгоритмы будут давать разные результаты. Кроме того, из-за отсутствия гарантий относительно

алгоритма PRNG, используемого функцией `rand`, качество случайных чисел, или *энтропия* значений, вызывает сомнения.

В прошлом из-за отсутствия четко определенного алгоритма работы `rand` в большинстве игр использовались собственные реализации PRNG. К счастью, в стандарт C++11 было включено определение стандартизованных и высококачественных генераторов псевдослучайных чисел. Даже при том что эти генераторы псевдослучайных чисел считаются непригодными для криптозащиты — когда случайные числа являются частью защищенного протокола, — их более чем достаточно для целей синхронизации экземпляров игры. В частности, в «Robo Cat RTS» используется реализация алгоритма Mersenne Twister, определяемого стандартом C++11. 32-разрядная версия алгоритма Mersenne Twister (MT19937) имеет период 2^{19937}, то есть последовательность чисел никогда не повторится за время существования Вселенной.

Генераторы случайных чисел в C++11 имеют несколько более сложный интерфейс, чем старые функции `rand` и `srand`, поэтому в «Robo Cat RTS» доступ к ним реализован посредством класса-обертки `RandGen`, представленного в листинге 6.13.

Листинг 6.13. Объявление класса RandGen

```
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;
};
```

Реализации некоторых функций класса `RandGen` приводятся в листинге 6.14.

Листинг 6.14. Отдельные функции из класса RandGen

```
void RandGen::StaticInit()
{
  sInstance = std::make_unique<RandGen>();

  // использовать начальное число по умолчанию
  // позднее мы выполним повторную инициализацию
  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);
}
```

Обратите внимание, что когда `RandGen` инициализируется в первый раз, в качестве начального значения используется класс `random_device`. Он поставляет случайное значение, зависящее от платформы. Устройства, генерирующие случайные числа, можно использовать для получения начального значения, но они не должны использоваться в качестве генераторов. Класс `uniform_int_distribution`, используемый в одной из функций, просто позволяет определить диапазон чисел и получать псевдослучайные числа в этом диапазоне. Данный подход предпочтительнее распространенной практики целочисленного масштабирования случайного результата. В C++11 вводится несколько дополнительных типов распределений.

Для синхронизации случайных чисел ведущий узел генерирует случайное число, которое затем используется как новое начальное значение, когда начинается обратный отсчет перед запуском игры. Это случайное число пересылается всем остальным узлам, чтобы гарантировать, что к началу шага с номером –2 все узлы инициализируют свои генераторы одним и тем же значением:

```
// выбрать начальное значение
uint32_t seed = RandGen::sInstance->GetRandomUInt32(0, UINT32_MAX);
RandGen::sInstance->Seed(seed);
```

Кроме того, когда в конце шага создается пакет с информацией о нем, каждый узел генерирует случайное целое число. Это число передается вместе с данными в составе пакета. Такое решение позволяет узлам легко проверить синхронность генераторов случайных чисел.

Имейте в виду, что если игре понадобятся случайные числа, не оказывающие влияния на ее состояние, можно создать еще один генератор. Ниже приводится пример имитации потери случайных пакетов — для этой цели нельзя использовать основной генератор игры, потому что в этом случае все узлы одновременно сымитируют потерю пакета. Однако будьте внимательны, используя несколько генераторов. Вы должны сделать все возможное, чтобы все остальные программисты, работающие над игрой, понимали, когда и какой генератор следует использовать.

Проверка синхронизации

Другие причины рассинхронизации могут быть не такими очевидными, как PRNG. Например, несмотря на очевидность детерминированной природы арифметики вещественных чисел, иногда могут возникать расхождения, обусловленные особенностями аппаратных реализаций. Так, более быстрые инструкции SIMD могут возвращать результаты, отличающиеся от результатов обычных инструкций вещественной арифметики. Кроме того, имеется возможность устанавливать

в процессоре разные флаги, изменяющие поведение арифметического устройства. Примером может служить флаг, определяющий строгое или нестрогое следование реализации IEEE 754.

Причиной рассинхронизации могут быть также банальные ошибки программиста. Может быть, программист не знал, как работает синхронизация, или просто допустил опечатку. В любом случае важно иметь в игре код, который на регулярной основе будет проверять синхронизацию. Благодаря ему ошибка, вызывающая рассинхронизацию, будет найдена и ликвидирована почти сразу после ее внесения.

Часто для проверки синхронизации используются *контрольные суммы*, подобно тому как они используются в сетевых пакетах для проверки целостности данных. Для этого в конце каждого полного шага вычисляется контрольная сумма состояния игры и затем передается в пакете с информацией о шаге, чтобы каждый узел смог убедиться, что все экземпляры игры получили одну и ту же контрольную сумму в конце шага.

Существует множество самых разных алгоритмов вычисления контрольных сумм. В «Robo Cat RTS» используется реализация алгоритма *циклической проверки четности с избыточностью* (Cyclic Redundancy Check, CRC), возвращающая 32-битное значение. Чтобы не писать собственную функцию вычисления CRC, в игре была использована функция `crc32` из открытой библиотеки zlib. Этот выбор был сделан всего лишь по причине удобства, потому что библиотека zlib уже включена в состав игры из-за необходимости поддержки файлов с изображениями в формате PNG. Кроме того, так как zlib проектировалась для обработки больших объемов данных, можно быть уверенными, что реализация вычисления CRC имеет высокую надежность и скорость работы.

В духе многократного использования программного кода функция `ComputeGlobalCRC`, представленная в листинге 6.15, задействует класс `OutputMemoryBitStream`. Все игровые объекты записывают свои данные в указанный им битовый поток посредством функции `WriteForCRC`. Запись выполняется в порядке возрастания сетевых идентификаторов. После записи всех объектов вычисляется контрольная сумма для всего буфера в потоке.

Листинг 6.15. Функция ComputeGlobalCRC

```
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;
}
```

Функция `ComputeGlobalCRC` имеет пару особенностей, которые следовало бы рассмотреть. Прежде всего, в поток записываются не все свойства игровых объектов. В случае с классом `RoboCat` записываются числовой идентификатор управляющего игрока, сетевой идентификатор, местоположение, уровень здоровья, состояние и сетевой идентификатор цели. Некоторые другие переменные-члены, такие как переменная, отслеживающая задержку между бросками клубков пряжи, не синхронизируются. Такая выборочность уменьшает время, необходимое для вычисления CRC.

Кроме того, так как CRC может вычисляться по частям, на самом деле совершенно не обязательно вычислять контрольную сумму только после записи всех данных в поток. Фактически копирование данных может оказаться менее эффективной операцией, чем вычисление CRC «на лету». Более того, можно даже написать интерфейс, похожий на `OutputMemoryBitStream`, — по сути, экземпляр класса, который просто вычисляет CRC получаемых им значений, но не сохраняет их в буфер. Однако чтобы не усложнять код, был повторно использован уже имеющийся класс `OutputMemoryBitStream`.

А теперь вернемся к задаче, ждущей решения, и вспомним, что функция `TryAdvanceTurn` в листинге 6.12 вызывает функцию `CheckSync`, когда пытается выполнить шаг. Эта функция просматривает в цикле все случайные числа и контрольные суммы, полученные от остальных узлов, и убеждается, что все узлы получили одно и то же случайное число и одно и то же значение CRC перед отправкой пакета с информацией о шаге.

Если `CheckSync` обнаружит факт рассинхронизации, «Robo Cat RTS» просто немедленно завершится. Более надежные системы могли бы использовать некоторую форму голосования. Допустим, что в игре участвуют четыре игрока. Если игроки с 1-го по 3-го вычислили контрольную сумму со значением А, а игрок 4 — со значением Б, это могло бы означать, что три экземпляра игры пока действуют синхронно. То есть игра могла бы быть продолжена при условии исключения из нее игрока 4.

ВНИМАНИЕ При разработке игр с топологией «точка-точка» и независимым выполнением игровой модели рассинхронизация доставляет немало беспокойств. Ошибки рассинхронизации часто оказываются самыми трудноуловимыми. Чтобы упростить поиск таких ошибок, важно реализовать систему журналирования, которую можно будет включить, чтобы увидеть, какие команды выполняются каждым узлом, со всеми сопутствующими подробностями.

При разработке примера игры «Robo Cat RTS» рассинхронизация могла возникать, если клиент вошел в состояние задержки, пока кот еще продолжал движение к цели. Как оказалось, это происходило потому, что после возобновления игры приостанавливавшийся экземпляр пропускал один этап. Ошибка была выявлена благодаря системе журналирования, которая фиксировала факт выполнения этапа и координаты каждого кота в конце каждого этапа. Это позволило заметить, что один из узлов пропускал этап. Без журналирования пришлось бы потратить намного больше времени, чтобы найти и исправить эту ошибку.

Намного более сложный сценарий в той же самой ситуации мог бы предусматривать репликацию полного состояния игры игроку 4, чтобы восстановить синхронизацию его экземпляра. Если объем данных игры велик, такой подход может оказаться неприменимым. Но знать о подобной возможности следует, потому что

она может пригодиться, когда в случае рассинхронизации важнее окажется не исключить, а сохранить участника в игре.

В заключение

Выбор топологии сети — одно из важнейших решений, принимаемых при создании сетевых игр. В топологии «клиент-сервер» один экземпляр игры выполняет функции сервера и управляет всей игрой. Остальные экземпляры игры являются клиентами и взаимодействуют только с сервером. Это обычно означает, что информация об объектах пересылается с сервера клиентам. В топологии «точка-точка» все экземпляры игры более или менее равноправны. Один из подходов к реализации игр с топологией «точка-точка» состоит в том, чтобы каждый узел выполнял игровую модель независимо от других.

С погружением в реализацию «Robo Cat Action» было рассмотрено несколько разных тем. Для поддержания модульной организации код игры был разбит на три отдельные цели: разделяемая библиотека, сервер и клиент. Серверный процесс приветствует новых клиентов, возвращая пакет «добро пожаловать» в ответ на пакет «привет», присланный клиентом. Система ввода клиента посылает пакеты ввода с информацией о действиях, выполняемых клиентом, включая перемещение кота и бросок клубка пряжи. Каждому клиенту на сервере соответствует свой прокси-объект, позволяющий серверу помнить, кому какие данные должны пересылаться, и хранить команды, посылаемые клиентами.

В разделе с описанием «Robo Cat RTS» обсуждались главные сложности, возникающие при создании игр с топологией «точка-точка» и выполняющие игровую модель независимо. Использование ведущего узла помогло присвоить определенной игре конкретный IP-адрес. Каждый узел хранит список адресов всех остальных узлов, участвующих в игре. Включение новых участников в игру осуществляется немного сложнее, чем в игре с топологией «клиент-сервер», потому что новый узел должен сообщить всем остальным узлам о своем существовании. Узлы сохраняют детерминированную согласованность, посылая пакеты с информацией о шаге в конце каждого 100-миллисекундного шага. Команды в этом пакете планируются для выполнения два шага спустя. Каждый узел переходит к следующему шагу, только когда все данные для предыдущего шага были приняты.

Наконец, были рассмотрены приемы синхронизации генераторов случайных чисел и использования контрольных сумм с целью обеспечить непротиворечивость состояния игры на каждом узле.

Вопросы для повторения

1. Чем отличаются обязанности клиента от обязанностей сервера в модели «клиент-сервер»?
2. Какую величину будет иметь задержка в худшем случае в игре с топологией «клиент-сервер» и сравнима ли она с худшей задержкой в игре с топологией «точка-точка»?

3. Сколько соединений устанавливается в игре с топологией «точка-точка» и сопоставимо ли это число с количеством соединений в игре с топологией «клиент-сервер»?
4. Опишите один из способов выполнения игровой модели в игре с топологией «точка-точка».
5. Текущая реализация «Robo Cat Action» не распространяет состояние ввода на несколько кадров, создавая перемещение. Реализуйте эту возможность.
6. Как можно было бы улучшить процедуру запуска в игре «Robo Cat RTS»? Реализуйте это улучшение.

Для дополнительного чтения

Bettner, Paul and Mark Terrano. «1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond». Презентация на конференции «Game Developer's Conference» в Сан-Франциско (Калифорния) в 2001 году.

7 Задержки, флуктуации и надежность

Сетевые игры действуют в суровых условиях, сражаясь за пропускную способность стареющих сетей и рассылая пакеты серверам и клиентам, разбросанным по всему миру. Зачастую данные теряются на полпути, что нетипично для локальных сетей, в которых ведется разработка. Эта глава исследует некоторые сетевые препятствия, стоящие на пути многопользовательских игр, и предлагает пути их обхода и решения проблем транспортировки, например создание собственного уровня надежности поверх транспортного протокола UDP.

Задержки

Игра, выпущенная в жестокий мир, должна уметь преодолевать трудности, отсутствующие в тепличной среде локальной сети, в которой ведется разработка. Первый из этих факторов — *задержки*. Слово «задержка» имеет разные значения для разных ситуаций. В контексте компьютерных игр под задержкой понимается интервал времени между идентифицируемой причиной и наблюдаемым эффектом. В зависимости от типа игры это может быть все что угодно: от интервала между щелчком мышью и реакцией игрового объекта на приказ в стратегии реального времени до интервала между движением руки игрока и изменением в отображении виртуальной реальности.

Несмотря на технический прогресс и старания продвинутых программистов, полностью избежать задержек все равно не удается, к тому же разные игровые жанры имеют разные допустимые пределы задержек. Игры виртуальной реальности обычно самые чувствительные, потому что мы, будучи людьми, ожидаем, что наши глаза будут наблюдать изменения в окружающей картине одновременно с поворотом головы. В таких случаях обычно задержки не должны превышать 20 мс, чтобы пользователь не терял эффект присутствия в моделируемой реальности. Следую-

щими по уровню чувствительности к задержкам являются игры-поединки, шутеры от первого лица и другие активные сюжетные игры. Задержки в таких играх могут лежать в диапазоне от 16 до 150 мс. При бóльших значениях задержек пользователь начинает ощущать вялость и плохую отзывчивость игры, независимо от частоты кадров. Самая высокая толерантность к задержкам у стратегий реального времени, что дает коду игры возможность выполнить дополнительные операции, как описывалось в главе 6 «Топологии сетей и примеры игр». Задержки в этих играх могут достигать 500 мс, не вызывая отрицательных эмоций у пользователя.

Уменьшение задержек — это один из способов, позволяющий вам как разработчику игр улучшить восприятие игры. Для этого необходимо понимать, какие факторы вносят свой вклад в задержку.

Несетевые задержки

Часто можно услышать ошибочное мнение, что сетевые задержки — главный источник задержек в играх. Действительно, обмен пакетами по сети является существенным источником задержек, но определенно не единственным. Можно назвать еще по крайней мере пять других источников, часть из которых вам неподвластна:

- **Задержка цикла ввода.** Промежуток времени между моментом, когда пользователь нажмет клавишу, и моментом, когда игра обнаружит факт нажатия, может оказаться весьма существенным. Представьте игру, выполняющуюся с частотой 60 кадров в секунду, которая опрашивает состояние джойстика в начале каждого кадра, затем обновляет состояния объектов и, наконец, отображает получившуюся сцену. Как показано на рис. 7.1, *а*, если пользователь нажмет клавишу «прыжок» спустя 2 мс после того, как игра проверит ввод, пройдет практически полный кадр, прежде чем игра обновит свое состояние в ответ на нажатую кнопку. Для клавиш, управляющих вращением сцены, можно повторно выполнить опрос их состояния в конце кадра и слегка скорректировать вывод, опираясь на изменение направления вращения, но такой прием обычно используется только в приложениях, наиболее чувствительных к задержкам. Из вышесказанного следует, что в среднем задержка между моментом нажатия клавиши и реакцией на нажатие составляет половину длительности кадра.
- **Задержка конвейера отображения.** Графические процессоры (GPU) выполняют команды отображения графики далеко не сразу после того, как центральный процессор (CPU) передаст пакет команд. Сначала драйвер добавляет команды в командный буфер, и затем GPU выполняет эти команды в какой-то момент в будущем. Чтобы отобразить очень много графики, графическому процессору может потребоваться почти полный кадр для вывода изображения на экран. На рис. 7.1, *б* показан подобный процесс, типичный для однопоточных игр. Он вносит задержку длительностью еще в один кадр.
- **Задержка многопоточного конвейера отображения.** Многопоточные игры могут вызывать еще более долгие задержки в конвейере отображения. В такой ситуации моделирование игры осуществляется одним или несколькими потоками выполнения; они изменяют игровой мир и передают результаты одному или нескольким потокам отображения. Потоки отображения затем посылают

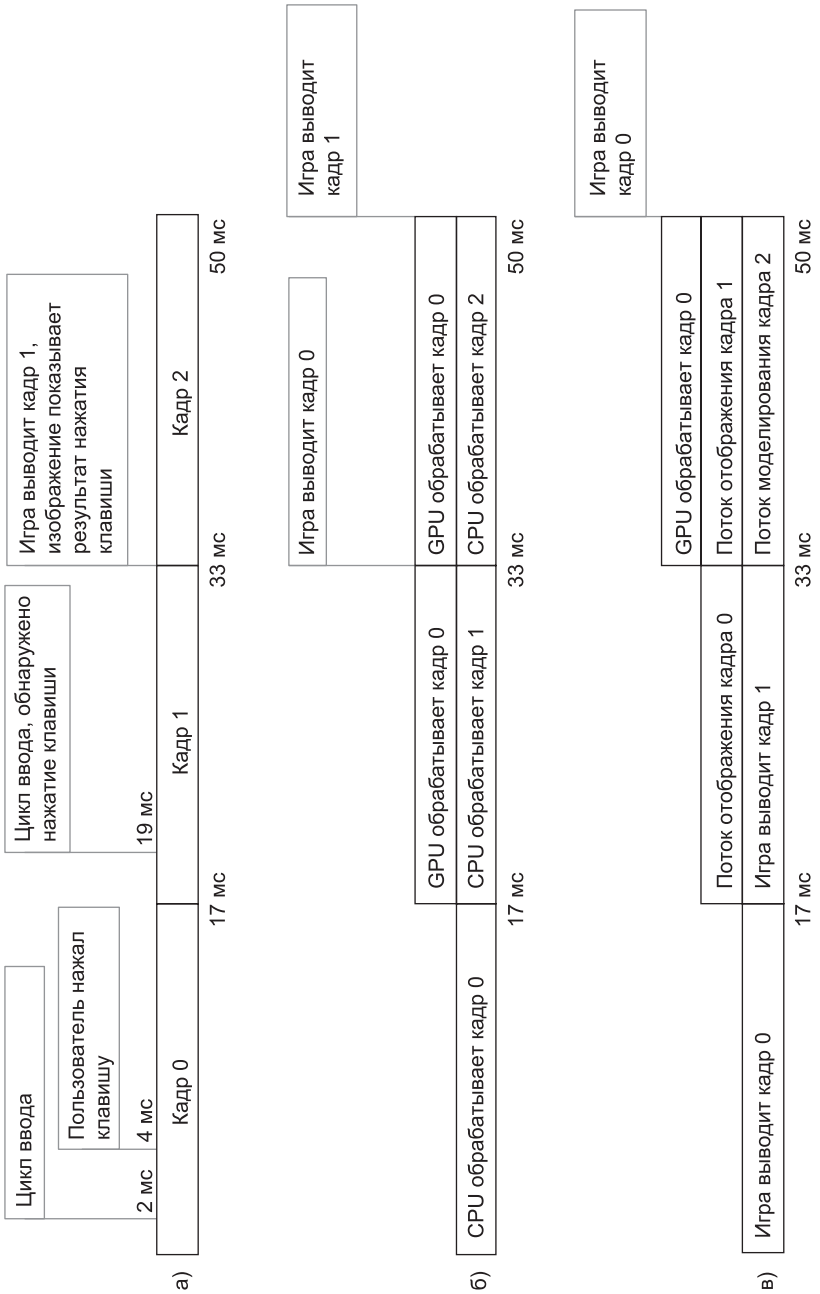


Рис. 7.1. Временны́е диаграммы задержек

пакеты команд графическому процессору, а в это время потоки моделирования уже подготовили следующий кадр. На рис. 7.1, *в* показано, как многопоточное отображение может добавить в копилку задержек еще один кадр.

- **Вертикальная синхронизация (VSync).** Чтобы избежать *разрывов* на экране, принято изменять изображение, отображаемое видеокартой, только в периоды вертикального обратного хода луча. Благодаря этому монитор никогда не будет отображать часть одного кадра и часть другого кадра одновременно. Это означает, что прежде чем передать команды графическому процессору, нужно дождаться, пока начнется обратный вертикальный ход луча на мониторе пользователя, что обычно случается каждую 1/60 секунды. Если обновление игрового кадра длится не более 16 мс, это не является проблемой. Но если для создания кадра потребуется хотя бы на 1 мс больше, к моменту, когда видеокарта будет готова изменить изображение, кадр еще не будет готов. В результате команда перемещения теневого буфера в основной будет задержана еще на 15 мс, пока не начнется следующий вертикальный обратный ход луча. Когда это случится, ваш пользователь ощутит дополнительную задержку вывода кадра.

ПРИМЕЧАНИЕ Разрыв изображения на экране происходит, если GPU копирует теневой буфер, когда монитор находится в середине цикла обновления изображения на экране. Обычно мониторы обновляют изображение по одной горизонтальной линии за раз, сверху вниз. Если изображение, выводимое на экран, изменится в середине процесса обновления, пользователь увидит в нижней части экрана новое изображение, а в верхней — старое. Если камера слишком быстро будет пересекать игровой мир, это может привести к эффекту сдвига, который выглядит, как если бы изображение было отпечатано на бумаге, затем разорвано пополам и немного сдвинуто по линии разрыва.

Большинство игр для PC содержат настройки, дающие пользователям возможность запретить вертикальную синхронизацию для повышения производительности, а некоторые новейшие жидкокристаллические мониторы, известные как G-SYNC, фактически поддерживают переменную скорость обновления, подстраиваемую под скорость отображения кадров, избавляя от задержек, связанных с вертикальной синхронизацией.

- **Частота кадров монитора.** Большинство жидкокристаллических мониторов (LCD) и мониторов высокой четкости (HDTV) обрабатывают свой ввод, прежде чем вывести изображение. Эта обработка может включать разуплотнение (de-interlacing), проверку защиты цифрового содержимого, передаваемого по широкополосным каналам (HDCP), и другие проверки, обусловленные управлением правами на цифровую интеллектуальную собственность (DRM), а также наложение визуальных эффектов, таких как масштабирование, подавление помех, выравнивание яркости, фильтрация изображения и многих других. На эту обработку тоже требуется время, и она легко может добавить десятки миллисекунд к времени ожидания пользователя. Некоторые телевизоры имеют специальный «игровой» режим, в котором задержки на дополнительную обработку сведены до минимума, но вам не стоит ориентироваться на подобную технику.
- **Время отклика пикселя.** Жидкокристаллические (LCD) мониторы страдают еще одной проблемой — пикселям требуется некоторое время, чтобы изменить свою яркость. Обычно это время измеряется единицами миллисекунд, но старые мониторы легко могут добавить задержку, составляющую половину

кадра. К счастью, данная задержка проявляется в виде следов на экране, а не как абсолютная задержка — изменение яркости начинается немедленно, но продолжаться может до нескольких миллисекунд.

Несетевые задержки представляют серьезную проблему и могут отрицательно сказываться на восприятии игры пользователем. Джон Кармак как-то раз высказался по этому поводу: «Я могу послать IP-пакет в Европу быстрее, чем пиксель на экран. Какого черта?» Учитывая величину задержки, уже имеющейся в однопользовательской игре, возникает сильное желание максимально уменьшить любые задержки, вызванные влиянием сети, при добавлении поддержки многопользовательского режима. Но для этого нужно иметь представление об основных причинах задержек, возникающих в сети.

Сетевые задержки

Несмотря на множество причин задержек, наиболее существенный вклад в многопользовательских играх обычно вносит задержка передачи пакета от узла отправителя к узлу получателя. Существуют четыре основных вида задержек, испытываемых пакетом в своем жизненном цикле:

1. **Задержка на обработку.** Вспомните, как действует любой маршрутизатор: читает пакеты из сетевого интерфейса, проверяет IP-адрес назначения, определяет следующую машину, куда передать пакет, и выводит пакет в соответствующий интерфейс. Время, затраченное на исследование адреса получателя и определение дальнейшего маршрута, называют *задержкой на обработку*. В задержку на обработку может также включаться время выполнения дополнительных операций на маршрутизаторе, таких как NAT или шифрование.
2. **Задержка на передачу.** Чтобы передать пакет дальше, маршрутизатор должен иметь интерфейс канального уровня, позволяющий пересылать пакеты через некоторый физический носитель. Протокол канального уровня управляет средней скоростью записи битов в носитель. Например, 1-мегабитное Ethernet-соединение позволяет передать в кабель Ethernet примерно 1 миллион битов в секунду. То есть запись одного бита в 1-мегабитный кабель Ethernet занимает примерно одну миллионную долю секунды (1 мкс), соответственно, запись полного 1500-байтного пакета занимает 12,5 мс. Это время, затраченное на запись битов в физический носитель, называют *задержкой на передачу*.
3. **Задержка в очереди.** Маршрутизатор может обрабатывать не более определенного числа пакетов в единицу времени. Если пакеты поступают быстрее, чем маршрутизатор их успевает обработать, они помещаются в приемную очередь, где ожидают обработки. Аналогично сетевой интерфейс может отправлять исходящие пакеты только по одному, поэтому после обработки, если соответствующий сетевой интерфейс занят, пакет помещается в исходящую очередь. Время, потраченное в этих очередях, называют *задержкой в очереди*.
4. **Задержка распространения.** Независимо от типа физического носителя, информация не может распространяться быстрее света. Соответственно, к общей задержке добавляется примерно 3,3 нс на каждый метр пути, который должен преодолеть пакет. Это означает, что даже в идеальных условиях пакету потре-

буется не менее 12 мс, чтобы пересечь территорию США. Время, проведенное в физическом носителе, называют *задержкой распространения*.

Некоторые из этих задержек можно оптимизировать, некоторые — нет. Задержка на обработку обычно оказывается самым малозначимым фактором, так как большинство современных маршрутизаторов имеют очень быстрые процессоры.

Задержка на передачу обычно зависит от типа соединения канального уровня у конечного пользователя. Пропускная способность обычно увеличивается с приближением к магистральным линиям Интернета, соответственно, на периферии задержка на передачу оказывается самой большой. Используйте высокоскоростные подключения для своих серверов — это очень важно. Кроме этого, уменьшить задержку на передачу можно, призывая конечных пользователей использовать более скоростные подключения к Интернету. Передача пакетов максимально большого размера также помогает уменьшить задержку за счет уменьшения числа байтов, расходуемых на заголовки. Чем большую долю пакетов будут занимать заголовки, тем больший вклад они будут вносить в задержку на передачу.

Задержка в очереди является результатом сохранения пакета на время ожидания перед передачей или обработкой. Минимизация затрат на обработку и передачу поможет уменьшить задержку в очереди. Стоит заметить, что для маршрутизации обычно требуется исследовать только заголовок пакета, поэтому уменьшить задержку в очереди можно, посылая несколько больших пакетов вместо множества маленьких. Например, пакет с 1400 байт данных требует таких же затрат времени на обработку, как и пакет с 200 байт. Если послать семь 200-байтных пакетов, последний будет вынужден ждать в очереди, пока не будут обработаны шесть предыдущих, в результате накопленная сетевая задержка окажется больше, чем если бы те же данные передавались в единственном большом пакете.

Задержка распространения часто является отличной целью для оптимизации. Так как эта задержка зависит от длины кабеля между узлами, обменивающимися данными, лучший способ уменьшить ее — выбирать узлы как можно ближе друг к другу. В играх с топологией «точка-точка» это означает, что при вступлении в игру предпочтение должно отдаваться менее географически удаленным узлам. В играх с топологией «клиент-сервер» это означает, что игровой сервер должен находиться как можно ближе к игрокам. Помните, что одной только географической близости недостаточно, чтобы гарантировать низкие задержки распространения: между географическими точками может не быть прямого соединения, из-за чего трафик будет направляться в обход. Выбирая место для размещения серверов, важно учитывать существующие и будущие маршруты.

ПРИМЕЧАНИЕ Иногда не получается рассредоточить игровые серверы географически, например, из-за желания охватить сетевой игрой целый континент. С такой ситуацией столкнулись разработчики известной массовой игры «League of Legends». Так как рассредоточение игровых серверов по всей стране не давало никаких преимуществ, они пошли другим путем и построили собственную сетевую инфраструктуру, договорившись с поставщиками услуг Интернета по всей Северной Америке, что последние обеспечат такое управление сетевым трафиком, которое поможет уменьшить сетевые задержки до минимально возможного уровня. Это весьма серьезное обязательство, но в случае успеха это самый очевидный и надежный способ сократить все четыре вида сетевых задержек.

В контексте сетевых взаимодействий инженеры иногда используют термин «задержка» для описания комбинации из всех четырех ее разновидностей. Поскольку термин «задержка» получился слишком общим, разработчики игр чаще пользуются понятием *время передачи/подтверждения* (Round Trip Time, RTT). Под RTT понимается время, которое требуется для передачи пакета от одного узла другому и для обратной передачи пакета ответа. Эта величина не только включает задержки на обработку, ожидание в очереди, передачу и распространение в обоих направлениях, но и учитывает скорость смены кадров на удаленном узле, так как именно от этой скорости зависит, насколько быстро узел сможет отправить ответ. Обратите внимание, что в каждом направлении пакеты могут двигаться с разной скоростью. Величина RTT редко бывает точно в два раза больше времени передачи пакета от одного узла другому. Но, как бы то ни было, в играх часто предполагается, что время передачи в одном направлении составляет ровно половину RTT.

Флуктуации

Получив достаточно хорошую оценку RTT, можно выполнить следующие шаги, которые будут описаны в главе 8 «Улучшенная обработка задержек», позволяющие ослабить влияние задержек и сделать их как можно менее заметными для клиента. Однако, работая над сетевым кодом, следует помнить, что величина RTT не всегда является константой. Для любых двух узлов время RTT между ними обычно колеблется вокруг какого-то значения, зависящего от средних значений задержек. Но любые из этих задержек могут изменяться с течением времени, что влечет за собой отклонение RTT от ожидаемой величины. Эти отклонения называют *флуктуациями*.

Любая из сетевых задержек может вносить свой вклад во флуктуации, однако для одних задержек это более характерно, чем для других:

- **Задержка на обработку.** Наименее значимая составляющая сетевой задержки — задержка на обработку — также менее всего подвержена флуктуациям. Время задержки на обработку может изменяться, когда маршрутизаторы динамически корректируют маршруты пакетов, но это оказывает очень незначительное влияние на общую величину задержки.
- **Задержка на передачу и задержка распространения.** Обе эти задержки зависят от выбора маршрута пакета: протоколы канального уровня определяют задержку на передачу, а протяженность маршрута — задержку распространения. То есть эти задержки изменяются, когда маршрутизаторы пытаются динамически балансировать трафик и изменяют маршруты, стараясь обойти перегруженные линии. Величина этих задержек может быстро изменяться в часы пик, а изменение маршрутов может существенно влиять на величину RTT.
- **Задержка в очереди.** Задержка в очереди зависит от числа пакетов, которые должен обработать маршрутизатор. С изменением числа пакетов, достигающих маршрутизатора, изменяется и задержка в очереди. В периоды большой нагрузки задержка в очереди и величина RTT могут возрастать весьма существенно.

Флуктуации отрицательно сказываются на результатах работы алгоритмов уменьшения RTT, но что еще хуже, они могут приводить к доставке пакетов не по порядку. На рис. 7.2 показано, как такое происходит. Узел А посылает пакеты 1, 2 и 3 в указанном порядке с интервалом 5 мс узлу Б. Пакету 1 потребовалось 45 мс, чтобы добраться до узла Б, но из-за внезапного увеличения трафика на маршруте пакету 2 понадобилось 60 мс. Вскоре после увеличения трафика маршрутизаторы динамически изменили маршрут, из-за чего пакету 3 потребовалось всего 30 мс, чтобы добраться до узла Б. В результате узел Б получил сначала пакет 3, затем пакет 1 и последним пакет 2.

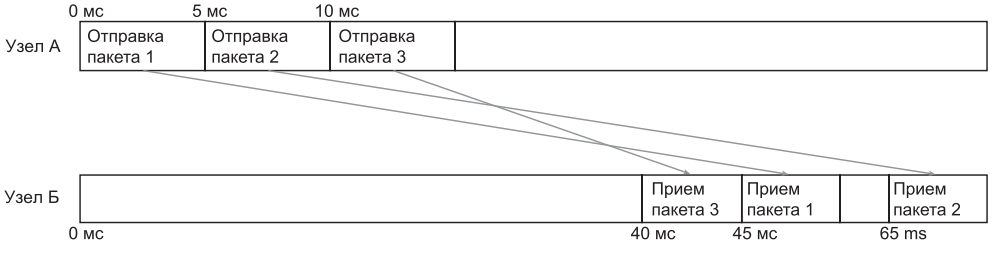


Рис. 7.2. Флуктуации вызывают нарушение порядка доставки пакетов

Чтобы предотвратить появление ошибок из-за нарушения порядка получения пакетов, используйте надежный транспортный протокол, такой как TCP, гарантирующий упорядоченную доставку пакетов, или реализуйте свою систему упорядочения пакетов, которая обсуждается во второй половине этой главы.

Старайтесь уменьшать влияние флуктуаций, насколько это возможно, тем самым вы улучшите восприятие игры. Приемы снижения влияния флуктуаций очень похожи на приемы, направленные на уменьшение задержек. Посылайте как можно меньше пакетов, чтобы уменьшить объем трафика. Размещайте серверы как можно ближе к игрокам, чтобы уменьшить вероятность столкновения с перегрузкой линий. Не забывайте, что скорость смены кадров также влияет на величину RTT, поэтому резкие изменения скорости смены кадров отрицательно скажутся на клиентах. Сложные операции должны выполняться на протяжении нескольких кадров, чтобы предотвратить флуктуации скорости их смены.

Потеря пакетов

Еще более существенной проблемой, чем задержки и флуктуации, является потеря пакетов. Одно дело, когда пакет слишком долго идет до адресата, и совсем другое, когда пакет никогда не достигает его.

Пакеты могут теряться по разным причинам.

- **Ненадежность физического носителя.** По своей природе передача данных — это передача электромагнитных импульсов. Любой внешний источник электромагнитных помех может вызвать искажение передаваемой информации. В случае повреждения данных канальный уровень обнаружит их при проверке

контрольных сумм и отбросит поврежденные кадры. Физические проблемы, такие как потеря соединения или даже близко расположенная микроволновая печь, могут вызвать повреждение или потерю сигнала.

- **Ненадежный канальный уровень.** Канальные уровни следуют своим правилам, определяющим, когда они могут или не могут передавать данные. Если линия связи канального уровня оказывается полностью забитой, исходящий кадр приходится выбрасывать. Так как канальный уровень не дает никаких гарантий надежности, это вполне оправданное действие.
- **Ненадежный сетевой уровень.** Как вы помните, когда маршрутизатор получает пакеты быстрее, чем может их обработать, он помещает их в очередь. В очереди может стоять только определенное число пакетов, их количество не может превышать некое фиксированное число. Когда очередь заполняется до предела, маршрутизатор начинает отбрасывать пакеты — либо находящиеся в очереди, либо вновь прибывающие.

Отбрасывание пакетов — суровая реальность, и потому следует проектировать сетевую архитектуру с учетом этой возможности. Независимо от наличия алгоритмов, решающих проблему потери пакетов, восприятие игры будет тем лучше, чем меньше пакетов будет теряться. Проектируя на верхнем уровне, старайтесь уменьшить вероятность потери пакетов. Используйте вычислительные центры с серверами, расположенные как можно ближе к игрокам, потому что чем меньше маршрутизаторов и линий связи на пути пакетов, тем ниже вероятность, что один из них отбросит ваши данные. Также старайтесь посылать как можно меньше пакетов: во многих маршрутизаторах емкость очереди определяется числом пакетов, а не общим объемом данных. В этих случаях риск «затопить» маршрутизаторы и переполнить очереди окажется выше, если игра будет посылать множество мелких пакетов вместо небольшого числа больших пакетов. Передача семи 200-байтных пакетов через нагруженный маршрутизатор потребует выделить для них семь слотов в очереди. Однако отправка тех же 1400 байт в одном пакете потребует только одного слота.

ВНИМАНИЕ Не все маршрутизаторы выделяют слоты в очереди по числу пакетов, некоторые выделяют место для конкретных отправителей, опираясь на входящую пропускную способность, и в этом случае пересылка данных мелкими пакетами может оказаться предпочтительнее. Если только один пакет из семи будет отброшен из-за особенностей выделения слотов с учетом пропускной способности, остальные шесть все-таки окажутся в очереди. Поэтому важно знать, как действуют маршрутизаторы в вашем вычислительном центре и вдоль высоконагруженных маршрутов, особенно потому, что использование маленьких пакетов ведет к непроизводительному расходованию полосы пропускания на передачу заголовков, о чем уже говорилось в предыдущих главах.

Когда очереди заполняются, маршрутизатор не обязательно будет отбрасывать вновь поступающие пакеты. Он может выбросить пакеты из очереди. Такое случается, когда маршрутизатор обнаружит, что входящий пакет имеет высший приоритет или более важен, чем пакет в очереди. Решения о приоритетах принимаются маршрутизаторами на основе данных QoS в заголовке сетевого уровня, а иногда основываются на результатах анализа содержимого пакетов. Некоторые маршрутизаторы даже специально настраиваются, чтобы уменьшить общий трафик, который

им придется обрабатывать: в первую очередь они начинают отбрасывать пакеты UDP и только потом пакеты TCP, потому что известно, что если отбросить пакеты TCP, отправитель автоматически повторит их передачу. Знание особенностей настройки маршрутизаторов вокруг вашего вычислительного центра и поставщиков услуг Интернета вашего целевого рынка может помочь выбрать типы пакетов и настроить шаблоны трафика, чтобы уменьшить потери пакетов. Наконец, самый простой способ уменьшить потери пакетов — использовать серверы с быстрыми и стабильными подключениями к Интернету, расположенные максимально близко к вашим клиентам.

Надежность: TCP или UDP?

Учитывая необходимость иметь некоторый уровень надежности почти в каждой многопользовательской игре, важно на самых ранних этапах разработки сделать выбор между TCP и UDP. Должна ли игра положиться на систему поддержки надежности, встроенную в TCP, или вы должны создать собственную настраиваемую систему надежности поверх UDP? Чтобы ответить на этот вопрос, нужно рассмотреть достоинства и недостатки каждого из транспортных протоколов.

Главное преимущество TCP — проверенная временем, устойчивая и стабильная реализация поддержки надежности. Без приложения дополнительных усилий этот протокол гарантирует не только доставку всех отправленных данных, но и доставку их в исходном порядке. Кроме того, он обладает сложным механизмом обнаружения заторов, ограничивающим вероятность потери пакетов путем выбора такой скорости их отправки, чтобы не вызвать переполнения в промежуточных маршрутизаторах.

Главный недостаток TCP в том, что он обязан обеспечивать гарантии надежности и упорядоченности доставки всех отправленных данных. В многопользовательской игре с быстро изменяющимся состоянием возможны три сценария, в которых такие гарантии надежности доставки порождают проблемы:

1. **Потеря низкоприоритетных данных, вызванная приемом высокоприоритетных данных.** Рассмотрим вкратце, как происходит обмен данными между двумя игроками в игре-шутере от первого лица, использующей топологию «клиент-сервер». Игрок А на узле А и игрок Б на узле Б сходятся в поединке. Неожиданно на расстоянии взрывается ракета, выпущенная из некоторого стороннего источника, и сервер посылает пакет узлу А, чтобы воспроизвести звук взрыва. Практически сразу после этого игрок Б выскакивает из-за угла перед игроком А и производит выстрел, и сервер посылает пакет с этой информацией узлу А. Из-за флуктуаций сетевого трафика первый пакет теряется, но второй, с информацией о действиях игрока Б, — нет. Звук взрыва имеет низкий приоритет для игрока А, тогда как действия врага, стреляющего в него, намного важнее. Игрок А, возможно, был бы не против, если бы потерявшийся пакет так и не пришел и он никогда не узнал бы о взрыве вдалеке. Однако из-за того, что TCP гарантирует упорядоченность пакетов, модуль TCP не передаст пакет с информацией о движении в игру сразу после приема. Вместо этого он будет

ждать, пока сервер не передаст повторно потерявшийся низкоприоритетный пакет, и только потом передаст для обработки высокоприоритетный пакет. Очевидно, такой порядок вещей может сильно огорчить игрока А.

2. **Нежелательное взаимовлияние двух отдельных упорядоченных потоков данных.** Даже в игре, где отсутствуют низкоприоритетные данные, а все данные доставляются получателям, система упорядочения протокола TCP все еще может вызывать проблемы. Представьте предыдущий сценарий, но вместо информации о взрыве первый пакет содержит текстовое сообщение, адресованное игроку А. Такие сообщения могут иметь критическую важность, поэтому для их передачи должен использоваться механизм, дающий гарантии надежной доставки. Кроме того, текстовые сообщения должны обрабатываться в определенном порядке, потому что при выводе не по порядку они могут вызвать путаницу. Однако порядок обработки текстовых сообщений имеет значение только в отношении других текстовых сообщений. Для игрока А, вероятно, было бы нежелательно, если бы потерявшийся пакет с текстовым сообщением помешал своевременной обработке пакета с информацией о произведенном выстреле. В игре, использующей TCP, такое развитие событий вполне возможно.
3. **Повторная передача устаревшего состояния игры.** Представьте, что игроки Б и А ведут поединок, отслеживая перемещения друг друга по карте. Игрок Б за 5 с перемещается из позиции $x = 0$ в позицию $x = 100$. Пять раз в секунду сервер посылает игроку А пакеты с последней координатой x игрока Б. Если сервер обнаружит, что какие-то или все пакеты потерялись, он повторит их передачу. Это означает, что когда игрок Б доберется до конечной точки с позицией $x = 100$, сервер может повторить передачу устаревших данных с позицией игрока Б, близкой к точке $x = 0$. Как результат игрок А увидит старую позицию игрока Б и произведет выстрел до того, как примет информацию, указывающую, что игрок Б переместился. Такое положение вещей совершенно неприемлемо для игрока А.

В дополнение к безальтернативной и тотальной надежности использование протокола TCP имеет еще ряд недостатков. Несмотря на то что механизм обнаружения заторов помогает предотвратить потерю пакетов, он по-разному настраивается на разных платформах, что иногда может приводить к передаче игровых пакетов с более низкой скоростью, чем хотелось бы. Алгоритм Нейгла здесь только ухудшает положение, так как может задерживать пакеты на период до половины секунды перед их отправкой. Обычно игры, использующие TCP в качестве транспортного протокола, запрещают действие алгоритма Нейгла, чтобы избежать этой проблемы, хотя при этом лишаются инструмента уменьшения потерь пакетов.

Наконец, реализация TCP выделяет ресурсы для управления соединениями и следит за всеми данными, которые может понадобиться переслать повторно. Часто этими ресурсами управляет операционная система, и очень сложно организовать управление их распределением с использованием собственного диспетчера памяти, если это потребуется.

Протокол UDP, напротив, не имеет встроенных механизмов поддержки надежности и управления потоком, как TCP. Однако он предоставляет пустой холст, на котором можно изобразить любую желаемую систему поддержки надежности.

Можно организовать надежную и ненадежную передачу данных или оформить несколько отдельных упорядоченных потоков надежной передачи данных. Можно также сконструировать систему, посылающую взамен потерявшихся пакетов только самую свежую информацию вместо повторной передачи устаревших данных. Можно разработать собственные механизмы управления памятью и получить полный контроль над группировкой данных в пакеты сетевого уровня.

Для разработки и тестирования всего этого требуется время. Собственная реализация, конечно же, не получится такой же зрелой и свободной от ошибок, как TCP. Затраты на разработку можно несколько уменьшить, пользуясь сторонними библиотеками UDP, такими как RakNet или Photon, правда, при этом придется пожертвовать определенной долей гибкости. Кроме того, применение UDP увеличивает риск потери пакетов, потому что маршрутизаторы могут быть настроены на первоочередное отбрасывание пакетов UDP, как описывалось выше. Таблица 7.1 суммирует различия между протоколами.

Таблица 7.1. Сравнение TCP и UDP

Характеристика	**TCP**	**UDP**
Надежность	Да. Все пакеты доставляются и обрабатываются в том же порядке, в каком были отправлены	Нет. Требуется собственная реализация, но позволяет организовать гибкую систему надежности
Управление потоком	Автоматически уменьшает скорость передачи, когда пакеты начинают теряться	Нет. В случае необходимости требуется реализовать свою систему управления потоком и определения заторов
Требования к памяти	ОС должна хранить копии всех отправленных данных до подтверждения их получения	Собственная реализация должна сама решить, какие данные должны храниться, а какие немедленно уничтожаться. Управление памятью осуществляется на уровне приложения
Присваивание приоритетов в маршрутизаторах	Может пользоваться преимуществом перед UDP	Пакеты UDP могут начать отбрасываться раньше, чем пакеты TCP

В большинстве случаев выбор транспортного протокола сводится к вопросам: требуется ли обязательное получение всех фрагментов данных, отправляемых игрой, и должны ли данные обрабатываться в строго определенном порядке? Если ответы в обоих случаях «да», подумайте об использовании протокола TCP. Часто это верное решение для пошаговых игр. Все фрагменты ввода должны быть получены всеми узлами и обработаны в одном и том же порядке, и протокол TCP вполне соответствует этим требованиям.

Если TCP не является идеальным выбором для игры, а таких игр большинство, следует использовать UDP с системой поддержки надежности на уровне приложения. Это означает, что придется использовать сторонние промежуточные решения или создавать свою собственную систему. В оставшейся части главы рассказывается, как создаются такие системы.

Извещение о доставке пакета

Если протокол UDP окажется наиболее подходящим выбором для игры, вам понадобится реализовать систему поддержки надежности доставки. Первое требование к такой системе — возможность узнать, был ли доставлен пакет получателю. Для этого нужно создать некоторый модуль, извещающий о доставке. Задача модуля — помогать вышестоящим модулям посылать пакеты удаленным узлам и сообщать им, были ли приняты отправленные пакеты. Не осуществляя повторную передачу самостоятельно, этот модуль позволит другим модулям самим решить, нужна ли повторная передача и какие данные при этом должны передаваться. Это главный источник гибкости системы поддержки надежности поверх UDP, которой так не хватает в TCP. В этом разделе исследуется `DeliveryNotificationManager`, одна из возможных реализаций такого модуля, идея которого была подсмотрена в диспетчере соединений игры «Starsiege: Tribes».

Диспетчер `DeliveryNotificationManager` должен решать три задачи:

1. В процессе отправки уникально идентифицировать и маркировать каждый исходящий пакет, чтобы можно было присвоить состояние доставки всем пакетам и передавать это состояние вышестоящим модулям некоторым способом.
2. На стороне получателя он должен исследовать входящие пакеты и послать подтверждение для каждого пакета, который решено будет обработать.
3. На стороне отправителя он должен обработать входящие подтверждения и сообщить вышестоящим модулям, какие пакеты были обработаны, а какие потеряны.

Такая система поддержки надежности поверх UDP также гарантирует, что пакеты никогда не будут обрабатываться не по порядку. То есть если старый пакет достигнет получателя после более новых пакетов, `DeliveryNotificationManager` сымитирует потерю пакета и проигнорирует его. Это очень полезно, так как предотвратит затирание свежей информации устаревшими данными из опоздавших пакетов. Данная особенность `DeliveryNotificationManager` несколько увеличит нагрузку, но на этом уровне такая реализация считается наиболее типичной и эффективной.

Маркировка исходящих пакетов

Диспетчер `DeliveryNotificationManager` должен идентифицировать каждый исходящий пакет, чтобы принимающая сторона могла указать, какой пакет подтверждается. Согласно методике, заимствованной из TCP, каждому пакету присваивается порядковый номер. Но в отличие от TCP порядковый номер не представляет число байтов в потоке. Он просто служит уникальным идентификатором для каждого отправляемого пакета.

Чтобы отправить пакет при помощи `DeliveryNotificationManager`, приложение создает поток `OutputMemoryBitStream` для пакета и передает его методу `DeliveryNotificationManager::WriteSequenceNumber()`, представленному в листинге 7.1.

Листинг 7.1. Маркировка пакетов последовательными номерами

```
InFlightPacket* DeliveryNotificationManager::WriteSequenceNumber(
  OutputMemoryBitStream& inPacket)
{
  PacketSequenceNumber sequenceNumber = mNextOutgoingSequenceNumber++;
  inPacket.Write(sequenceNumber);

  ++mDispatchedPacketCount;

  mInFlightPackets.emplace_back(sequenceNumber);
  return &mInFlightPackets.back();
}
```

Метод `WriteSequenceNumber` присваивает исходящему пакету очередной порядковый номер и увеличивает его, готовясь отправить следующий пакет. При такой организации никакие два соседних пакета не получат одинаковые номера, и каждый будет иметь уникальный идентификатор.

Далее метод конструирует экземпляр `InFlightPacket` и добавляет его в контейнер `mInFlightPackets`, который следит за всеми пакетами, доставка которых еще не была подтверждена. Объекты `InFlightPacket` понадобятся позднее, для обработки подтверждений и передачи признака доставки. После того как `DeliveryNotificationManager` получит возможность отметить пакет порядковым номером, приложению останется только записать фактические данные в пакет и послать его адресату.

ПРИМЕЧАНИЕ Тип `PacketSequenceNumber` определяется директивой `typedef`, поэтому не составит труда изменить число битов в порядковом номере. В данном случае тип `PacketSequenceNumber` определен как синоним `uint16_t`, но в зависимости от числа пакетов, которые предполагается послать, можно использовать числа с большим или меньшим количеством битов. Цель — использовать как можно меньше битов при минимальной вероятности быстро исчерпать диапазон доступных чисел и столкнуться с очень старым пакетом, имеющим порядковый номер, присвоенный ему до того, как диапазон был исчерпан и начал использоваться заново. В поисках минимально возможного числа битов для проверки на этапе разработки и отладки можно добавить дополнительный 32-разрядный счетчик. Затем, в процессе подготовки окончательной версии, этот счетчик следует удалить.

Прием пакетов и отправка подтверждений

Когда адресат примет пакет, он передаст `InputMemoryBitStream` с пакетом методу `ProcessSequenceNumber()` собственного экземпляра `DeliveryNotificationManager`, который представлен в листинге 7.2.

Листинг 7.2. Обработка порядковых номеров входящих пакетов

```
bool DeliveryNotificationManager::ProcessSequenceNumber(
  InputMemoryBitStream& inPacket)
{
  PacketSequenceNumber sequenceNumber;

  inPacket.Read(sequenceNumber);
```

```
  if(sequenceNumber == mNextExpectedSequenceNumber)
  {
    // ожидался? добавить подтверждение в список и обработать пакет
    mNextExpectedSequenceNumber = sequenceNumber + 1;
    AddPendingAck(sequenceNumber);
    return true;
  }

  // порядковый номер меньше ожидаемого? Просто отбросить старый пакет.
  else if(sequenceNumber < mNextExpectedSequenceNumber)
  {
    return false;
  }

  // иначе несколько пакетов было пропущено
  else if(sequenceNumber > mNextExpectedSequenceNumber)
  {
    // считать пропущенные пакеты потерявшимися и
    // определить следующий ожидаемый порядковый номер...
    mNextExpectedSequenceNumber = sequenceNumber + 1;

    // добавить подтверждение для пакета и обработать его
    // когда отправитель обнаружит разрыв в последовательности
    // подтверждений, он сможет повторить передачу
    AddPendingAck(sequenceNumber);
    return true;
  }
}
```

Метод `ProcessSequenceNumber()` возвращает значение типа `bool`, определяющее необходимость обработки пакета приложением. Именно так `DeliveryNotificationManager` предотвращает обработку пакетов не по порядку. Переменная-член `mNextExpectedSequenceNumber` хранит следующий порядковый номер, который адресат ожидает получить в пакете. Так как все отправленные пакеты маркируются последовательно увеличивающимися числами, принимающий узел легко может предсказать, какой порядковый номер должен храниться в следующем входящем пакете. С учетом этого возможны три ситуации при чтении порядкового номера:

- **Входящий порядковый номер соответствует ожидаемому.** В этом случае приложение должно подтвердить получение пакета и обработать его. Диспетчер `DeliveryNotificationManager` должен увеличить значение `mNextExpectedSequenceNumber` на 1.
- **Входящий порядковый номер меньше ожидаемого.** Такое возможно, если более старый пакет будет получен после более новых. Чтобы избежать нарушения порядка обработки, узел не должен обрабатывать такой пакет. Также он не должен подтверждать получение пакета, потому что подтверждаться должны только обрабатываемые пакеты. Здесь есть один крайний случай, который следует рассмотреть. Если текущее значение `mNextExpectedSequenceNumber` близко к максимально возможному числу, которое может быть представлено типом `PacketSequenceNumber`, а входящий порядковый номер близок к минимальному, возможна ситуация, когда произошло исчерпание диапазона доступных номеров

и началось его повторное использование. В зависимости от частоты, с которой игра посылает пакеты, и числа битов в `PacketSequenceNumber` такое возможно либо нет. Если такое возможно и значения `mNextExpectedSequenceNumber` и входящего порядкового номера предполагают такую вероятность, эту ситуацию следует обрабатывать по аналогии со следующей.

- **Входящий порядковый номер больше ожидаемого.** Такое возможно, когда один или более пакетов потерялись или задержались. Так или иначе, адресата достигает очередной пакет, но его порядковый номер оказывается выше ожидаемого. В этом случае приложение все равно должно обработать пакет и подтвердить его получение. В отличие от TCP диспетчер `DeliveryNotificationManager` не гарантирует обработку каждого пакета по порядку. Он лишь обещает не обрабатывать пакеты в ином порядке и сообщать, когда пакеты будут отбрасываться. Поэтому он спокойно может подтверждать и обрабатывать пакеты, пришедшие после того, как предыдущие пакеты были потеряны. Кроме того, чтобы предотвратить обработку более старых пакетов, если таковые будут получены, `DeliveryNotificationManager` должен присвоить своей переменной `mNextExpectedSequenceNumber` значение, на единицу большее порядкового номера текущего пакета.

ПРИМЕЧАНИЕ Первый и третий случаи фактически выполняют одну и ту же операцию. В реализации они выделены в разные ветви, только чтобы показать, что это разные ситуации, но вообще их можно объединить в одну ветвь, с проверкой условия `sequenceNumber ≥ mNextExpectedSequenceNumber`.

Метод `ProcessSequenceNumber()` сам не посылает никаких подтверждений. Он просто вызывает метод `AddPendingAck()`, чтобы сохранить порядковый номер, который должен быть подтвержден. Такой подход предпринят для эффективности. Если узел получит множество пакетов от другого узла, с его стороны было бы неэффективно посылать отдельное подтверждение для каждого входящего пакета. Даже TCP может не подтверждать каждый отдельно взятый пакет. В многопользовательской игре серверу может потребоваться послать клиенту несколько пакетов с размерами, близкими к MTU, прежде чем клиент вернет серверу какие-либо данные. В подобных случаях предпочтительнее накопить все необходимые подтверждения на стороне клиента и отправить их серверу все сразу в ближайшем пакете.

Диспетчер `DeliveryNotificationManager` может накопить несколько непоследовательных подтверждений. Для большей эффективности он сохраняет их в векторе элементов типа `AckRange` в своей переменной `mPendingAcks`. Добавление в вектор осуществляется при помощи метода `AddPendingAck()`, представленного в листинге 7.3.

Листинг 7.3. Добавление подтверждений, ожидающих отправки

```
void DeliveryNotificationManager::AddPendingAck(
PacketSequenceNumber inSequenceNumber)
{
  if(mPendingAcks.size() == 0 ||
    !mPendingAcks.back().ExtendIfShould(inSequenceNumber))
```

```
  {
    mPendingAcks.emplace_back(inSequenceNumber);
  }
}
```

Сам тип AckRange представляет коллекцию последовательных порядковых номеров для подтверждения. Он хранит первый порядковый номер в своей переменной mStart и число порядковых номеров в переменной mCount. То есть необходимость в нескольких элементах AckRange возникает, только когда обнаруживается разрыв в порядковых номерах. Реализация AckRange представлена в листинге 7.4.

Листинг 7.4. Реализация 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)
  {
    // пусть число подтверждений должно умещаться в 8 бит...
    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
```

```
  {
    // по умолчанию!
    mCount = 1;
  }
}
```

Метод `ExtendIfShould()` проверяет, можно ли добавить порядковый номер в конец диапазона, не разрывая его. Если ответ положительный, он увеличивает счетчик и сообщает вызывающей программе, что диапазон расширен. В противном случае метод возвращает `false`, чтобы вызывающая программа знала, что должна создать новый экземпляр `AckRange` для непоследовательного порядкового номера.

Методы `Write()` и `Read()` сначала сериализуют начальное число непрерывного диапазона, а затем счетчик. Вместо непосредственной сериализации счетчика эти методы учитывают, что в большинстве случаев будет подтверждаться единственный пакет. Поэтому для эффективности сериализации счетчика эти методы используют прием энтропийного кодирования с ожидаемым значением 1. Кроме того, они сериализуют счетчик как 8-разрядное целое без знака, предполагая, что больше 256 подтверждений никогда не потребуется. В действительности даже 8 бит — слишком много для счетчика, поэтому их количество можно было бы уменьшить.

Когда принимающий узел будет готов отправить пакет ответа, он запишет все накопленные подтверждения в исходящий пакет вызовом `WritePendingAcks()` сразу вслед за собственным порядковым номером. Реализация `WritePendingAcks()` приводится в листинге 7.5.

Листинг 7.5. Запись подтверждений, ожидающих отправки

```
void DeliveryNotificationManager::WritePendingAcks(
OutputMemoryBitStream& inPacket)
{
  bool hasAcks = (mPendingAcks.size() > 0);
  inPacket.Write(hasAcks);
  if(hasAcks)
  {
    mPendingAcks.front().Write(inPacket);
    mPendingAcks.pop_front();
  }
}
```

Так как подтверждения будут включаться не в каждый пакет, метод сначала записывает единственный бит, сообщающий о наличии подтверждений, а затем — единственный экземпляр `AckRange`. Так делается потому, что потеря пакета — это исключение, а не правило, и обычно будет только один ожидающий экземпляр `AckRange`. Конечно, можно организовать запись нескольких диапазонов, но для этого потребуется записать дополнительный признак с числом экземпляров `AckRange` в пакете. И потом, гибкость, безусловно, нужна, но она не должна ложиться тяжким бременем на оформление пакета с ответом. Исследование типового трафика игры поможет вам создать систему, достаточно гибкую, чтобы обрабатывать крайние случаи, и эффективную в типовых случаях: например, если вы уверены, что игре

никогда не потребуется подтверждать более одного пакета, можете вообще удалить систему подтверждения сразу нескольких пакетов и сэкономить несколько битов в пакете.

Прием подтверждений и передача состояния доставки

Отправив пакет с данными, узел должен быть готов принять и обработать любые подтверждения. Приняв пакет с подтверждениями, `DeliveryNotificationManager` определяет, какие пакеты были доставлены получателю, и извещает соответствующие модули об успешной доставке. Когда ожидаемые подтверждения не возвращаются, `DeliveryNotificationManager` делает вывод, что пакеты потерялись, и извещает соответствующие модули о неудаче.

ВНИМАНИЕ Будьте осторожны: отсутствие подтверждения не является бесспорным признаком потери пакета с данными. Данные могли быть благополучно доставлены, но потерялся пакет с подтверждениями. Отправитель не имеет никакой возможности различить эти две ситуации. В TCP такая проблема отсутствует, потому что пакет, пересылаемый повторно, имеет тот же порядковый номер, что и оригинальный пакет. Если модуль TCP примет дубликат пакета, он знает, что может просто игнорировать его.

В случае с `DeliveryNotificationManager` ситуация иная. Так как в повторной передаче не всегда используются те же данные, каждый пакет уникален и порядковые номера не используются повторно. То есть, не получив подтверждения, клиентский модуль может решить повторно послать некоторые данные, которые в действительности уже были приняты получателем. По этой причине данные должны уникально идентифицироваться вышестоящим модулем, чтобы избежать дублирования. Например, если `ExplosionManager` использует `DeliveryNotificationManager` для надежной отправки событий взрывов через Интернет, он должен уникально идентифицировать эти взрывы, чтобы один и тот же взрыв не случился дважды на стороне получателя.

Для обработки подтверждений и передачи состояния доставки используется метод `ProcessAcks()`, представленный в листинге 7.6.

Листинг 7.6. Обработка подтверждений

```
void DeliveryNotificationManager::ProcessAcks(
  InputMemoryBitStream& inPacket)
{
  bool hasAcks;
  inPacket.Read(hasAcks);
  if(hasAcks)
  {
    AckRange ackRange;
    ackRange.Read(inPacket);

    // для каждого отправленного пакета с порядковым номером,
    // меньшим начального значения подтвержденного диапазона,
    // сообщить о неудаче...
    PacketSequenceNumber nextAckdSequenceNumber =
      ackRange.GetStart();
    uint32_t onePastAckdSequenceNumber =
      nextAckdSequenceNumber + ackRange.GetCount();
```

```
    while(nextAckdSequenceNumber < onePastAckdSequenceNumber &&
       !mInFlightPackets.empty())
    {
      const auto& nextInFlightPacket = mInFlightPackets.front();

      // если номер пакета меньше подтвержденного номера,
      // значит, подтверждение о его доставке не было получено,
      // следовательно, он, вероятно, не был доставлен
      PacketSequenceNumber nextInFlightPacketSequenceNumber =
        nextInFlightPacket.GetSequenceNumber();
      if(nextInFlightPacketSequenceNumber < nextAckdSequenceNumber)
      {
        // скопировать и удалить пакет, чтобы вышестоящий
        // модуль не обнаружил его в списке ожидающих подтверждения
        auto copyOfInFlightPacket = nextInFlightPacket;
        mInFlightPackets.pop_front();
        HandlePacketDeliveryFailure(copyOfInFlightPacket);
      }
      else if(nextInFlightPacketSequenceNumber==
          nextAckdSequenceNumber)
      {
        HandlePacketDeliverySuccess(nextInFlightPacket);
        // получен!
        mInFlightPackets.pop_front();
        ++nextAckdSequenceNumber;
      }
      else if(nextInFlightPacketSequenceNumber >
          nextAckdSequenceNumber)
      {
        // некоторая часть из этого диапазона могла быть уже удалена
        // (возможно, по тайм-ауту), проверить остаток диапазона
        nextAckdSequenceNumber = nextInFlightPacketSequenceNumber;
      }
    }
  }
}
```

Чтобы обработать `AckRange`, диспетчер `DeliveryNotificationManager` должен определить, какие из прежде отправленных пакетов, хранящихся в списке `mInFlightPackets`, попадают в этот диапазон. Так как подтверждения должны приниматься по порядку, метод предполагает, что любые пакеты с порядковыми номерами ниже этого диапазона были потеряны, и сообщает о неудавшейся доставке. Затем он сообщает об успешной доставке всех пакетов, попавших в диапазон. В каждый конкретный момент в списке отправленных может храниться множество пакетов, но, к счастью, нет нужды исследовать все пакеты в списке. Так как вновь отправляемые пакеты добавляются в конец `mInFlightPackets` в порядке очередности, все пакеты располагаются в списке в порядке возрастания их номеров. То есть когда поступает новый диапазон `AckRange`, метод выполняет обход содержимого `mInFlightPackets` по порядку, сравнивая номера пакетов с диапазоном `AckRange`. Пока не будет найден первый пакет, попадающий в диапазон, он сообщает, что очередной пакет был потерян. Затем, дойдя до первого пакета из указанного диапазона, метод начинает сообщать об успешной доставке. Он продолжает сообщать

об успехе для всех пакетов, входящих в диапазон AckRange, и может прекратить обход, встретив первый пакет с номером выше этого диапазона.

Заключительный оператор else-if обрабатывает крайний случай, когда номер первого пакета в списке mInFlightPackets попадает не в начало, а в середину диапазона AckRange. Такое может случиться, если подтвержденный пакет прежде считался потерянным. В этом случае ProcessAcks() просто «перепрыгивает» к порядковому номеру пакета и сообщает, что все последующие пакеты в диапазоне были доставлены успешно.

У кого-то может возникнуть вопрос: «Как пакет, прежде считавшийся потерянным, может быть подтвержден позднее?» Такое может случиться, если подтверждение задержалось где-то в сети. Так же как TCP выполняет повторную отправку пакета в отсутствие подтверждения, диспетчер DeliveryNotificationManager тоже должен следить за подтверждениями, срок ожидания которых истек. Это особенно полезно при разреженном трафике, когда было получено непоследовательное подтверждение, говорящее о потере единственного пакета. Для проверки пакетов с истекшим временем ожидания подтверждения приложение должно в каждом кадре вызывать метод ProcessTimedOutPackets(), представленный в листинге 7.7.

Листинг 7.7. Проверка пакетов с истекшим временем ожидания подтверждения

```
void DeliveryNotificationManager::ProcessTimedOutPackets()
{
  uint64_t timeoutTime = Timing::sInstance.GetTimeMS() – kAckTimeout;
  while( !mInFlightPackets.empty())
  {
    // пакеты отсортированы, поэтому все пакеты с истекшим
    // временем ожидания подтверждения должны находиться в начале
    const auto& nextInFlightPacket = mInFlightPackets.front();
    if(nextInFlightPacket.GetTimeDispatched() < timeoutTime)
    {
      HandlePacketDeliveryFailure(nextInFlightPacket);
      mInFlightPackets.pop_front();
    }
    else
    {
      // для остальных пакетов, отправленных позже,
      // время ожидания еще не истекло
      break;
    }
  }
}
```

Метод GetTimeDispatched() возвращает время создания отправленного пакета. Так как пакеты добавляются в список mInFlightPackets по порядку, проверка должна продолжаться лишь до первого пакета с еще не истекшим временем ожидания. В этой точке гарантируется, что для всех остальных отправленных пакетов время ожидания еще не истекло.

Чтобы сообщить о доставке или потере пакетов, вышеупомянутые методы вызывают HandlePacketDeliveryFailure() и HandlePacketDeliverySuccess(), представленные в листинге 7.8.

Листинг 7.8. Слежение за состоянием доставки

```
void DeliveryNotificationManager::HandlePacketDeliveryFailure(
  const InFlightPacket& inFlightPacket)
{
  ++mDroppedPacketCount;
  inFlightPacket.HandleDeliveryFailure(this);
}

void DeliveryNotificationManager::HandlePacketDeliverySuccess(
  const InFlightPacket& inFlightPacket)
{
  ++mDeliveredPacketCount;
  inFlightPacket.HandleDeliverySuccess(this);
}
```

Эти методы увеличивают счетчики `mDroppedPacketCount` и `mDeliveredPacketCount` соответственно. Благодаря этому `DeliveryNotificationManager` может определить процент успешных доставок и оценить процент потерь на будущее. Если процент потерь окажется слишком большим, он сможет известить соответствующие модули, чтобы те снизили скорость передачи или сообщили пользователю, что что-то не так с подключением к сети. `DeliveryNotificationManager` может также сложить эти значения с размером вектора `mInFlightPacket` и сравнить полученную сумму со счетчиком `mDispatchedPacketCount`, увеличивающимся в `WriteSequenceNumber()`.

Предыдущие два метода вызывают методы `HandleDeliveryFailure()` и `HandleDeliverySuccess()` экземпляра пакета `InFlightPacket`, извещающие вышестоящий модуль о состоянии доставки. Чтобы понять, как они работают, стоит взглянуть на определение класса `InFlightPacket`, представленное в листинге 7.9.

Листинг 7.9. Класс InFlightPacket

```
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;
};
```

СОВЕТ Хранить информацию о типах передаваемых данных в виде неупорядоченного ассоциативного массива `unordered_map` очень удобно для демонстрационных целей. Однако итерации через `unordered_map` страдают неэффективностью и порождают множество промахов кэша. Если число типов данных, подлежащих передаче, невелико, намного эффективнее для каждого типа определить свою переменную-член или хранить их в фиксированном массиве с индексами, закрепленными за разными типами. Если число типов больше, чем «несколько», возможно, имеет смысл хранить их в сортированном векторе.

Каждый экземпляр `InFlightPacket` хранит контейнер с указателями на экземпляры `TransmissionData`. `TransmissionData` — это абстрактный класс со своими собственными методами `HandleDeliverySucess()` и `HandleDeliveryFailure()`. Каждый модуль, посылающий данные через `DeliveryNotificationManager`, может определить свой подкласс `TransmissionData`. Затем, записывая данные в поток пакета, модуль создаст экземпляр своего подкласса `TransmissionData` и вызовет его метод `SetTransmissionData()` для добавления в `InFlightPacket`. Когда `DeliveryNotificationManager` известит модуль об успешной или неудачной доставке пакета, модуль сможет точно узнать, что именно хранится в данном пакете, и определить, какие действия лучше предпринять. Если модулю понадобится повторно переслать какие-то данные, он сможет это сделать. Если потребуется послать более новую версию данных, он также сможет это сделать. Если возникнет необходимость изменить свои переменные, он сможет сделать и это. Таким способом `DeliveryNotificationManager` закладывает прочный фундамент для строительства системы поддержки надежности на основе UDP.

ПРИМЕЧАНИЕ Каждая пара взаимодействующих узлов имеет собственную пару экземпляров `DeliveryNotificationManager`. То есть в топологии «клиент-сервер», если сервер обслуживает 10 клиентов, он должен создать 10 экземпляров `DeliveryNotificationManager`, по одному для каждого клиента. А на каждом клиенте будет использоваться собственный экземпляр `DeliveryNotificationManager` для взаимодействий с сервером.

Надежная репликация объектов

Мы можем использовать `DeliveryNotificationManager` для обеспечения надежной доставки данных путем повторной пересылки любых данных, доставка которых потерпела неудачу. Для этого достаточно определить класс `ReliableTransmissionData`, наследующий `TransmissionData`, который будет хранить все данные, отправленные в пакете, затем создать внутри метода `HandleDeliveryFailed()` новый пакет и повторно отправить данные. Этот подход напоминает реализацию надежности в TCP, однако он не полностью использует потенциал `DeliveryNotificationManager`. Более удачная версия поддержки надежности не обязана повторно посылать те же данные. Вместо этого она могла бы посылать последнюю версию данных. В этом разделе рассказывается, как расширить `ReplicationManager` из главы 5 для поддержки повторной передачи более свежих данных по аналогии с тем, как это делает диспетчер фантомов в игре «Starsiege: Tribes».

Класс `ReplicationManager` из главы 5 имеет очень простой интерфейс. Вышестоящие модули создают потоки вывода, подготавливают пакеты и затем вызывают `ReplicateCreate()`, `ReplicateUpdate()` или `ReplicateDestroy()`, чтобы создать, изменить или удалить объект на другом конце соединения. Слабые места этого решения в том, что `ReplicationManager` не определяет, какие данные помещаются в пакеты, и не сохраняет информацию об этих данных. Это не добавляет надежности.

Чтобы обеспечить надежную доставку данных, `ReplicationManager` должен иметь возможность повторно отправлять любые данные, как только обнаружит, что пакет с этими данными был потерян. С этой целью приложение должно регулярно опрашивать `ReplicationManager`, передавая подготовленный пакет и предлагая записать в него свои данные. При такой организации как только диспетчер `ReplicationManager` обнаружит потерю пакета, он сможет записать необходимые ему данные в предоставленный пакет. Частоту предложения новых пакетов диспетчеру `ReplicationManager` приложение может регулировать, опираясь на процент потерь или любую другую эвристику.

Можно сделать еще один шаг вперед и организовать работу механизма так, чтобы `ReplicationManager` записывал данные в исходящий пакет, только когда клиент предлагает заполнить его. В этом случае вместо создания пакета при каждом изменении реплицируемых данных игровая модель могла бы просто сообщать диспетчеру `ReplicationManager` об этих данных, а уже `ReplicationManager` мог бы позаботиться об их записи при первой же возможности. В результате образуется еще один уровень абстракции между игровой моделью и сетевым кодом. Игровому коду больше не нужно создавать пакеты или выполнять сетевые операции. Вместо этого он может просто извещать `ReplicationManager` о наличии существенных изменений, а о периодической записи этих изменений в пакет позаботится сам `ReplicationManager`.

Этот же подход применим для реализации усовершенствованной поддержки надежности. Рассмотрим три основные команды: создать, изменить и удалить. Когда игровой код посылает диспетчеру `ReplicationManager` команду репликации некоторого объекта, `ReplicationManager` может использовать эту команду и объект для записи соответствующего состояния в будущий пакет. Затем он может сохранить команду, указатель на целевой объект и биты состояния в экземпляр

InFlightPacket. Если ReplicationManager обнаружит потерю пакета, он найдет соответствующий экземпляр InFlightPacket, определит команду и объект, использовавшиеся при создании потерявшегося пакета, и затем запишет свежие данные в новый пакет, используя ту же команду, объект и биты состояния. Это шаг вперед по сравнению с TCP, потому что ReplicationManager не использует исходные данные, которые могли устареть, для записи в новый пакет. Вместо этого он использует текущее состояние целевого объекта, который может оказаться на полсекунды новее, чем объект в потерявшемся пакете.

Для поддержки такой системы ReplicationManager должен иметь интерфейс, позволяющий игровой модели передавать запросы репликации. Для каждого игрового объекта модель может потребовать создать его, изменить свойства или уничтожить. ReplicationManager хранит последнюю команду репликации для каждого объекта, поэтому он может записать соответствующие данные в пакет, как только это будет предложено сделать. Команды репликации типа ReplicationCommand сохраняются в mNetworkReplicationCommand — переменной-члене, отображающей сетевые идентификаторы объектов в соответствующие им последние команды репликации. Интерфейс передачи команд репликации, а также внутреннее устройство механизма ReplicationCommand демонстрируются в листинге 7.10.

Листинг 7.10. Передача команд репликации

```
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;
}
```

Команда создания объекта отображает сетевой идентификатор объекта в экземпляр `ReplicationCommand` с кодом операции создания и битами, определяющими свойства объекта для репликации, как описывалось в главе 5. Команда изменения объекта отмечает поразрядной операцией ИЛИ (OR) биты состояния как изменившиеся, чтобы диспетчер `ReplicationManager` знал, что соответствующие им свойства требуют репликации. Игровые системы должны посылать команду изменения всякий раз, когда изменяются данные, требующие репликации. Наконец, команда уничтожения отыскивает экземпляр `ReplicationCommand` по сетевому идентификатору объекта и записывает в него код команды уничтожения. Обратите внимание, что если для объекта была отдана команда уничтожения, она заменит код любой предыдущей команды, поскольку в методологии последнего состояния бессмысленно посылать обновления для объекта, который уже был уничтожен. После передачи команды `ReplicationManager` заполнит следующий предложенный ему пакет при помощи метода `WriteBatchedCommands()`, представленного в листинге 7.11.

Листинг 7.11. Запись отданных команд

```
void ReplicationManager::WriteBatchedCommands(
  OutputMemoryBitStream& inStream, InFlightPacket* inFlightPacket)
{
  ReplicationManagerTransmissionDataPtr repTransData = nullptr;
  // обойти все команды и при необходимости выполнить репликацию
  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);
        }

        // создать данные для передачи, если это еще не было сделано
        if(!repTransData)
        {
          repTransData =
            std::make_shared<ReplicationManagerTransmissionData>(
              this);
          inFlightPacket->SetTransmissionData
            ('RPLM',repTransData);
```

```
        }

        // сохранить все, что записано в пакет, и очистить состояние
        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()` начинает с обхода массива команд репликации. Если обнаруживается сетевой идентификатор с командой, имеющей признак изменения или код команды удаления, он записывает `ReplicationHeader` и состояние в точности, как это делалось в главе 5. Затем создается экземпляр `ReplicationTransmissionData`, если он еще не был создан, и добавляется в `InFlightPacket`. Эта операция выполняется не в начале метода, а только после того, как будет выяснено, что состояние объекта изменилось и требует репликации. Затем в отправляемые данные добавляется сетевой идентификатор, код операции и биты состояния, в результате получается полная копия всего, что было записано в пакет. Наконец, в команде репликации сбрасывается признак наличия изменений, чтобы исключить повторную репликацию данных, пока они не изменятся. По завершении метода мы получаем пакет, содержащий все данные для репликации, переданные игровой моделью, и `InFlightPacket` с информацией, использованной в ходе репликации.

Когда `ReplicationManager` узнает о судьбе пакета от `DeliveryNotificationManager`, он ответит вызовом одного из двух методов, представленных в листинге 7.12.

Листинг 7.12. Ответ на извещение о состоянии доставки пакета

```
void ReplicationManagerTransmissionData::HandleDeliveryFailure(
  DeliveryNotificationManager* inDeliveryNotificationManager) const
{
  for(const ReplicationTransmission& rt: mReplications)
  {
    int networkId = rt.GetNetworkId();
    GameObject* go;
    switch(rt.GetAction())
    {
      case RA_Create:
```

```
      {
        // воссоздать, если еще не уничтожен
        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:
        // после подтверждения можно посылать как команду
        // обновления, а не создания
        mReplicationManager->HandleCreateAckd(networkId);
        break;
      case RA_Destroy:
        mReplicationManager->RemoveFromReplication(networkId);
        break;
    }
  }
}
```

`HandleDeliveryFailure()` реализуют магию усовершенствованной поддержки надежности. Если потерявшийся пакет содержал команду создания, он повторно посылает команду создания. Если пакет содержал команду изменения, устанавливаются соответствующие биты состояния, чтобы новые значения были отправлены при первой же возможности. Наконец, если пакет содержал команду удаления, повторно посылается команда удаления. В случае успешной доставки `HandleDeliverySuccess()` выполняет некоторые сопутствующие операции. Если пакет содержал команду

создания, код команды создания заменяется кодом команды изменения, чтобы объект не создавался повторно, когда в следующий раз игровая модель сообщит о его изменении. Если пакет содержал команду удаления, соответствующий сетевой идентификатор удаляется из массива `mNetworkIdToReplicationCommandMap`, потому что для него не должно посылаться никаких других команд репликации.

Оптимизация на основе отправленных пакетов

В работу `ReplicationManager` можно внести существенную оптимизацию, также подсмотренную в реализации диспетчера фантомов из игры «Starsiege: Tribes». Представьте следующую ситуацию: автомобиль движется по игровому миру в течение 1 с. Если сервер посылает клиенту состояние автомобиля 20 раз в секунду, каждый пакет будет содержать обновленную позицию автомобиля. Если пакет, отправленный спустя 0,9 с после начала движения, потеряется, может пройти 200 мс, прежде чем `ReplicationManager` на сервере обнаружит это и попытается повторить передачу. К этому моменту автомобиль может уже остановиться. Так как сервер постоянно посылал обновления в процессе движения автомобиля, на пути к клиенту уже могут находиться пакеты с обновленной информацией о местоположении автомобиля. Очевидно, что бессмысленно посылать текущие координаты автомобиля, если пакет с теми же данными уже находится в пути. Если бы у диспетчера `ReplicationManager` была возможность анализировать отправленные данные, он мог бы избежать избыточной отправки тех же данных. К счастью, такая возможность есть! Когда `ReplicationManager` впервые узнает о потере пакета, он выполняет поиск в списке `mInFlightPackets` диспетчера `DeliveryNotificationManager` и проверяет экземпляр `ReplicationTransmissionData` во всех найденных пакетах. Если обнаружится, что состояние для данного объекта и свойства уже отправлено, то делается вывод, что повторная отправка данных не требуется: они уже в пути! В листинге 7.13 приводится измененная реализация варианта `RA_Update` в методе `HandleDeliveryFailure()`, которая делает все, что описано выше.

Листинг 7.13. Предотвращение избыточной повторной отправки данных

```
void ReplicationManagerTransmissionData::HandleDeliveryFailure(
  DeliveryNotificationManager* inDeliveryNotificationManager) const
{
  ...
    case RA_Update:
      go = mReplicationManager->GetLinkingContext()
        ->GetGameObject(networkId);
      if(go)
      {
        // обыскать все отправленные пакеты,
        // сбросить признак наличия изменений
        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();
              }
            }
          }
        }
        // если признак изменения состояния не был сброшен,
        // повторить передачу
        if( state )
        {
          mReplicationManager->BatchStateDirty(networkId, state);
        }
      }
      break;
  ...
}
```

Данная реализация сначала определяет биты состояния в потерявшемся пакете, соответствующие изменившимся свойствам объекта. Затем она выполняет итерации по всем отправленным пакетам в списке `mInFlightPacket` диспетчера `DeliveryNotificationManager`. Для каждого пакета делается попытка найти запись о переданных данных, созданную диспетчером `ReplicationManager`. Если такая запись обнаруживается, выполняется поиск в содержащихся в ней экземплярах `ReplicationTransmission`. Если сетевой идентификатор объекта в найденной операции репликации совпадает с сетевым идентификатором объекта в пропавшем пакете, в исходном значении состояния сбрасываются все биты, установленные в найденном состоянии. Тем самым `ReplicationManager` пытается избежать пересылки любой информации, которая уже находится в пути. Если к окончанию проверки всех пакетов в исходном состоянии не останется ни одного установленного бита, значит, повторная передача какого бы то ни было состояния не требуется.

Вышеупомянутая «оптимизация» может потребовать значительных затрат вычислительных ресурсов в том случае, когда теряется полный пакет. Однако такая потеря происходит относительно редко, а из-за того, что полоса пропускания часто оказывается более дорогостоящим ресурсом, чем процессорное время, данная оптимизация может давать немалые выгоды. Как обычно, применимость оптимизации во многом зависит от конкретных условий в игре.

Имитация реальных условий работы

Учитывая опасности, поджидающие вашу игру в реальном мире, важно создать тестовое окружение, способное имитировать задержки, флуктуации и потерю паке-

тов. С этой целью можно сконструировать модуль тестирования, располагающийся между сокетом и остальной игрой и имитирующий реальные условия работы. Для имитации потерь пакетов определите вероятность, с которой будут отбрасываться пакеты. Затем, при поступлении очередного пакета, с помощью генератора случайных чисел решите, сбросить пакет или пропустить его в приложение. Для имитации задержек и флуктуаций выберите среднюю величину задержки и параметры распределения флуктуаций. Когда будет получен очередной пакет, следует рассчитать момент времени, когда пакет прибыл бы в реальных условиях, добавив величину задержки и флуктуации к фактическому времени прибытия. Затем, вместо немедленной передачи пакета в игру для обработки, нужно отметить его расчетным значением времени прибытия и вставить в отсортированный список пакетов. Наконец, в каждом кадре игры необходимо исследовать отсортированный список и передать для обработки только те пакеты, расчетное время прибытия которых оказалось меньше текущего времени. В листинге 7.14 показано, как все это можно организовать.

Листинг 7.14. Имитация потерь, задержек и флуктуаций

```
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)
    {
      // обработать ошибку
    }
    else
    {
      // следует ли обработать пакет?
      if(RoboMath::GetRandomFloat() >= mDropPacketChance)
      {
        // да, поместить пакет в очередь для обработки позднее
        float simulatedReceivedTime =
          Timing::sInstance.GetTimef() +
          mSimulatedLatency +
          (RoboMath::GetRandomFloat() – 0.5f) *
          mDoubleSimulatedMaxJitter;
        // вставить в список с учетом сортировки по времени
        auto it = mPacketList.end();
        while(it != mPacketList.begin())
        {
          --it;
```

```
          if(it->GetReceivedTime() < simulatedReceivedTime)
          {
            // должен прибыть после этого элемента,
            // поэтому отступить на шаг назад и прерывать цикл
            ++it;
            break;
          }
        }
        mPacketList.emplace(it, simulatedReceivedTime,
        inputStream, fromAddress);
      }
    }
  }
}

void RLSimulator::ProcessQueuedPackets()
{
  float currentTime = Timing::sInstance.GetTimef();
  // заглянуть в первый пакет...
  while(!mPacketList.empty())
  {
    ReceivedPacket& packet = mPacketList.front();
    // пора обрабатывать?
    if(currentTime > packet.GetReceivedTime())
    {
      ProcessPacket(packet.GetInputStream(),
             packet.GetFromAddress());
      mPacketList.pop_front();
    }
    else
    {
      break;
    }
  }
}
```

СОВЕТ Для еще более точной имитации можно учесть тот факт, что пакеты обычно теряются или задерживаются группами следующих друг за другом пакетов. Когда анализ случайного числа показывает, что пакет должен быть потерян, можно задействовать еще один генератор случайных чисел, чтобы с его помощью определить число последующих пакетов, которые также должны потеряться.

В заключение

Реальный мир — жестокое место для многопользовательских игр. Игроки хотят получать немедленную реакцию на свои действия, а силы природы препятствуют этому. Даже без сетевого компонента в видеоиграх есть множество причин, вызывающих задержки, включая ограниченную частоту опроса устройств ввода, задержку на подготовку изображения и задержку, связанную с особенностями работы монитора. В реальной сети многопользовательские игры сталкиваются с задержками распространения, задержками на передачу и задержками в очереди.

В арсенале разработчиков игр имеются средства для борьбы с этими задержками, но они могут оказаться слишком дорогостоящими и не подойти для вашей игры.

Флуктуации условий в сети могут приводить к запаздыванию пакетов, доставке их не по порядку или даже к потере. Чтобы дать игроку возможность получить удовольствие от игры, вы должны обеспечить определенный уровень надежности передачи. Один из способов гарантировать надежную доставку — использовать транспортный протокол TCP. Но несмотря на то что TCP является проверенным решением, он имеет несколько недостатков. TCP хорошо подходит для игр, в которых важно обеспечить надежную доставку абсолютно всех данных, но совершенно не годится для типичных игр, где актуальность данных важнее надежности доставки. Для таких игр предпочтительнее использовать протокол UDP как обеспечивающий большую гибкость.

Используя UDP, вы получаете возможность реализовать собственную поддержку надежности. Основой такой реализации обычно является система оповещения, сообщающая игре об успешной доставке пакетов или об их потере. Сохраняя информацию о каждом отправляемом пакете, игра может затем выбирать, как действовать после получения извещения о судьбе пакета.

На основе системы оповещения можно построить различные модули поддержки надежности. Обычно в ответ на событие потери пакета такой модуль выполняет повторную передачу самого последнего состояния объекта, как это делает диспетчер фантомов в игре «Starsiege: Tribes». С этой целью он следит за состоянием каждого отправленного пакета и, получив уведомление о потере пакета, осуществляет повторную передачу самой последней версии любой информации, которая уже не находится в пути.

Прежде чем развернуть игру в реальном мире, важно протестировать систему поддержки надежности в управляемом окружении. На основе генераторов случайных чисел и буфера входящих пакетов можно сконструировать систему, имитирующую потерю пакетов, задержки и флуктуации. С ее помощью вы сможете оценить, как действует поддержка надежности и вся игра в целом при различных условиях работы сети.

Покончив с низкоуровневыми проблемами реального мира, можно вплотную подойти к задержкам более высокого уровня. Глава 8 «Улучшенная обработка задержек» рассказывает, как создать у игроков ощущение почти полного отсутствия задержек.

Вопросы для повторения

1. Назовите пять составляющих несетевых задержек.
2. Назовите четыре составляющие сетевых задержек.
3. Опишите по одному из способов борьбы с каждой из сетевых задержек.
4. Как расшифровывается аббревиатура «RTT» и что она означает?
5. Что такое «флуктуация»? Какие причины могут вызывать флуктуации?

6. Дополните реализацию `DeliveryNotificationManager::ProcessSequenceNumber()` так, чтобы она правильно обрабатывала ситуацию перехода порядкового номера из максимального значения в 0.
7. Добавьте в класс `DeliveryNotificationManager` буферизацию и сохранение всех пакетов, полученных в одном кадре, до того как `DeliveryNotificationManager` решит, что какие-то пакеты устарели и их нужно отбросить.
8. Объясните, как `ReplicationManager` может использовать `DeliveryNotificationManager` для поддержки усовершенствованной надежности по протоколу TCP и отправки самых свежих данных в случае потери пакетов.
9. Используйте `DeliveryNotificationManager` и `ReplicationManager`, чтобы реализовать игру в пятнашки для двух игроков. Сымитируйте реальные условия работы, чтобы увидеть, как ваша реализация справляется с потерей пакетов, задержками и флуктуациями.

Для дополнительного чтения

Almes, G., S. Kalidindi, and M. Zekauskas. (1999, сентябрь). «A One-Way Delay Metric for IPPM». Доступно по адресу: *https://tools.ietf.org/html/rfc2679*. Проверено 28 января 2016.

Carmack, John (2012, апрель). Сообщение в Твиттере. Доступно по адресу: *https://twitter.com/id_aa_carmack/status/193480622533120001*. Проверено 28 января 2016.

Carmack, John (2012, май). «Transatlantic ping faster than sending a pixel to the screen?» Доступно по адресу: *http://superuser.com/questions/419070/transatlantic-ping-faster-than-sending-a-pixel-to-the-screen/419167#419167*. Проверено 28 января 2016.

Frohnmayer, Mark and Tim Gift. (1999). «The TRIBES Engine Networking Model». Доступно по адресу: *http://gamedevs.org/uploads/tribes-networking-model.pdf*. Проверено 28 января 2016.

Hauser, Charlie (2015, январь). «NA Server Roadmap Update: Optimizing the Internet for League and You». Доступно по адресу: *http://boards.na.leagueoflegends.com/en/c/help-support/AMupzBHw-na-server-roadmap-update-optimizing-the-internet-for-league-and-you*. Проверено 28 января 2016.

Paxson, V., G. Almes, J. Mahdavi, and M. Mathis. (1998, май). «Framework for IP Performance Metrics». Доступно по адресу: *https://tools.ietf.org/html/rfc2330*. Проверено 28 января 2016.

Savage, Phil (2015, январь). «Riot Plans to Optimise the Internet for League of Legends Players». Доступно по адресу: *http://www.pcgamer.com/riot-plans-to-optimise-the-internet-for-league-of-legends-players/*. Проверено 28 января 2016.

Steed, Anthony and Manuel Fradinho Oliveira. (2010). «Networked Graphics». Morgan Kaufman.

8 Улучшенная обработка задержек

Задержки — враги разработчиков многопользовательских игр. Ваша задача — вызвать у игроков ощущение, что игровой сервер располагается в соседней комнате, хотя в действительности он может находиться на противоположном конце страны. В этой главе исследуются некоторые приемы, помогающие создать такое ощущение.

Клиент как простой терминал

Рассмотрим топологию «клиент-сервер». Тим Суини как-то написал: «The server is the man!» (Сервер — это человек!). Этим он хотел сказать, что в сетевой системе «Unreal» сервер являлся единственным узлом, владеющим истиной и хранящим состояние игры. Это традиционное требование любого клиент-серверного окружения, устойчивого к взлому: сервер — единственный узел, выполняющий существенные игровые операции. Это означает, что всегда есть некоторая задержка между моментом, когда игрок предпринимает какое-либо действие, и моментом, когда он начинает наблюдать результат этого действия. Рисунок 8.1 иллюстрирует это на примере цикла приема/подтверждения пакета.

В этом примере *время приема/подтверждения* (Round Trip Time, RTT) между клиентом A и сервером составляет 100 мс. В момент 0 игровой персонаж игрока A на клиенте A находится в состоянии покоя, с координатой Z, равной 0. Затем игрок A нажимает клавишу, отвечающую за прыжок. Если допустить, что задержка примерно симметрична, то через 50 мс, или 1/2 RTT, пакет, несущий ввод игрока A, достигнет сервера. Приняв ввод, сервер начнет выполнение прыжка и присвоит координате Z игрового персонажа значение 1. Он отправит новое состояние, которое достигнет клиента A еще через 50 мс, или 1/2 RTT. Клиент A обновит коорди-

нату Z игрового персонажа в соответствии с состоянием, полученным от сервера, и отобразит результат на экране. То есть с момента нажатия клавиши пройдет полных 100 мс, прежде чем игрок А увидит результат.

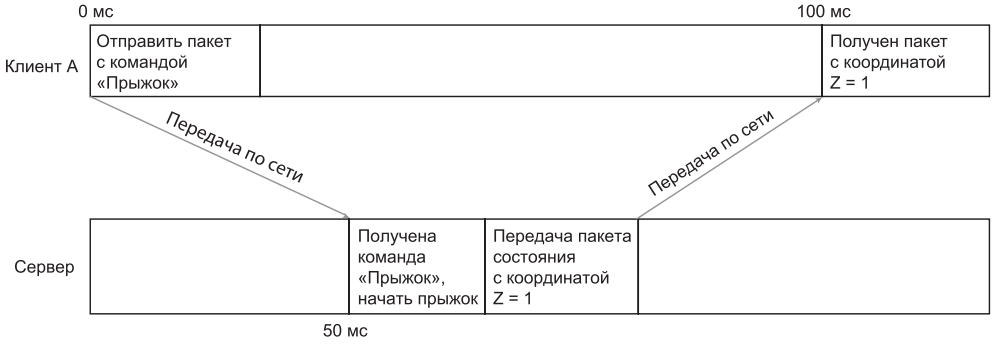


Рис. 8.1. Цикл приема/подтверждения пакета

Вывод из этого примера следующий: результаты выполнения команды на сервере появляются всегда на 1/2 RTT раньше, чем их увидит удаленный игрок. Другими словами, если игрок наблюдает только окончательные результаты выполнения игровой модели, полученные от сервера, наблюдаемое игроком состояние будет всегда старше текущего истинного состояния на сервере не менее чем на 1/2 RTT. В зависимости от загруженности сети, физической удаленности и особенностей работы промежуточных аппаратных средств это время может достигать 100 мс и больше.

Несмотря на существенную задержку между вводом и ответом, многие ранние многопользовательские игры действовали именно так. Оригинальная игра «Quake» была одной из тех игр, которые выжили, несмотря на подобную задержку. В «Quake» и многих других клиент-серверных играх того времени клиенты посылали ввод на сервер, а сервер выполнял игровую модель и рассылал результаты обратно клиентам для отображения. Клиенты в этих играх действовали как *простые терминалы* (dumb terminals): простые терминалы не выполняют никаких функций, кроме отображения информации на экране и отправки событий мыши или ввода с клавиатуры. Поскольку они всего лишь отображали состояние, полученное от сервера, они не могли отобразить неправильное состояние. Даже при том что это могло происходить с задержкой, какое бы состояние ни показал простой терминал, оно всегда было истинным для некоторого момента времени. Поскольку во всей системе состояние всегда было согласованным и никогда ошибочным, этот метод сетевых взаимодействий можно классифицировать как *консервативный алгоритм*. Пусть и ценой заметной для пользователя задержки, консервативный алгоритм по крайней мере никогда не давал неправильных результатов.

Помимо задержки, простым терминалам свойственна еще одна проблема. На рис. 8.2 приводится продолжение примера, в котором игрок А выполняет прыжок.

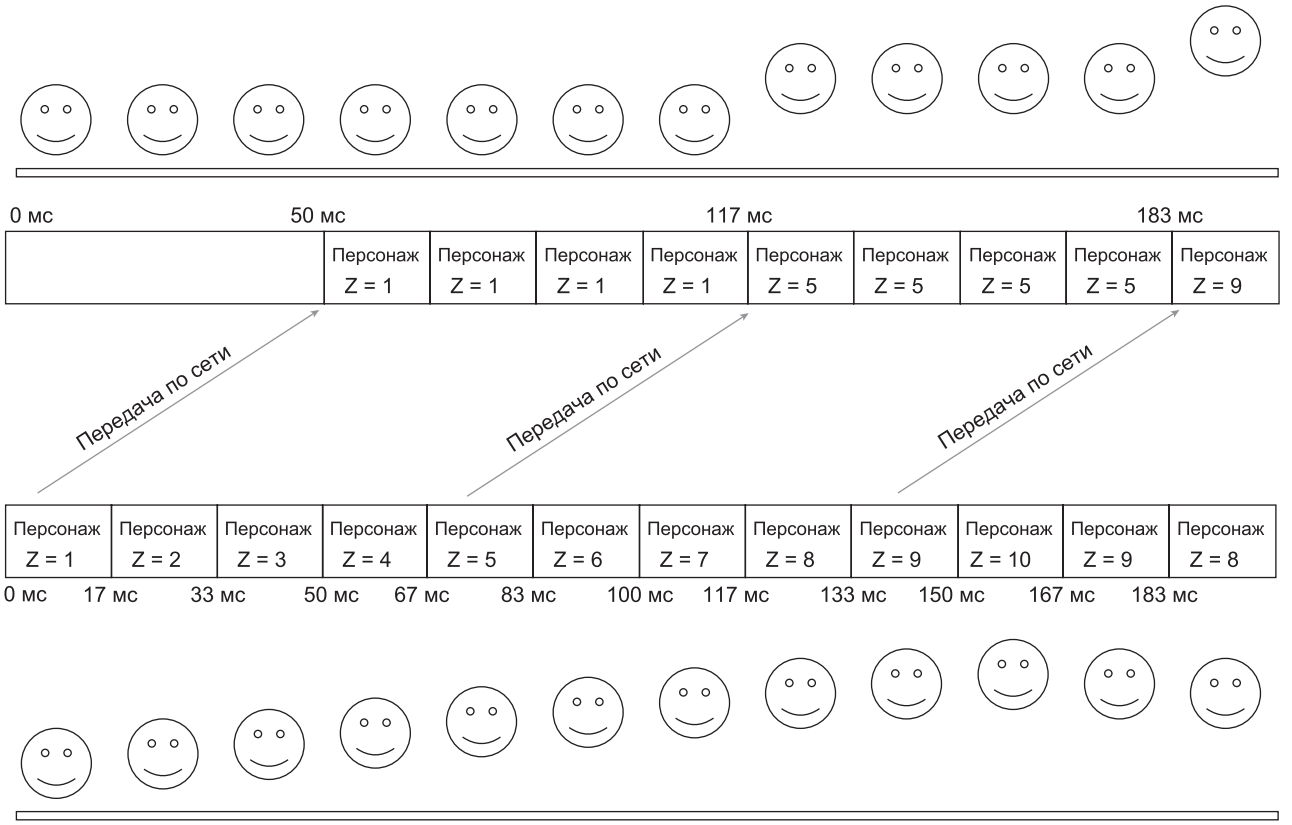


Рис. 8.2. Выполнение прыжка с частотой следования пакетов 15 раз в секунду

Благодаря использованию мощного графического процессора (GPU) клиент A может работать со скоростью 60 кадров в секунду. Сервер также может работать со скоростью 60 кадров в секунду. Но из-за ограничений пропускной способности линии связи между сервером и клиентом A сервер посылает обновления не чаще 15 раз в секунду. Допустим, что в начале прыжка игровой персонаж перемещается вверх со скоростью 60 единиц в секунду, сервер постепенно увеличивает координату Z персонажа на 1 в каждом кадре. Однако он может посылать обновления клиенту только через каждые четыре кадра. Когда клиент A принимает состояние, он обновляет координату Z игрового персонажа, но тогда персонаж должен в течение четырех кадров оставаться на одном месте, пока не поступит новая информация с сервера. Это означает, что игрок A будет видеть одну и ту же картинку на экране четыре кадра подряд. Даже при том, что игрок потратил немалые деньги на производительный GPU, способный развивать скорость до 60 кадров в секунду, он получит возможность играть с частотой 15 кадров в секунду из-за ограничений сети. Это обстоятельство определенно может вызывать недовольство.

Существует и третья проблема. Подобного рода задержки в шутере от первого лица не только вызывают ощущение плохой отзывчивости, но также мешают вести поединки с другими игроками. Без своевременного обновления позиций противников очень сложно понять, куда следует целиться. Игрок может сильно расстроиться, если, стреляя в противника, он обнаружит, что промазал только лишь потому, что фактически противник находился в этом месте 100 мс тому назад. Множество подобных нестыковок могут заставить игрока переключиться на другую игру.

Создавая игру в модели клиент-сервер, невозможно избежать проблем, связанных с задержками. Но есть возможность уменьшить их влияние на восприятие игрока, и в следующих разделах мы исследуем некоторые часто используемые приемы обработки задержек.

Интерполяция на стороне клиента

Подтормаживание, вызванное низкой частотой обновления состояния сервером, может заставить игроков думать, что игра действует медленнее, чем на самом деле. Один из способов смягчить ситуацию — использовать *интерполяцию на стороне клиента*. При использовании интерполяции на стороне клиента игровые персонажи не перемещаются скачкообразно с получением новой порции данных с сервера. Вместо этого всякий раз, когда клиент получает новое состояние объекта, он плавно интерполирует это состояние в течение некоторого интервала времени. Это называется *локальным фильтром восприятия* (local perception filter). Его работу иллюстрирует рис. 8.3.

Пусть IP (Interpolation Period) — это *период интерполяции* в миллисекундах, то есть интервал времени, в течение которого клиент будет интерполировать старое состояние в новое, а PP (Packet Period) — *период пакета* в миллисекундах, то есть интервал времени, в течение которого сервер ждет возможности послать новый пакет. Клиент завершает интерполяцию спустя IP мс после прибытия пакета. Соответственно, если IP меньше PP, клиент закончит интерполяцию раньше, чем прибудет новый пакет, и игрок все еще будет наблюдать скачкообразное

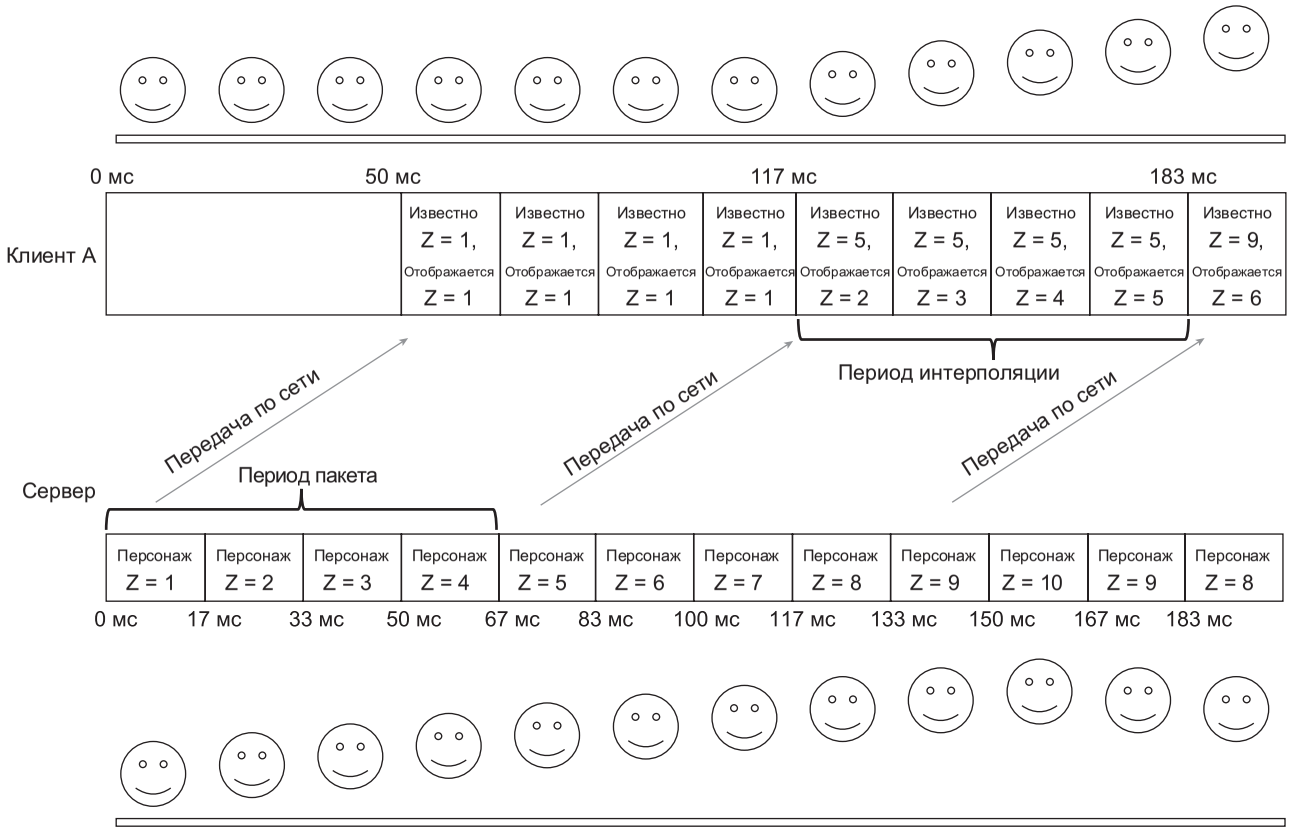


Рис. 8.3. Временна́я диаграмма интерполяции на стороне клиента

изменение картинки. Чтобы обеспечить плавное изменение состояния игры в каждом кадре, величина IP должна быть не меньше PP. В этом случае, когда клиент завершит интерполяцию до заданного состояния, он уже получит следующее состояние и сможет повторить этот процесс сначала.

Как вы наверняка помните, простой терминал, не выполняющий интерполяцию, всегда отстает от сервера на 1/2 RTT. Если не отобразить вновь прибывшее состояние сразу же, картинка перед игроком будет отставать еще больше. Игры с интерполяцией на стороне клиента отображают состояние, которое отстает от фактического примерно на 1/2 RTT + IP мс. Поэтому, чтобы уменьшить задержку, величина IP должна быть минимальной. Это требование, а также тот факт, что для устранения скачкообразного изменения состояния величина IP должна быть больше или равна величине PP, означают, что величина IP должна быть в точности равна величине PP.

С этой целью сервер может сообщить клиенту, как часто он предполагает посылать пакеты, или клиент сам может вычислить величину PP эмпирическим путем, измеряя частоту поступления пакетов. Обратите внимание, что сервер должен установить период пакета, опираясь на пропускную способность, а не на задержки. Сервер может посылать пакеты с той частотой, с какой их способна пропустить сеть между сервером и клиентом. Это означает, что воспринимаемая игроком задержка в играх такого рода, использующих интерполяцию на стороне клиента, зависит не только от сетевых задержек, но и от пропускной способности сети.

Продолжая предыдущий пример, если принять за данность, что сервер посылает данные с частотой 15 пакетов в секунду, период пакета составит 66,7 мс. Это означает добавление задержки в 66,7 мс к 1/2 RTT, равной 50 мс. Однако с интерполяцией картинка меняется более плавно, чем без интерполяции, и это обстоятельство способствует появлению более благоприятных ощущений у игрока, благодаря чему проблема задержки становится менее острой.

Игры, позволяющие игроку управлять камерой, получают дополнительное преимущество, помогающее ослабить ощущение задержки. Если местоположение камеры не оказывает существенного влияния на развитие ситуации в игре, ее перемещения можно полностью обрабатывать на стороне клиента. Передвижение и стрельба требуют передачи данных на сервер и обратно, потому что эти действия оказывают прямое влияние на игру. Но простое перемещение камеры может никак не сказываться на моделировании, и если это так, клиент может обновлять картинку на экране, не дожидаясь ответа от сервера. Локальная обработка действий с камерой дает игроку постоянную обратную связь, когда он пытается перемещать ее. В паре с плавной интерполяцией это может помочь еще больше ослабить негативные ощущения, вызванные увеличившейся задержкой.

Интерполяция на стороне клиента считается консервативным алгоритмом: несмотря на то что иногда этот прием может помочь представить состояние, которое сервер никогда не передавал явно, он отображает всего лишь промежуточные состояния между двумя точками, которые сервер действительно моделировал. Клиент сглаживает переход из одного состояния в другое, но он никогда не пытается угадать, что делает сервер, и потому никогда не окажется в неверном состоянии. Однако, как будет показано в следующих разделах, это верно не для всех приемов.

Прогнозирование на стороне клиента

Интерполяция на стороне клиента может сгладить течение игры, но она никак не приблизит игрока к тому, что в действительности происходит на сервере. Даже с очень коротким периодом интерполяции игрок все равно будет наблюдать состояние, отстающее от состояния на сервере не менее чем на 1/2 RTT. Чтобы показать состояние, максимально приближенное к истинному, игра должна переключиться с интерполяции на *экстраполяцию*. С помощью экстраполяции клиент может на основе принятого старого состояния предугадать более свежее состояние и отобразить ее на экране. Приемы, осуществляющие подобную экстраполяцию, часто называют *прогнозированием на стороне клиента*.

Чтобы экстраполировать текущее состояние, клиент должен иметь возможность выполнить ту же модель, которая выполняется на сервере. Получив новое состояние, клиент знает, что оно отстает от истинного на 1/2 RTT мс. Чтобы приблизить состояние к истинному, клиент выполняет моделирование на дополнительный интервал 1/2 RTT. В результате на экране окажется довольно точная экстраполяция состояния игровой модели на сервере. Для поддержания актуальности этого состояния клиент продолжает выполнять моделирование в каждом кадре и отображать результаты на экране. В какой-то момент клиент примет от сервера пакет с новым состоянием и смоделирует его за 1/2 RTT мс, в результате чего оно идеально совпадет с состоянием, которое клиент уже рассчитал, опираясь на предыдущее полученное состояние.

Чтобы выполнить экстраполяцию на 1/2 RTT, требуется сначала определить величину RTT. Так как часы на сервере и клиенте могут быть рассинхронизированы, простейший прием, когда сервер включает отметку времени в пакет, а клиент проверяет ее, не сработает. Вместо этого клиент должен определить полный период RTT и разделить его пополам. На рис. 8.4 показано, как это сделать.

Рис. 8.4. Определение периода RTT

Клиент посылает серверу пакет с отметкой времени по локальным часам клиента. Получив его, сервер копирует отметку времени в новый пакет и посылает обратно клиенту. Когда клиент примет новый пакет, он вычислит разность между текущими показаниями своих часов и отметкой времени в пакете и получит точное количество времени, потребовавшееся на пересылку пакета серверу и получение ответа, — оценку RTT. Используя эту информацию, клиент сможет оценить, насколько стара остальная информация в пакете, и экстраполировать полученное состояние.

ВНИМАНИЕ Не забывайте, что 1/2 RTT — это лишь приближенная оценка возраста данных. Трафик не обязательно течет в обоих направлениях с одинаковой скоростью, а потому фактическое время передачи пакета между сервером и клиентом может быть больше или меньше 1/2 RTT. Но, как бы то ни было, 1/2 RTT — достаточно хорошая оценка для многих практических применений.

В игре «Robo Cat Action», обсуждавшейся в главе 6, клиент уже посылает серверу пакеты с отметкой времени, поэтому осталось лишь организовать возврат сервером самой последней отметки времени клиенту. В листинге 8.1 показаны изменения в `NetworkManagerServer`, которые реализуют эту операцию.

Листинг 8.1. Возврат клиенту его отметки времени

```
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);
  }
}
```

Каждый входящий пакет ввода сервер передает методу `HandleInputPacket`, который вызывает метод `AddMoveIfNew` списка перемещений для каждого перемещения в пакете. `AddMoveIfNew` проверяет отметку времени в перемещении, сравнивая с отметкой времени самого последнего принятого перемещения. Если она оказывается более свежей, метод добавляет перемещение в список и обновляет значение самой последней отметки времени. Если `AddMoveIfNew` добавил какое-либо перемещение, метод `HandleInputPacket` отмечает самую последнюю отметку времени как изменившуюся, чтобы `NetworkManager` знал, что должен послать ее клиенту. Когда, наконец, придет время послать пакет клиенту, `NetworkManager` обнаружит, что отметка времени клиента изменилась, и запишет ее в пакет. Клиент примет эту отметку времени, найдет разность между ней и текущим временем и получит точный интервал времени, прошедшего с момента отправки ввода на сервер до получения соответствующего ответа.

Расчет траектории движения

Большинство аспектов игрового моделирования имеют детерминированную природу, поэтому клиент может производить экстраполяцию, просто выполняя копию серверного кода. Модель полета пули в воздухе одинакова что на сервере, что на клиенте. Мячи отскакивают от стен и от пола и подчиняются одному и тому же закону всемирного тяготения. Если клиент имеет копию программного кода, моделирующего поведение искусственного интеллекта, он сможет моделировать даже движение игровых объектов, наделенных искусственным интеллектом, чтобы максимально синхронизировать их с сервером. Однако существует один класс объектов с абсолютно недетерминированным поведением, не поддающихся точному моделированию: объекты, которые управляются человеком. Клиент не может знать, что думают удаленные игроки, какие действия они собираются инициировать или куда двигаться. Это усложняет экстраполяцию. В этом случае лучшее решение для клиента — делать предположения, а затем корректировать их по мере получения обновлений с сервера.

Расчет траектории движения в сетевой игре — это процесс предсказания поведения сущности на основе предположения, что она продолжит делать то, что

делает в настоящий момент. Если управляемый человеком персонаж бежит, можно предположить, что он продолжит бег в том же направлении. Если управляемый человеком самолет выполняет вираж, можно предположить, что он продолжит выполнять вираж.

Когда моделируемый объект управляется человеком, расчет траектории движения должен производить тот же программный код, что и на сервере, но в отсутствие изменяющегося ввода игрока. Это означает, что в дополнение к координатам объектов, управляемых игроками, сервер должен пересылать все параметры, используемые в моделировании будущих координат. В их число входят скорость, ускорение, состояние прыжка или какие-то другие величины, в зависимости от специфики игры.

Пока удаленные игроки продолжают делать то, что они делают, расчет траектории движения позволяет клиентам точно предсказывать текущее состояние на сервере. Но стоит удаленному игроку сделать неожиданное действие, и состояние игровой модели на стороне клиента отклонится от истинного состояния, поэтому оно должно быть скорректировано. Так как расчеты опираются на предположения и выполняются до получения фактической информации, расчет траектории движения не считается консервативным алгоритмом — его называют *оптимистическим алгоритмом*. Он надеется на лучшее, часто дает точный прогноз, но иногда результаты расчетов оказываются ошибочными и требуют корректировки. Эту ситуацию иллюстрирует рис. 8.5.

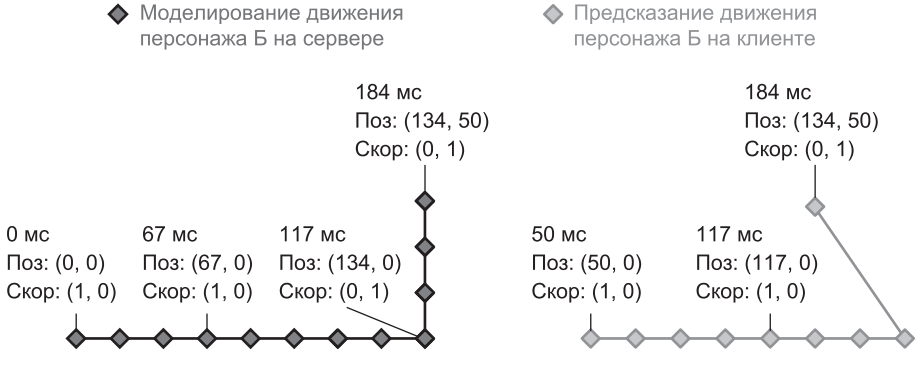


Рис. 8.5. Ошибочный расчет траектории движения

Пусть время RTT составляет 100 мс, а скорость смены кадров — 60 кадров в секунду. В момент 50 мс клиент А получает информацию, что персонаж игрока Б находится в точке (0, 0) и движется вдоль оси X в сторону положительных значений со скоростью 1 единица в миллисекунду. Так как это состояние имело место 1/2 RTT тому назад, клиент А предполагает, что персонаж Б продолжает двигаться с постоянной скоростью и через 50 мс, то есть к моменту отображения на стороне клиента А, должен оказаться в точке (50, 0). Далее в течение четырех кадров, до получения следующего пакета с состоянием, клиент А продолжает моделировать движение персонажа Б в каждом кадре. В четвертом кадре, в момент 117 мс, он

прогнозирует, что персонаж Б должен оказаться в точке (117, 0). В этот момент от сервера приходит пакет, согласно которому персонаж Б имеет скорость (1, 0) и находится в точке (67, 0). Клиент снова выполняет экстраполяцию на 1/2 RTT и обнаруживает, что истинная позиция персонажа совпала с вычисленной.

Пока все хорошо. Клиент А продолжает моделировать движение персонажа Б и в четвертом кадре предсказывает, что тот окажется в точке (184, 0). Но в этот момент он принимает пакет, в котором сервер сообщает, что персонаж находится в точке (134, 0) и его скорость стала (0, 1). Вероятно, игрок Б остановился, заметив противника, и решил атаковать его. Моделирование на 1/2 RTT вперед дает в результате позицию (134, 50) — совершенно не ту, что предсказал алгоритм расчета траектории движения на клиенте А. Игрок выполнил неожиданное, непредсказуемое действие, из-за чего состояние локальной модели на клиенте А отклонилось от истинного состояния игрового мира.

Когда клиент обнаружит, что локальная модель имеет существенные расхождения, у него есть три способа исправить ситуацию:

- **Мгновенное исправление состояния.** Игрок может заметить скачкообразное перемещение объектов, но это лучше, чем продолжать отображать неточную модель. Не забывайте, однако, что вновь полученное состояние все еще отстает от истинного состояния на сервере на 1/2 RTT, поэтому клиент должен использовать алгоритм расчета траектории движения и последнее состояние для прогнозирования позиции через 1/2 RTT.
- **Интерполяция.** Как рассказывалось выше, игра может плавно интерполировать изменение состояния в течение нескольких кадров. То есть можно вычислить и сохранить отклонение для каждого параметра с ошибочным значением (позиция, угол поворота и др.), которое затем учитывать в каждом последующем кадре. Как вариант можно просто переместить объект до половины пути к истинному состоянию и ждать, пока с сервера придет следующее состояние, чтобы продолжить коррекцию. Один из популярных методов, основывающийся на интерполяции кубическим сплайном, заключается в создании траектории, соответствующей обеим позициям и скорости для плавного перемещения из предсказанного состояния в скорректированное. Ссылка на дополнительную информацию об этой методике приводится в разделе «Для дополнительного чтения».
- **Коррекция состояния второго порядка.** Даже плавная интерполяция может вызывать неприятные ощущения из-за внезапного изменения скорости почти стационарного объекта. Для сглаживания этих изменений игра корректирует параметры второго порядка, такие как ускорение, приводя модель в правильное состояние. Этот прием требует сложных математических вычислений, но способен сделать коррекцию менее заметной.

Обычно в играх используются комбинации перечисленных методов исходя из величины расхождений и особенностей игры. В динамичных играх, таких как шутеры, для коррекции незначительных отклонений обычно используется интерполяция, а для коррекции существенных — мгновенное исправление. В менее динамичных играх, таких как имитаторы полетов или модели гигантских механических робо-

тов, используется коррекция состояния второго порядка для любых расхождений, кроме особенно больших.

Расчет траектории движения хорошо подходит для моделирования удаленных игроков, потому что локальный игрок в действительности не знает точно, что они делают. Когда игрок А видит, что персонаж игрока Б пересекает игровое поле на экране, расчетное положение персонажа отклоняется от истинного, только когда игрок Б меняет направление, но игроку А трудно определить, когда это происходит; фактически игрок А не может знать, когда игрок Б изменил ввод, если только они не находятся в одной комнате. Поэтому для него имитация выглядит непротиворечиво, даже при том, что клиентское приложение всегда предсказывает развитие событий на 1/2 RTT вперед от момента, переданного с сервера.

Прогнозирование и переигровка шагов клиента

Расчет траектории движения не может скрыть задержку от локального игрока. Представьте, что игрок А на клиенте А начинает движение вперед. Алгоритм расчета траектории использует состояние, присланное сервером, поэтому с момента, когда он нажмет на клавишу «вперед», и до момента, когда ввод попадет на сервер, пройдет 1/2 RTT, и только после этого сервер скорректирует скорость его персонажа. Затем понадобится еще 1/2 RTT, чтобы изменившееся значение скорости вернулось к клиенту А, и только после этого игра сможет задействовать алгоритм расчета траектории. В результате возникает задержка длиной в период RTT между моментом, когда игрок нажмет клавишу, и моментом, когда он увидит результат.

Существует лучшая альтернатива. Игрок А вводит все свои команды непосредственно на клиенте А, поэтому игра на клиенте А может напрямую использовать ввод для моделирования. Как только игрок А нажмет клавишу «вперед», клиент может начать моделировать движение его персонажа. Когда пакет ввода достигнет сервера, он также начнет моделирование, изменяя состояние персонажа А. Однако не все так просто.

Проблема возникает, когда сервер посылает обратно клиенту А пакет с состоянием его персонажа. Как вы помните, когда используется прием прогнозирования на стороне клиента, все входящие состояния должны моделироваться с опережением на дополнительную величину 1/2 RTT, чтобы привести локальную модель как можно ближе к истинному состоянию игрового мира. Моделируя действия удаленных игроков, клиент может просто использовать алгоритм расчета траектории и прогнозировать положение персонажей удаленных игроков, исходя из предположения неизменности ввода. Обычно обновленное входящее состояние точно соответствует состоянию, спрогнозированному клиентом, а если нет, клиент может плавно скорректировать местоположение персонажей удаленных игроков, используя прием интерполяции. Но этот способ не годится для персонажей локальных игроков. Локальные игроки точно знают, где находятся их персонажи, и сразу заметят попытку интерполяции. Они не должны наблюдать эффект дрейфа или сглаживания, когда изменяют свой ввод. В идеале перемещения должны выглядеть для локального игрока так, как если бы он играл в однопользовательскую несетевую игру.

Одно из возможных решений состоит в полном игнорировании состояния локального персонажа, полученного от сервера. Клиент А может определить состояние персонажа А, основываясь исключительно на результатах локального моделирования, и игрок А будет наблюдать гладкое, непротиворечивое перемещение своего персонажа без каких-либо задержек. К сожалению, из-за этого могут возникнуть расхождения между состоянием персонажа на клиенте А и истинным состоянием на сервере. Если персонаж игрока Б врежется в персонажа игрока А, клиент А не сможет точно спрогнозировать, как сервер обработает это столкновение. Только сервер знает истинное местоположение персонажа Б. Клиент А имеет лишь расчетную аппроксимацию, поэтому он не в состоянии обработать столкновение в точности, как это сделает сервер. На сервере персонаж А может оказаться в огненной яме, но проскочить мимо и остаться без повреждений на клиенте и тем самым вызвать неразбериху. Так как клиент А игнорирует любое входящее состояние персонажа А, клиент и сервер никогда не смогут синхронизировать свои модели.

К счастью, существует лучшее решение. Получив состояние персонажа А с сервера, клиент А сможет использовать ввод игрока А, чтобы повторно смоделировать любые изменения в состоянии, инициированные обработкой состояния на сервере. Вместо прогнозирования на величину 1/2 RTT с применением алгоритма расчета траектории движения клиент может смоделировать интервал 1/2 RTT, используя точно известный ввод пользователя А, который был задействован при первоначальном моделировании. Введя понятие *шага* и связав состояние ввода с отметкой времени, клиент сможет узнать, что делал игрок А все это время. Всякий раз, получая пакет с состоянием локального персонажа, клиент сможет выяснить, какие его шаги сервер не успел получить, приступая к вычислению этого состояния, и затем применить их локально. Если не случилось ничего неожиданного в виде события, инициированного удаленным игроком, в результате должно получиться то же самое состояние, которое клиент уже спрогнозировал.

Чтобы добавить в игру «Robo Cat Action» поддержку переигровки шагов, сначала добавим шаги в список и сохраним, пока сервер не учтет их при расчете нового состояния игры. В листинге 8.2 показаны необходимые для этого изменения.

Листинг 8.2. Сохранение шагов

```
void NetworkManagerClient::SendInputPacket()
{
  const MoveList& moveList = InputManager::sInstance->GetMoveList();
  if(moveList.HasMoves())
  {
    OutputMemoryBitStream inputPacket;
    inputPacket.Write(kInputCC);
    mDeliveryNotificationManager.WriteState(inputPacket);

    // записать три последних шага для надежности!
    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();
  }
}
```

Обратите внимание, что теперь `SendInputPacket` не очищает список шагов после отправки пакета. Шаги продолжают храниться в списке, чтобы их можно было использовать для переигровки после получения состояния от сервера. При этом мы получаем дополнительное преимущество: так как теперь шаги хранятся дольше, чем пакет, клиент посылает три последних шага из списка. То есть если любой пакет ввода будет потерян по пути к серверу, у шага будет еще две возможности добраться до сервера. Это не гарантирует надежности, но существенно увеличивает вероятность успеха.

Когда клиент получит пакет с состоянием, он вызовет `ReadLastMoveProcessedOnServerTimestamp` для обработки шагов с отметками времени, которые вернул сервер. Обнаружив такой шаг, он вычтет отметку времени из значения текущего времени, чтобы определить величину RTT, которая пригодится для расчета траектории движения. Затем вызовет `RemovedProcessedMoves`, чтобы удалить шаги с отметками времени, меньшими данной отметки или совпадающими с ней. Это означает, что по завершении `ReadLastMoveProcessedOnServerTimestamp` локальный список шагов

на клиенте будет содержать только те из них, которые еще не достигли сервера и потому должны быть применены к любому состоянию, полученному от сервера. В листинге 8.3 приводятся дополнения к методу `RoboCat::Read()`.

Листинг 8.3. Переигровка шагов

```
void RoboCatClient::Read(InputMemoryBitStream& inInputStream)
{
  float oldRotation = GetRotation();
  Vector3 oldLocation = GetLocation();
  Vector3 oldVelocity = GetVelocity();

  //... Код чтения состояния опущен ...
  bool isLocalPlayer =
    (GetPlayerId() == NetworkManagerClient::sInstance->GetPlayerId());
  if(isLocalPlayer)
  {
    DoClientSidePredictionAfterReplicationForLocalCat(readState);
  }
  else
  {
    DoClientSidePredictionAfterReplicationForRemoteCat(readState);
  }

  // если это не пакет с командой создания, сгладить любые скачки
  if(!IsCreatePacket(readState))
  {
    InterpolateClientSidePrediction(
      oldRotation, oldLocation, oldVelocity, !isLocalPlayer);
  }
}

void RoboCatClient::DoClientSidePredictionAfterReplicationForLocalCat(
  uint32_t inReadState)
{
  // переиграть шаги, только если получены новые координаты
  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)
  {
    // смоделировать движение на дополнительный интервал RTT
    float rtt = NetworkManagerClient::sInstance->GetRoundTripTime();
```

```
    // разбить на фрагменты размером framelength, чтобы не пройти
    // сквозь стену и не сотворить ничего из ряда вон...
    float deltaTime = 1.f / 30.f;
    while(true)
    {
      if(rtt < deltaTime)
      {
        SimulateMovement(rtt);
        break;
      }
      else
      {
        SimulateMovement(deltaTime);
        rtt -= deltaTime;
      }
    }
  }
}
```

Метод `Read` сначала сохраняет текущее состояние объекта, чтобы позднее он смог узнать, требуется ли сглаживать какие-нибудь корректировки. Затем он обновляет состояние, читая его из пакета, как описывалось в предыдущих главах. После обновления запускается алгоритм прогнозирования на стороне клиента, чтобы сдвинуть полученное состояние вперед на 1/2 RTT. Если реплицированный объект контролируется локальным игроком, вызывается `DoClientSidePredictionAfterReplicationForLocalCat`, чтобы произвести переигровку хода. В противном случае вызывается `DoClientSidePredictionAfterReplicationForRemoteCat`, чтобы выполнить расчет траектории движения.

`DoClientSidePredictionAfterReplicationForLocalCat` сначала проверяет, были ли получены новые координаты. Если нет, значит, дополнительное моделирование не требуется. В противном случае метод выполняет обход всех оставшихся шагов в списке и применяет их к локальному персонажу `RoboCat`. Тем самым моделируются все действия игрока, пока не учтенные сервером. Если на сервере не случилось ничего неожиданного, эта функция должна оставить кота в том же состоянии, в каком он был прежде, до вызова метода `Read`.

Если получена информация об удаленном коте, `DoClientSidePredictionAfterReplicationForRemoteCat` смоделирует новое местоположение, используя последнее известное состояние. Для этого вызывается `SimulateMovement` с соответствующим интервалом времени, без обработки ввода вызовом `ProcessInput`. И снова, если на сервере не произошло ничего неожиданного, удаленный персонаж должен остаться в том же состоянии, в каком находился до вызова `Read`. Однако в отличие от локального персонажа вероятность неожиданного изменения состояния удаленного персонажа намного выше: удаленные игроки часто отдают различные команды, изменяя направление движения, увеличивая или уменьшая скорость и т. д.

После прогнозирования на стороне клиента метод `Read()` в заключение вызывает `InterpolateClientSidePrediction()`, чтобы обработать любые изменения состояния. Получив старое состояние, метод интерполяции сможет определить, насколько велики отклонения, если они вообще имеются, и сгладит переход из старого состояния в новое.

Сокрытие задержки с применением хитростей и оптимистического алгоритма

Запаздывание реакции не единственный признак, сообщающий игроку о задержке. Когда игрок нажимает клавишу, чтобы произвести выстрел, он ожидает, что его ружье выстрелит немедленно. Когда игрок выполняет бросок, атакуя противника, он ожидает, что его персонаж немедленно бросит огненный шар. Переигровка шагов не обрабатывает подобные ситуации, поэтому необходимо что-то еще. Как правило, создание снарядов на клиенте организовать слишком сложно, потому что они создаются самим сервером, а их состояния пересылаются клиентам после создания, но есть более простое решение.

Почти все действия в видеоиграх имеют некоторые *визуальные признаки*, сообщающие, что что-то произошло. Вспышка сообщает о выстреле из бластера, а волшебники делают пассы руками и бормочут заклинания, прежде чем выплеснуть фонтан огня. Обычно длительность воспроизведения этих признаков никак не меньше времени передачи пакета на сервер и получения ответа. Это означает, что приложение-клиент может дать локальному игроку мгновенную обратную связь в ответ на любой ввод и воспроизвести соответствующий анимационный эффект, ожидая, пока моделирование будет выполнено на сервере. То есть клиент должен лишь воспроизвести начальный анимационный эффект и звук, но не создавать сам снаряд. Если все сложится удачно, пока клиент воспроизводит визуальный эффект, сервер примет пакет ввода, создаст огненный шар и перешлет его клиенту к моменту, когда его появление будет выглядеть как логичный результат броска. Алгоритм расчета траектории сдвинет снаряд вперед на 1/2 RTT, и все будет выглядеть так, как если бы игрок бросил огненный шар без задержки. Если возникнет какая-то проблема — например, если сервер знает, что недавно игрок был лишен возможности стрелять, но эта информация еще не была передана клиенту, — оптимистичный алгоритм воспроизведет анимационный эффект выстрела, а снаряд не появится. Однако это довольно редкая ситуация, и с ней можно мириться, учитывая очевидные преимущества.

Возврат на стороне сервера

Применение разнообразных приемов прогнозирования на стороне клиента может дать игрокам ощущение хорошей отзывчивости игры даже при наличии умеренных задержек. Однако существует еще один типичный вид игровых действий, плохо поддающихся прогнозированию на стороне клиента, — применение оружия мгновенного действия на больших расстояниях. Когда игрок, вооруженный снайперской винтовкой, ловит противника в перекрестье прицела и нажимает спусковой крючок, он ожидает увидеть точное попадание. Но из-за неточности расчета траектории движения вполне может получиться, что противник, находящийся точно на линии прицеливания на стороне клиента, на сервере окажется немного в стороне. Это может стать проблемой для игр, полагающихся на реалистичность и использующих оружие мгновенного действия.

Решение этой проблемы было найдено и реализовано в популярном игровом движке Valve Source Engine, легшем в основу таких игр, как «Counter-Strike», где он отвечает за ощущение точности стрельбы. Суть решения заключается в том, чтобы вернуть состояние на сервере к тому, что наблюдал игрок, когда прицеливался и производил выстрел. То есть если игрок уверен в идеальном прицеливании, попадание будет стопроцентным.

С этой целью в методы прогнозирования, обсуждавшиеся выше, следует внести несколько корректировок:

- **Использование интерполяции на стороне клиента без расчета траекторий персонажей удаленных игроков.** Сервер должен точно знать, что наблюдали игроки в любой момент времени. Поскольку прием расчета траектории опирается на опережающее моделирование, основанное на некоторых предположениях, он может привнести излишние сложности для сервера и потому должен быть отключен. Чтобы ослабить влияние любых флуктуаций или задержек, клиент, в свою очередь, должен использовать прием интерполяции, как было описано ранее в этой главе. Период интерполяции должен точно совпадать с периодом пакета, длительность которого определяется сервером. Интерполяция на стороне клиента вводит дополнительную задержку, но, как оказывается, она практически не заметна для игрока из-за переигровки шагов и действия алгоритма возврата на стороне сервера.
- **Использование прогнозирования и переигровки шагов для персонажа локального игрока.** Несмотря на то что клиент не должен прогнозировать поведение удаленных игроков, это должно осуществляться для локального игрока. Без прогнозирования и переигровки шагов локального персонажа локальный игрок постоянно будет замечать задержки, вызванные передачей данных по сети и работой алгоритма интерполяции. Однако за счет немедленного моделирования его действий локальный игрок никогда не будет чувствовать отставания, независимо от величины задержки.
- **Включение представления клиента в каждый пакет ввода, посылаемый на сервер.** Клиент должен добавлять в каждый пакет ввода идентификаторы кадров, между которыми он выполняет интерполяцию в данный момент, и процент выполнения интерполяции. Это даст серверу точную информацию о состоянии игрового мира на стороне клиента.
- **Хранение на сервере координат всех объектов, связанных с несколькими последними кадрами.** Когда сервер получает от клиента пакет ввода с командой на выстрел, он должен найти в своем хранилище два кадра, между которыми клиент выполнял интерполяцию в момент выстрела, и использовать процент выполнения интерполяции из пакета для возврата всех соответствующих объектов к состоянию, в котором они находились в момент выстрела. Затем построить прямую линию от позиции персонажа, чтобы определить точку попадания.

Возврат на стороне сервера гарантирует, что в случае точного прицеливания игрок попадет в цель на сервере. Это даст выстрелившему игроку чувство глубокого удовлетворения. Однако метод не лишен недостатков. Так как возврат на сервере происходит на величину задержки пересылки пакета между сервером и клиентом, игрок-жертва может испытать разочарование. Например, игрок А может думать,

что находится в безопасности, так как убежал от игрока Б и скрылся за углом. Но, если у игрока Б особенно ненадежное подключение к сети, он может видеть мир, отстающий от мира игрока А на 300 мс. То есть на его компьютере игрок А мог еще не скрыться за углом. Если он прицелится и выстрелит, сервер выполнит возврат и сообщит игроку А, что тот застрелен, даже при том, что игрок уверен, что успел нырнуть за угол. Как обычно в разработке игр, все имеет свою цену. Поэтому используйте описанные приемы, только если они подходят к специфике вашей игры.

В заключение

Задержки могут нарушить непротиворечивость восприятия многопользовательской игры, однако существует несколько стратегий, помогающих редуцировать проблему. Фактически применение одного или нескольких из этих приемов стало обязательным условием в современных играх.

Интерполяция на стороне клиента с локальным фильтром восприятия сглаживает процесс изменения состояния, не допуская его скачкообразного отображения. Период интерполяции, равный периоду между изменениями состояния, сохраняет целостность восприятия изменений игроком, но усиливает ощущение задержки. Этот прием никогда не отобразит на стороне клиента ошибочное состояние.

Прием прогнозирования на стороне клиента маскирует задержки и синхронизирует состояние игры на стороне клиента с истинным состоянием на сервере, используя экстраполяцию вместо интерполяции. Достигнув клиента, обновления устаревают по меньшей мере на 1/2 RTT, поэтому клиент может приблизить полученное состояние к истинному, смоделировав ход игры еще на 1/2 RTT.

Расчет траектории движения помогает на основе последнего известного состояния объекта экстраполировать его состояние в будущем. Этот алгоритм оптимистично предполагает, что удаленные игроки не изменят свой ввод. Однако ввод изменяется довольно часто, поэтому не редка ситуация, когда сервер посылает клиенту состояние, отличающееся от его экстраполяции. Когда это происходит, клиент имеет множество способов встроить вновь полученное истинное состояние в свою модель и скорректировать картинку, отображаемую на экране.

Прогнозирование и переигровка хода помогают клиенту немедленно моделировать результаты, опираясь на ввод локального игрока. При получении состояния локального игрока с сервера клиент моделирует это состояние на 1/2 RTT вперед, переигрывая любые ходы, которые еще не были обработаны сервером. Часто это приводит полученное от сервера состояние в соответствие с моделью на стороне клиента. В случае неожиданного события на стороне сервера, например столкновения с другими игроками, клиент сможет плавно внедрить это состояние, скорректировав локальную модель.

Для полной компенсации задержки при обработке случаев применения оружия мгновенного действия игры могут использовать прием возврата на стороне сервера. Сервер сохранят позиции объектов для нескольких последних кадров и фактически возвращает состояние к моменту на стороне клиента, когда был произведен выстрел. Это придает стреляющему ощущение собственной меткости, но может

вызывать непонимание у пострадавших игроков, получающих ранения уже после того, как они оказались в безопасности.

Вопросы для повторения

1. Что означает выражение «клиент как простой терминал»? В чем основная выгода для игр от использования таких клиентов?
2. В чем основное преимущество интерполяции на стороне клиента? А основной недостаток?
3. Как сильно отстает от истинного состояние игры на клиенте, действующем как простой терминал?
4. В чем разница между консервативным и оптимистичным алгоритмами? Приведите примеры обоих.
5. Когда оправданно применение расчета траектории движения? Как с его помощью прогнозируется местоположение объектов?
6. Приведите три способа корректировки предсказанного состояния, если оно оказывается неправильным.
7. Опишите систему, помогающую локальному игроку не замечать задержек в отношении его собственных действий.
8. Какую проблему решает прием возврата на стороне сервера? В чем его главное достоинство? А основной недостаток?
9. Добавьте в «Robo Cat Action» поддержку клубков, способных мгновенно поражать противника, и реализуйте возврат на стороне сервера для определения поражения цели.

Для дополнительного чтения

Aldridge, David. (2011, март). «Shot You First: Networking the Gameplay of HALO: REACH». Доступно по адресу: *http://www.gdcvault.com/play/1014345/I-Shot-You-First-Networking*. Проверено 28 января 2016.

Bernier, Yahn W. (2001) «Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization». Доступно по адресу: *https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization*. Проверено 28 января 2016.

Caldwell, Nick. (2000, февраль) «Defeating Lag with Cubic Splines». Доступно по адресу: *http://www.gamedev.net/page/resources/_/technical/multiplayer-and-network-programming/defeating-lag-with-cubic-splines-r914*. Проверено 28 января 2016.

Carmack, J. (1996, август). «Here is the New Plan». Доступно по адресу: *http://fabiensanglard.net/quakeSource/johnc-log.aug.htm*. Проверено 28 января 2016.

Sweeney, Tim. «Unreal Networking Architecture». Доступно по адресу: *https://udn.epicgames.com/Three/NetworkingOverview.html*. Проверено 28 января 2016.

9 Масштабируемость

Необходимость масштабирования игры влечет за собой необходимость решения множества новых задач, отсутствующих в играх небольшого масштаба. В этой главе рассматриваются некоторые проблемы, возникающие с увеличением масштаба игры, и примеры их решения.

Область видимости и релевантность объектов

В главе 1, в обсуждении модели «Tribes», упоминалось понятие *области видимости*, или *релевантности*, объекта. В этом контексте считается, что объект находится в области видимости, или является релевантным для конкретного клиента, и этот клиент должен информироваться обо всех изменениях в объекте. Для небольших игр допустимо считать видимыми, или релевантными, все объекты для всех клиентов в игре. Это естественно означает, что все изменения в объектах на стороне сервера будут пересылаться всем клиентам. Однако такой подход не годится для большой игры — большой не только в смысле объема трафика, но и в смысле времени обработки на стороне клиента. В игре с 64 игроками знание о действиях игроков, чьи персонажи находятся на расстоянии нескольких километров, неактуально. В этом случае рассылка информации о таком далеком игроке всем участникам может оказаться напрасной тратой ресурсов. Соответственно, имеет смысл не посылать клиенту А информацию об объекте Ж, если сервер посчитает, что клиент слишком далек от объекта. Дополнительная выгода от уменьшения объема информации, посылаемой каждому клиенту, — сокращение возможностей для «читерства», о чем подробнее рассказывается в главе 10 «Безопасность».

Однако определение релевантности редко оказывается простой задачей. Представьте, что некий объект фактически является персонажем, представляющим другого игрока. Допустим, что в рассматриваемой игре есть табло, отображающее уровень здоровья каждого персонажа, независимо от расстояния до него. В этом сценарии уровень здоровья всегда является релевантным, даже если другая инфор-

мация о персонаже релевантна не всегда. То есть сервер всегда должен присылать уровни здоровья других игроков, даже если прочая информация о них может быть нерелевантной. Кроме того, разные объекты могут обновляться с разной частотой, в зависимости от их приоритетов, что добавляет нам проблем. Ради простоты в этом разделе будут рассматриваться вопросы определения релевантности объектов только целиком (а не отдельных их свойств). Но помните, что в коммерческих играх релевантность обычно определяется более сложным образом.

Вернемся к примеру игры с 64 игроками. Идею считать удаленные объекты нерелевантными называют *пространственным подходом*. Несмотря на то что простая проверка расстояния является очень быстрым способом определения релевантности, обычно она недостаточно надежна, чтобы служить единственным механизмом определения релевантности. Чтобы понять причины, рассмотрим пример игрока в шутере от первого лица. Допустим, что в первоначальном проекте игры предусматривалась поддержка только двух видов оружия: пистолета и штурмовой винтовки. На основе положений проекта программист решил определять видимость объектов, исходя из дальнобойности оружия, — все, что недосягаемо для штурмовой винтовки, считается находящимся вне пределов видимости. В процессе тестирования объем сетевого трафика находился в допустимых пределах. Однако позднее проектировщики решили добавить снайперскую винтовку, дальность стрельбы которой в два раза превышает дальность стрельбы штурмовой винтовки. В результате число релевантных объектов существенно увеличилось.

Имеются также другие проблемы, обусловленные использованием расстояния как единственного признака определения релевантности объектов. Игрок в середине игрового мира с большей вероятностью окажется в досягаемости других объектов, чем игрок, находящийся на периферии. Кроме того, использование одного только расстояния присваивает одинаковый вес объектам впереди и позади игрока, что не совсем верно. При простом пространственном подходе к определению релевантности объектов релевантными будут считаться все объекты вокруг игрока, даже те, которые отделены от него стеной. Эта ситуация изображена на рис. 9.1.

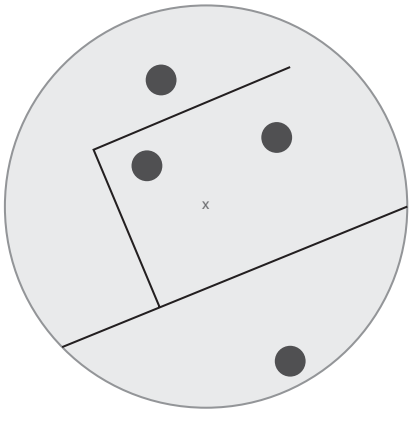


Рис. 9.1. Релевантность окружающих объектов по отношению к игроку, отмеченному крестиком, должна быть разной

В оставшейся части раздела мы сосредоточимся на подходах более сложных, чем простая проверка расстояния. Многие из этих приемов также часто используются для определения *прямой видимости* с целью оптимизации отображения на ранних этапах, что позволяет исключить из обработки невидимые объекты. Однако, учитывая природу задержек в сетевых играх, в методику определения прямой видимости обычно требуется внести дополнительные изменения, чтобы ее можно было использовать для определения релевантности объектов.

Статические зоны

Один из приемов, помогающих уменьшить число релевантных объектов, заключается в том, чтобы разбить мир на несколько *статических зон*. При этом релевантными будут считаться только объекты, находящиеся в одной статической зоне с игроком. Этот прием часто используется в играх с общими мирами, таких как MMORPG. Например, одной из подобных зон может быть город, где игроки встречаются для обмена товарами, другой зоной может быть лес, где игроки сражаются с монстрами. В этом случае не имеет смысла посылать игрокам, находящимся в лесу, информацию об игроках, находящихся в городе.

Существуют два разных способа обработки переходов через границы зон. Один из них заключается в отображении экрана загрузки зоны, куда осуществляется переход. Это дает клиенту достаточно времени, чтобы принять информацию обо всех объектах в новой зоне. Для более гладкого перехода может быть желательно организовать плавное проявление/растворение объектов в зависимости от изменения их релевантности. Если при переходе между зонами ландшафт не должен изменяться, его можно просто хранить на стороне клиента, чтобы зона за спиной игрока не исчезала полностью. Однако имейте в виду, что сохранение ландшафта на стороне клиента сопряжено с некоторыми проблемами безопасности. Одно из решений предполагает шифрование данных, о чем подробнее рассказывается в главе 10 «Безопасность».

Недостатком статических зон является то, что они проектируются на основе предположения о равномерном распределении игроков между зонами. Однако это требование очень сложно удовлетворить в массовых играх MMORPG. В местах встреч, таких как города, всегда будет более высокая концентрация игроков, чем в отдаленных зонах. Эта ситуация может осложняться наличием игровых событий, заставляющих игроков собираться в одном месте, например, для боя с особенно сильным противником Высокая концентрация игроков в одной зоне может отрицательно сказываться на производительности игры для всех игроков в зоне.

Решения, препятствующие переполнению зоны, могут различаться в разных играх. В MMORPG «Asheron's Call», например, если игрок попытается войти в зону, где слишком много игроков, он будет телепортирован в соседнюю зону. Даже при том, что это не идеальное решение, оно прекрасно подходит для игр, не способных работать с большим количеством игроков в одной зоне. Другие игры могут разбивать зоны на множество экземпляров, о чем подробно рассказывается далее в этой главе.

Несмотря на надежность, показанную в играх с общим миром, статические зоны обычно не используются в сюжетных играх по двум причинам. Во-первых, в большинстве сюжетных игр сражения происходят на значительно меньшей территории, чем в массовых играх MMO, даже при том, что имеются некоторые известные исключения, такие как «PlanetSide». Во-вторых, что, пожалуй, еще более важно, темп большинства сюжетных игр предполагает недопустимость задержек при пересечении границ между зонами.

Использование пирамиды видимости

Как вы помните, в трехмерных играх *область поля зрения* имеет вид усеченной пирамиды, представляющей область мира, которая затем проецируется на двумерную плоскость для отображения на экране. Область поля зрения описывается горизонтальным углом поля зрения, отношением ширины к высоте и расстоянием между ближней и дальней плоскостями отсечения. Когда применяется преобразование проекции, объекты, находящиеся внутри поля зрения полностью или частично, считаются видимыми, тогда как другие объекты — невидимыми.

Область поля зрения обычно формируется путем отсечения по *пирамиде видимости*. Например, если объект находится за пределами пирамиды, он считается невидимым, и игра не должна расходовать время на пересылку треугольников объекта в вершинный шейдер. Один из способов реализовать отсечение по пирамиде видимости — представить область поля зрения в виде пространства, ограниченного шестью плоскостями — сторонами усеченной пирамиды. Тогда упрощенное представление объекта, такого как сфера, можно проверить на местоположение относительно плоскостей, чтобы определить, находится ли объект внутри пирамиды. Подробное математическое описание этого метода можно найти в книге Кристера Эриксона, ссылка на которую приводится в конце главы.

Отсечение по пирамиде видимости, определяющей область поля зрения, имеет большое значение, но, принимая во внимание задержки, использование одного только этого приема для определения видимости объекта в сетевой игре влечет за собой некоторые проблемы. Например, если использовать только область поля зрения, объекты за спиной игрока должны считаться невидимыми. Это обстоятельство перерастет в проблему, если допустить, что игрок может мгновенно повернуться на 180 градусов. Определенно потребуется некоторое время, пока команда на поворот достигнет сервера, а сервер, соответственно, вернет состояния объектов, оказавшихся в поле зрения. Возникшая в результате задержка может оказаться недопустимой, особенно если объект за спиной — это персонаж противоборствующего игрока. Кроме того, этот подход все еще игнорирует наличие стен. Описанная проблема изображена на рис. 9.2.

Одно из решений — использовать оба подхода: на основе пирамиды видимости и на основе определения расстояния. В частности, проверку на расположение ближе дальней плоскости отсечения можно объединить с проверкой на попадание в пирамиду видимости. В этом случае все объекты, находящиеся к игроку ближе выбранного расстояния или попадающие в область поля зрения, можно было бы считать видимыми, а все остальные — невидимыми. Тогда при быстром

повороте удаленные объекты по-прежнему будут либо попадать в поле зрения, либо выпадать из него, и стены все так же будут игнорироваться, но видимость ближних объектов изменяться не будет. Иллюстрация этого решения изображена на рис. 9.3.

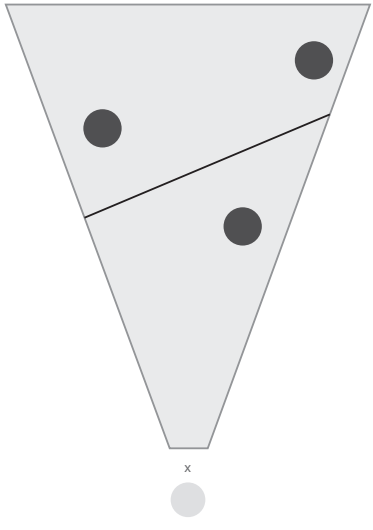


Рис. 9.2. Объект вне поля зрения непосредственно за спиной игрока (обозначенного крестиком)

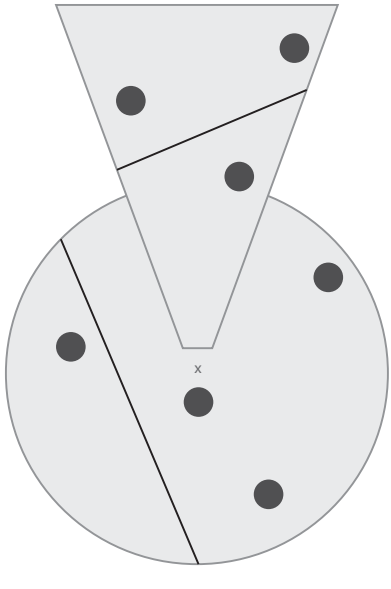


Рис. 9.3. Объединение пирамиды видимости с меньшим радиусом для определения релевантности объектов

Другие приемы определения видимости

Представьте сетевую игру, имитирующую автогонки по замкнутой трассе, проходящей через город. Любому, кто ездил на автомобиле, очевидно, что протяженность видимого участка дороги может существенно изменяться. На прямых плоских участках дорога видна далеко. Однако в поворотах протяженность видимого участка значительно сокращается. Точно так же она сокращается при движении с горы или в гору. Идею определения протяженности видимого участка можно непосредственно внедрить в сетевую игру автогонок. В частности, если серверу известно местоположение автомобиля игрока, он сможет определить, насколько далеко видна трасса с этой точки. Видимая область практически наверняка будет намного меньше, чем область, ограниченная пирамидой видимости, что в идеале должно привести к уменьшению числа объектов в поле зрения.

Описанная идея привела к появлению приема определения *потенциально видимого набора* объектов (Potentially Visible Set, PVS). Прием PVS используется для получения ответа на вопрос: «Какие объекты потенциально видимы с той или иной точки в игровом мире?» Несмотря на сходство со статическими зонами, области в PVS обычно намного меньше, чем отдельные зоны. Статическая зона может включать город с множеством зданий, а область PVS — отдельную комнату внутри здания. Кроме того, в решении на основе статических зон релевантными считаются только объекты, находящиеся в той же статической зоне. Этим оно отличается от решения на основе PVS, в котором релевантными будут считаться также объекты, находящиеся в соседних областях, считающихся потенциально видимыми.

Типичная реализация PVS делит мир на множество выпуклых многоугольников (или, при необходимости, на множество выпуклых многогранников). Для каждого выпуклого многоугольника заранее вычисляется множество других, потенциально видимых выпуклых многоугольников. Во время выполнения сервер определяет, в каком многоугольнике находится игрок, и на основе предварительно сгенерированных многоугольников определяет множество всех потенциально видимых объектов. Затем эти объекты могут помечаться как релевантные для данного игрока.

Рисунок 9.4 иллюстрирует результат работы PVS в гипотетической игре, имитирующей автогонки. Местоположение игрока отмечено на рисунке крестиком, закрашенная часть представляет потенциально видимую область. В фактической реализации можно было бы немного расширить потенциально видимую область в обоих направлениях. В этом случае объекты, находящиеся на небольшом удалении от видимой области, также могли бы отмечаться как видимые. В игре, имитирующей автогонки, где автомобили мчатся с бешеной скоростью, особенно важно учесть задержку во взаимодействиях с сервером и обеспечить своевременное обновление видимых объектов.

Система PVS хорошо подходит для шутеров от первого лица, таких как «Doom» или «Quake», где поединки происходят в ограниченном пространстве. Для игр подобного рода также можно порекомендовать использовать родственную методику, называемую *порталами*. В системе с порталами каждая комната считается отдельной областью, а каждая дверь или окно — порталом. Пирамиду видимости, образуемую порталом, можно объединить с областью поля зрения и тем самым

значительно уменьшить число релевантных объектов. Такая система требует большего объема вычислений, чем PVS, но если в игре уже используются порталы для уменьшения нагрузки на клиентов, дополнить ее определением видимости объектов на стороне сервера будет не очень сложно.

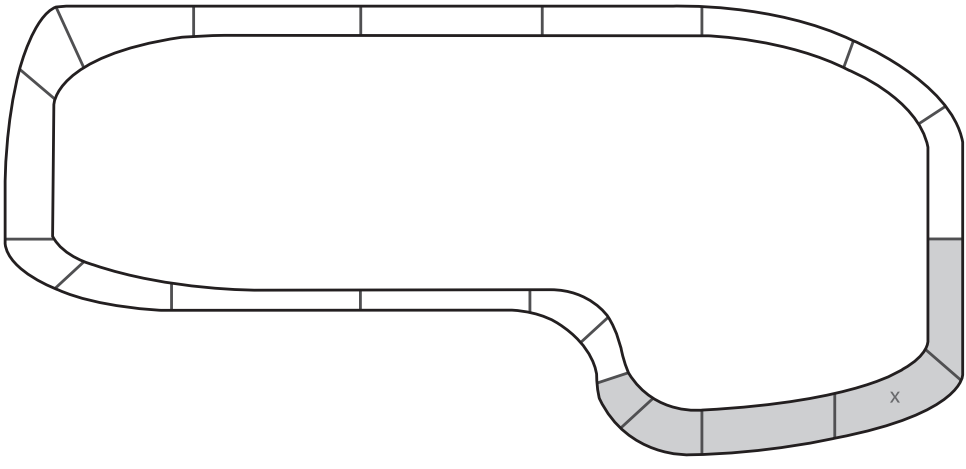


Рис. 9.4. Пример использования PVS в игре, имитирующей автогонки

Аналогично, в некоторых играх с успехом можно использовать приемы иерархического отсечения, такие как двоичное разбиение пространства (Binary Space Partition, BSP), *квадродерево* (quadtree) или *октодерево* (octree). Каждый из этих приемов разбивает объекты в игровом мире на древовидные структуры. Детальное обсуждение подобных приемов можно найти в уже упоминавшейся книге Кристера Эриксона. Имейте в виду, что использование любого из таких продвинутых приемов определения видимости объектов существенно увеличивает время, необходимое для расчетов. Это становится очевидным, если учесть, что процесс определения видимости должен быть повторен для каждого клиента, подключенного к серверу. Если только перед вами не стоит задача обеспечить репликацию всех объектов, использование этих иерархических систем определения видимости объектов едва ли можно признать необходимым. Для большинства сюжетных игр вполне достаточно хорошо отлаженной системы PVS, а во многих играх может не требоваться и уровень детальности, обеспечиваемый системой PVS.

Релевантность невидимых объектов

Важно отметить, что видимость конкретного объекта не всегда напрямую определяет его релевантность. Возьмем для примера игру, в которой игроки могут бросать гранаты. Если граната взрывается в соседней комнате, важно, чтобы объект, представляющий гранату, был передан всем клиентам, находящимся поблизости, даже если граната им не видна. Это объясняется тем, что игрок ожидает услышать звук взрыва гранаты, даже если граната была невидима в момент взрыва.

Одним из возможных решений этой проблемы является обработка гранаты, отличающаяся от обработки прочих объектов. Например, репликацию гранаты можно осуществлять, опираясь на расстояние до игрока, а не на ее видимость. Другой вариант — передавать эффект взрыва клиентам, для которых граната не является релевантным объектом, через RPC. Этот второй подход позволяет уменьшить объем данных, пересылаемых клиентам, которые должны воспроизвести звук взрыва (и, возможно, визуальный эффект), за счет исключения процесса передачи фактического объекта — гранаты. При этом информация о взрыве гранаты может быть передана клиентам, которые в действительности не могут услышать его, но если это особый случай и в игре не участвует огромное число объектов, данный прием не должен значительно увеличить объем трафика.

Если в игре широко используются аудиоэффекты, на сервере можно даже вычислять информацию о звуках столкновения с преградой, чтобы определить их релевантность. Но для большей реалистичности такие вычисления вообще следует производить на клиенте — едва ли вы встретите коммерческую игру, способную вычислить релевантность аудиоэффекта на сервере с такой степенью точности. Решения на основе RPC или определения радиуса вполне достаточно для большинства игр.

Сегментирование серверной среды выполнения

Идея *сегментирования* серверной среды выполнения заключается в том, чтобы настроить обработку игровой модели одновременно на нескольких серверах. Большинство сюжетных игр по своей природе используют этот подход, потому что каждая активная игра имеет ограниченное число активных игроков — обычно от 8 до 16. Число игроков в игре в значительной степени определяется замыслом игры, но такая организация имеет свои бесспорные технические выгоды. Идея состоит в том, что при наличии нескольких отдельных серверов нагрузка на какой-то отдельный сервер не должна оказаться неподъемной.

Примерами игр, в которых используется прием сегментирования, служат «Call of Duty», «League of Legends» и «Battlefield». Поскольку каждый сервер выполняет свой экземпляр игры, действия игроков в разных играх никак не мешают друг другу. Однако многие из этих игр сохраняют статистическую информацию, опыт, уровни и другие сведения в общей базе данных. Это означает, что каждый серверный процесс имеет доступ к некоторой базе данных, являющейся частью игровой службы — понятия, более подробно рассматриваемого в главе 12 «Игровые службы».

Прием сегментирования серверной среды выполнения часто используется, когда один компьютер в состоянии выполнять сразу несколько серверных процессов. Во многих играх с большим бюджетом разработчикам предоставляются компьютеры в вычислительных центрах, способные выполнять несколько серверных процессов. В таких играх часть игровой архитектуры должна обслуживать распределение процессов на каждом компьютере. Одно из решений состоит в том, чтобы реализовать мастер-процесс, решающий, когда должен быть запущен новый игровой процесс

и на каком компьютере. Когда игра завершается, серверный процесс может перед выходом записать любые необходимые данные в базу. Затем, когда игроки решат начать новый сеанс, мастер-процесс определит, какой из компьютеров нагружен меньше всего, и запустит на нем новый серверный процесс. Для запуска серверов также можно использовать облачный хостинг (cloud hosting), о чем рассказывается в главе 13 «Облачный хостинг для выделенных серверов».

Сегментирование серверной среды выполнения также используется как расширение приема организации статических зон, применяемого в массовых играх. В частности, каждая статическая зона или коллекция статических зон может обслуживаться отдельным серверным процессом. Например, популярная игра «World of Warcraft» включает множество континентов. Каждый континент обслуживается отдельным серверным процессом. Когда игрок перемещается с одного континента на другой, перед ним выводится экран загрузки, который отображается, пока состояние персонажа не будет передано серверному процессу, обслуживающему новый континент. Каждый континент состоит из нескольких статических зон. В отличие от континентов пересечение границ статических зон происходит незаметно, потому что все зоны на континенте обслуживаются одним и тем же серверным процессом. На рис. 9.5 показано, как могла бы выглядеть конфигурация подобной гипотетической игры MMORPG. Каждый шестиугольник представляет статическую зону, а пунктирная линия — путь игрока между двумя континентами.

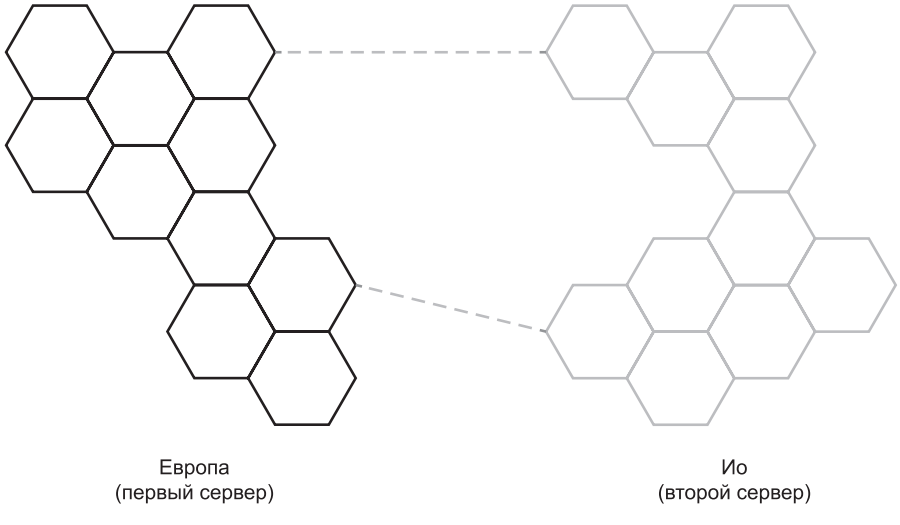


Рис. 9.5. Сегментирование серверной среды выполнения в гипотетической игре MMORPG для обслуживания отдельных континентов, но не статических зон

Как и в случае со статическими зонами, сегментирование серверной среды выполнения оправданно, только если игроки равномерно распределены между серверами. Если какому-то одному серверу придется обслуживать слишком большое число игроков, он может столкнуться с проблемой нехватки производительности. Эта

проблема не характерна для игр с фиксированным числом игроков, но она может проявляться в массовых играх. Способ решения проблемы зависит от характера игры. Некоторые игры просто ограничивают число игроков и вынуждают их ждать своей очереди, пока сервер не освободится. Игра «Eve Online» замедляет темп работы. Такой режим называют *режимом растяжения времени* (time dilation). Он позволяет серверу обслужить всех игроков, которых в другой ситуации обслужить было бы невозможно.

Клонирование

Прием *клонирования* (instancing) подразумевает наличие одной общей игры, поддерживающей несколько отдельных экземпляров. Этот прием обычно используется в играх с общим игровым миром, где все персонажи находятся на одном и том же сервере, но могут принадлежать разным экземплярам игры. Например, многие массовые игры MMORPG используют клонирование с целью организации системы подземелий для фиксированного числа игроков. При таком подходе группы игроков могут действовать по заранее подготовленным сценариям, свободным от вмешательства других игроков. В большинстве игр, реализующих такого рода клонирование, имеются порталы или похожие конструкции, помогающие игрокам перемещаться из общей зоны в клоны.

Иногда клонирование используется для решения проблемы переполнения зон. Например, игра «Star Wars: The Old Republic» устанавливает ограничение на число игроков, которые могут одновременно находиться в некоторой зоне. Если число игроков становится слишком большим, из оригинального экземпляра зоны клонируется второй экземпляр. Это порождает некоторые сложности для игроков. Если два игрока решили встретиться в какой-то зоне, они могут оказаться в двух разных экземплярах (клонах) этой зоны. В игре «The Old Republic» на этот случай предусмотрено решение, позволяющее игроку перемещаться в экземпляр, принадлежащий группе.

С точки зрения проектирования прием клонирования больше подходит для игр с одним или с небольшим числом игроков, имеющих персонажей, связанных с общим миром. Некоторые игры используют клонирование как способ организации зон на пути поисков. Однако этот прием имеет свой недостаток — клонирование снижает ощущение единства мира.

С точки зрения производительности, пока стоимость оборота экземпляров находится под контролем, клонирование может быть выгодным. Прием клонирования способен гарантировать релевантность не более X игроков в каждый момент времени, особенно если зоны способны порождать отдельные клоны. Более того, клонирование можно даже совместить с сегментированием серверной среды выполнения, чтобы дополнительно снизить нагрузку на определенные серверные процессы. Так как вход в экземпляр практически всегда сопряжен с выводом экрана загрузки на клиенте, нет причин, которые препятствовали бы перемещению клиента на другой сервер, как это реализовано при перемещении между континентами в «World of Warcraft», обслуживаемыми разными серверами.

Система приоритетов и частота обновления

В некоторых играх производительность сервера — не самое узкое место. Основная проблема заключается в объеме данных, которые приходится передавать клиентам по сети. Эта проблема особенно остро стоит в играх для мобильных устройств, которые должны учитывать особенности работы мобильной сети. В главе 5 обсуждались некоторые пути решения подобной проблемы, такие как частичная репликация объектов. Однако если на этапе тестирования выяснится, что игра предъявляет слишком высокие требования к полосе пропускания, следует рассмотреть некоторые дополнительные приемы.

Одно из решений заключается в том, чтобы присвоить разным объектам разные приоритеты. Объекты с высшим приоритетом могут передаваться в первую очередь, а объекты с низшим приоритетом — только в отсутствие высокоприоритетных объектов. Этот прием можно рассматривать как способ нормирования полосы пропускания: когда имеется ограниченная полоса пропускания, она будет использоваться только для передачи наиболее важных объектов.

При использовании системы приоритетов важно иногда давать возможность передавать низкоприоритетные объекты, иначе эти объекты никогда не будут обновляться на клиентах. Для этого можно определить разную частоту репликации разных объектов. Например, важные объекты могут обновляться пару раз в секунду, менее важные — каждую пару секунд. Частоту обновления можно совместить с системой приоритетов и на их основе вычислять некоторый динамический уровень приоритета, когда приоритет низкоприоритетных объектов становится тем выше, чем больше времени прошло с момента последнего его обновления.

Аналогичную систему приоритетов можно применить к удаленным вызовам процедур. Если определенные вызовы оказываются не соответствующими состоянию игры, их можно исключать из передачи в отсутствие достаточной пропускной способности. Это напоминает надежную и ненадежную передачу пакетов, описывавшуюся в главе 2.

В заключение

Сокращение объема данных, передаваемых каждому клиенту, является важной задачей, которая должна решаться по мере увеличения масштаба сетевой игры. Одно из решений состоит в уменьшении общего числа объектов в области видимости конкретного клиента. Самый простой способ — считать особенно далекие от клиента объекты не попадающими в его область видимости, однако такой подход, не учитывающий индивидуальных особенностей игры, не во всех случаях дает положительные результаты. Другой подход, особенно популярный в играх с общим миром, — деление игрового мира на статические зоны. В этом случае только игроки в одной зоне будут считаться релевантными по отношению друг к другу.

Для снижения числа релевантных объектов можно также задействовать приемы определения видимости. Даже в том случае, когда использовать прием определения попадания в пирамиду видимости не рекомендуется, в комбинации с приемом определения релевантности по расстоянию он может давать неплохие результаты.

В играх с четким секционированием игрового пространства, таких как шутеры в ограниченном пространстве или автогонки, с успехом можно использовать PVS. Прием PVS позволяет определить, какие области видны из конкретной точки в уровне. Существуют и другие методики определения видимости, такие как порталы, способные давать дополнительные выгоды в отдельных случаях. Наконец, в некоторых ситуациях видимость объекта не может служить критерием его релевантности, как, например, в ситуации со взрывом гранаты.

Сегментирование серверной среды выполнения также можно использовать для уменьшения нагрузки на каждый отдельный сервер. Этот прием подходит и для сюжетных игр с фиксированным числом игроков, и для игр с огромным общим миром, где отдельные зоны могут обслуживаться разными серверными процессами. Аналогично, методика клонирования позволяет создавать самостоятельные копии отдельных областей общего мира, лучше контролируемые с точки зрения производительности и конструирования.

Существуют также приемы, не связанные с релевантностью объектов, которые можно использовать для ограничения трафика в сетевых играх. Один из них заключается в назначении приоритетов разным объектам и удаленных вызовах процедур, чтобы наиболее важная информация передавалась в первую очередь. Другое решение основано на уменьшении частоты репликации обновлений для всех объектов, кроме наиболее важных.

Вопросы для повторения

1. Перечислите недостатки определения релевантности объекта только по расстоянию до него.
2. Что такое «статические зоны» и какие потенциальные выгоды они несут?
3. Как можно использовать пирамиду видимости для выбора объектов? Что получится, если релевантность объектов определять только с применением пирамиды видимости?
4. Что такое «потенциально видимый набор» и чем этот прием отличается от статических зон?
5. Если игра с общим миром страдает от переполнения зон, какие решения этой проблемы вы могли бы предложить?
6. Перечислите несколько решений, отличных от уменьшения числа релевантных объектов, для снижения требований сетевой игры к пропускной способности.

Для дополнительного чтения

Ericson, Christer. «Real-Time Collision Detection». San Francisco: Morgan Kaufmann, 2004.

Fannar, Halldor. «The Server Technology of EVE Online: How to Cope With 300,000 Players on One Server». Презентация на конференции разработчиков игр в г. Остин (Техас, США), 2008.

10 Безопасность

Азартные игроки всегда искали способы сжульничать. С ростом популярности сетевых игр защита от несанкционированного доступа к игре стала непременным условием обеспечения безопасности игровой среды. В этой главе рассматриваются некоторые распространенные уязвимости и способы борьбы с ними.

Перехват пакетов

В обычном режиме функционирования сети пакеты проходят через несколько разных компьютеров, встречающихся на пути от отправителя к получателю. Промежуточные маршрутизаторы должны считать заголовки пакетов, чтобы определить, куда их направлять. И как уже говорилось в главе 2, иногда механизмы преобразования сетевых адресов подменяют адреса в пакетах. Однако, учитывая открытую природу передаваемых данных, ничто не мешает любым компьютерам, встречающимся на пути, исследовать данные в пакетах.

Иногда такое исследование может выполняться в ходе стандартных сетевых операций. Например, некоторые маршрутизаторы проводят глубокое исследование содержимого пакетов с целью обеспечить определенный *уровень качества обслуживания* (quality of service) и в первую очередь пропустить высокоприоритетные пакеты. Система поддержки качества обслуживания анализирует содержимое пакетов, и если пакет определяется ею как часть потока передачи файла между узлами, ему присваивается более низкий приоритет в отличие от пакета с данными голосового потока, передаваемого по протоколу VoIP (voice-over IP — голос через IP).

Но существует форма исследования пакетов, которая выполняется не всегда с благими намерениями. Под *перехватом пакетов* (packet sniffing) обычно подразумевается операция чтения данных из пакета с целью, не связанной с обеспечением нормального функционирования сети. Делаться это может с самыми разными целями, включая попытку перехватить имя пользователя и пароль или мошенни-

ческим путем получить какие-либо дополнительные преимущества в сетевой игре. В оставшейся части этого раздела основное внимание будет уделено конкретным приемам перехвата пакетов в сетевых играх.

Атака «незаконный посредник»

Атака вида «*незаконный посредник*» (man-in-the-middle) возможна, если на полпути между отправителем и получателем находится компьютер, способный перехватывать пакеты втайне от отправителя и получателя. Эта ситуация изображена на рис. 10.1. Следует отметить, что существует множество способов организации атак этого вида. Любая информация, передаваемая компьютером с незащищенным или общедоступным подключением к сети Wi-Fi, может быть перехвачена другим компьютером в этой же сети. (Именно поэтому рекомендуется использовать шифрованный канал VPN при работе с общедоступной сетью Wi-Fi где-нибудь в местном кафе.) В проводной сети пакеты могут перехватываться на машине-шлюзе либо вредоносным программным обеспечением, либо не в меру любопытным системным администратором. Кроме того, если по какой-то причине правоохранительные органы нацелились на вашу игру, они также могут способствовать установке программного обеспечения у поставщика услуг Интернета, чтобы получить доступ к данным.

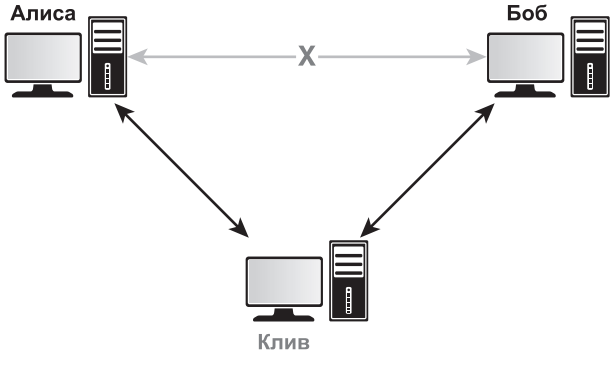


Рис. 10.1. Атака вида «незаконный посредник», в ходе которой Клив читает переписку между Алисой и Бобом

Технически игрок может преднамеренно установить специальное программное обеспечение для перехвата трафика игры. Это проблематично сделать на закрытой платформе, такой как игровая консоль, но следует помнить, что игрок с персональным компьютером всегда будет иметь полный доступ ко всем данным, передаваемым по сети. Поэтому в дальнейшем обсуждении атак вида «незаконный посредник» будет предполагаться, что «посредник» — это третья сторона, не известная ни отправителю, ни получателю.

Обычно противостояние атакам «незаконный посредник» осуществляется путем шифрования всех передаваемых данных. В сетевой игре, прежде чем приступать

к внедрению какой-либо системы шифрования, следует оценить, содержит ли игра уязвимые данные, требующие шифрования. Если игра выполняет какие-нибудь микротранзакции, когда игрок покупает какие-либо игровые элементы, любые данные, связанные с покупками, обязательно должны шифроваться. Если игра сохраняет или даже просто обрабатывает информацию о кредитных картах, юридически обязательным требованием может стать соблюдение стандарта защиты информации в индустрии платежных карт (Payment Card Industry Data Security Standard, PCI DSS). Однако даже если в игре не совершается никаких покупок, любая игра, где игрок авторизуется в своей учетной записи, такой как MOBA или MMO, должна шифровать данные, имеющие отношение к процессу авторизации. В обоих этих случаях третье лицо может иметь материальный стимул для перехвата информации о кредитной карте или данных авторизации. Поэтому очень важно, чтобы игра защищала ценные данные, принадлежащие игроку, от незаконного посредника.

С другой стороны, если единственные данные, которые игра передает по сети, — это состояние объектов (и другая подобная информация), обеспечение безопасности перестает быть настолько актуальным. Такие данные можно оставить незашифрованными, и это не повлечет за собой больших проблем. Однако шифрование данных все еще может пригодиться для предотвращения атак вида «перехват пакетов на целевом компьютере», о которых будет рассказано далее.

Если вы пришли к выводу, что игра посылает данные, которые должны быть защищены от постороннего, используйте систему шифрования, доказавшую свою надежность. В частности, для передачи секретной информации хорошо подходит система *шифрования с открытым ключом*. Представьте, что Алиса и Боб решили настроить обмен зашифрованными сообщениями друг с другом. Прежде чем начать общаться, они должны сгенерировать закрытые и открытые ключи. Закрытые ключи остаются в личной собственности того, кто их сгенерировал, — они никогда не должны передаваться кому бы то ни было. Когда Алиса и Боб свяжутся друг с другом в первый раз, они обменяются открытыми ключами. Затем, когда Алиса подготовит сообщение для Боба, она зашифрует его с помощью открытого ключа Боба. После этого сообщение можно будет расшифровать только с помощью закрытого ключа Боба. То есть сообщения от Алисы для Боба сможет прочитать только Боб, и никто другой, а Боб, со своей стороны, сможет послать сообщение Алисе, прочитать которое сможет только она. В этом заключается суть шифрования с открытым ключом (рис. 10.2).

В игре, имеющей сервер авторизации, клиентам передается открытый ключ сервера. Когда клиент пожелает авторизоваться, его имя и пароль шифруются с помощью открытого ключа сервера, и такой пакет с данными для авторизации можно расшифровать только при помощи закрытого ключа сервера, который, хотелось бы надеяться, известен только серверу!

Наиболее популярной системой шифрования с открытым ключом в настоящее время является система RSA, разработанная Ривестом, Шамиром и Адельманом в 1977 году. В RSA открытый ключ основан на очень большом *полупростом* (semiprime) числе, в том смысле, что оно является произведением двух простых чисел. Закрытый ключ основывается на разложении полупростого числа на простые

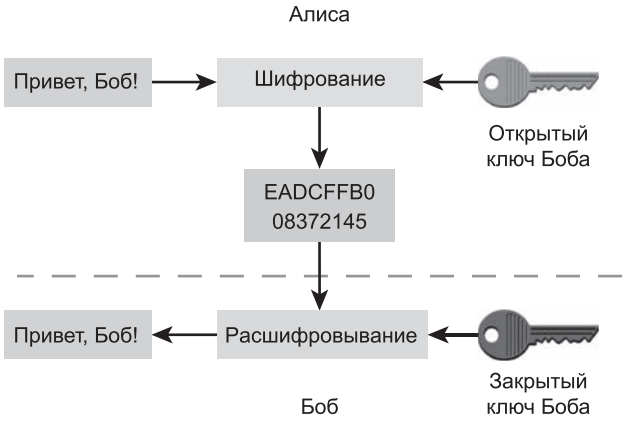


Рис. 10.2. Алиса и Боб переписываются, используя шифрование открытым ключом

множители. Надежность системы обусловлена отсутствием алгоритма разложения чисел на простые множители за полиномиальное время, а разложение 1024- или 2048-битных чисел методом простого перебора в настоящее время практически невозможно за обозримое время, даже на самом мощном в мире суперкомпьютере.

Учитывая широкую известность алгоритма RSA, было бы напрасной тратой времени пытаться реализовать его своими силами. Вместо этого мы рекомендуем воспользоваться открытой реализацией RSA, например, входящей в состав библиотеки OpenSSL. Так как OpenSSL выпускается под свободной лицензией, эту библиотеку можно использовать даже в коммерческих проектах.

ВЗЛОМ RSA

Существует несколько сценариев взлома RSA, и любой из них может оказаться фатальным в краткосрочной перспективе. Первый сценарий — создание достаточно мощного квантового компьютера. Существует квантовый алгоритм разложения числа на простые множители за *полиномиальное время* — алгоритм Шора (Shor's algorithm). Однако к моменту написания этих строк самый мощный из имеющихся в мире квантовых компьютеров сумел лишь разложить число 21 на множители 7 и 3, поэтому может понадобиться еще много лет, прежде чем появится квантовый компьютер, способный разлагать 1024-битные числа. Другой сценарий — открытие алгоритма целочисленного разложения за полиномиальное время для обычных компьютеров.

Фатальность этих сценариев объясняется широким использованием RSA и подобных алгоритмов для обеспечения безопасности взаимодействий в Интернете. Если система RSA будет взломана, многие ключи, используемые в HTTPS, SSH и подобных им протоколах, перестанут быть закрытыми. Большинство криптографов сходятся во мнении, что система RSA в конечном счете будет взломана, и именно поэтому в настоящее время активно ведутся исследования в области криптографии, целью которых является создание новых систем, взломать которые за полиномиальное время нельзя даже с помощью квантового компьютера.

Перехват пакетов на целевом компьютере

Итак, противостоять атакам вида «незаконный посредник» приходится только в играх, передающих уязвимые данные, но при этом любая сетевая игра уязвима для злонамеренного перехвата пакетов на целевом компьютере. В этом случае шифрование является лишь мерой устрашения, но не полноценной защитой. Дело в том, что игра, на какой бы платформе она ни работала, всегда может быть взломана, поэтому шифрование данных не помешает злоумышленнику узнать, как их расшифровать. Где-то в игре должен быть код, реализующий расшифровывание полученных данных. После выяснения схемы расшифровывания данные можно будет читать с легкостью, как если бы они не шифровались отправителем.

С другой стороны, реверс-инжиниринг кода, осуществляющего расшифровку, и поиск закрытого ключа на клиенте требуют некоторого времени. Поэтому шифрование остается одним из способов осложнить жизнь потенциальным мошенникам. Этому же способствует постоянная смена ключей шифрования и их местоположения в памяти. В этом случае расшифровка потребует от мошенника повторять процесс реверс-инжиниринга всякий раз с появлением обновленной версии игры. Аналогично, постоянная смена в игре формата и порядка следования пакетов послужит дополнительным заслоном на пути мошенников, полагающихся на устаревший формат пакетов, и заставит нечистых на руку игроков тратить время на изучение нового формата и разработку новой реализации мошенничества. То есть регулярное изменение формата пакетов и/или ключей шифрования сделает создание мошеннических сценариев для вашей игры более затратным. Хотелось бы надеяться, что большинство таких игроков, занимающихся мошенничеством, рано или поздно прекратит свои попытки. Но, как бы то ни было, следует признать, что вы никогда не сможете воспрепятствовать перехвату пакетов на целевом компьютере отдельными лицами.

Стоит рассмотреть цели, с которыми игроками осуществляется перехват пакетов на целевых компьютерах. Обычно игрок, сидящий за целевым компьютером, пытается использовать *информационный читинг*, то есть пытается получить сведения, которые он не должен знать. Типичный способ борьбы с мошенничеством в этом случае заключается в ограничении объема информации, передаваемой каждому узлу. В игре с архитектурой «клиент-сервер» сервер в состоянии ограничить количество данных, посылаемых клиенту. Например, представьте, что сетевая игра поддерживает невидимый режим перемещения игроков. Если сервер все еще посылает обновления для персонажей, действующих в невидимом режиме, игрок сможет выяснить их местоположение путем анализа пакетов. С другой стороны, если передача обновлений местоположения будет приостановлена, пока персонаж находится в невидимом режиме, клиент никак не сможет определить текущую позицию персонажа.

Мы рекомендуем всегда исходить из того, что любые данные, посылаемые каждому узлу, могут быть исследованы игроком, пытающимся смошенничать. То есть если игра будет передавать каждому узлу только самую важную и действительно необходимую информацию, это сузит поле для мошенничества. Такой подход проще реализовать в топологии «клиент-сервер», потому что в топологии «точка-точка»

требуется посылать каждому узлу все данные, имеющие отношение к игре в целом. В топологии «точка-точка» должны использоваться другие приемы противодействия мошенничеству.

Проверка ввода

В противоположность приемам перехвата пакетов, о которых рассказывалось выше, целью *проверки ввода* является предотвращение действий, которые игрок не имеет права выполнять. Этот метод борьбы с мошенничеством одинаково хорошо подходит как для игр с топологией «клиент-сервер», так и для игр с топологией «точка-точка». Суть методов на основе проверки ввода сводится к простой предпосылке, что игра никогда не должна слепо выполнять команды, полученные из сетевых пакетов. Команда должна быть сначала проверена на допустимость в данном месте и в данное время.

Например, представьте, что от игрока A получен пакет с командой «огонь». Получатель никогда не должен предполагать, что игрок A имеет безусловное право выстрелить из ружья. Он должен сначала убедиться, что игрок A имеет указанное оружие, что оружие заряжено и готово к стрельбе. Если какое-то из условий не выполнено, запрос на открытие огня должен быть отвергнут.

Далее следует убедиться, что команда для игрока A получена от клиента, который управляется игроком A. Как вы наверняка помните, обе версии игры «Robo Cat» в главе 6 выполняют такую проверку. В клиент-серверной версии игры каждому клиенту соответствует свой прокси-объект на сервере, поэтому когда на сервер поступает команда, он пропускает только команды, соответствующие узлу, связанному с данным прокси-объектом. В RTS-версии с топологией точка-точка каждая команда запускается определенным игроком. Когда принимается сетевой пакет с командой, он оказывается связан с определенным узлом. Когда подходит время выполнить команду, узлы отвергнут любые команды, относящиеся к персонажам, не принадлежащим узлам, приславшим эти команды.

В случае обнаружения недопустимой команды можно отключить нарушителя от игры, но помните, что недопустимая команда могла быть получена по ошибке — возможно, из-за задержки в сети. Представьте, например, что игрок может произносить заклинания, нападая на соперников. Предположим также, что в этой игре одни игроки могут вызывать немоту у других игроков, то есть последние не смогут произносить заклинания в течение некоторого периода времени. Теперь представьте, что у игрока A вызвали немоту и сервер должен послать соответствующий пакет игроку A. Вполне возможно, что в период с момента, когда сервер послал такой пакет, до момента, когда игрок A принял его, он произнес заклинание. То есть игрок A может послать недопустимую команду, но не по злому умыслу. Поэтому было бы ошибкой отключать игрока A. Обычно в таких ситуациях применяется более консервативный подход, заключающийся в простом игнорировании недопустимого ввода.

Несмотря на то что проверка ввода клиента на сервере или другом узле (в топологии «точка-точка») выполняется достаточно легко, выполнить на клиенте такую

же проверку команд, полученных от сервера, совсем не просто. Это не будет проблемой, если игра выполняется на сервере, который управляется разработчиками, но все меняется, если сервер управляется игроками.

В модели с уполномоченным сервером только сервер имеет полную информацию о состоянии игры. Поэтому если он сообщает клиенту, что тот ранен, клиенту сложно будет проверить, насколько законно такое сообщение. Это вдвойне верно, потому что в типичной конфигурации клиент не имеет возможности напрямую взаимодействовать с другими клиентами. То есть клиент А не может проверить, действительно ли команда пришла от клиента Б, — он вынужден принять на веру информацию, поступившую от сервера.

Простейшее и единственное надежное решение проблемы рассылки мошеннических данных с сервера — не позволять игрокам управлять игровым сервером. С появлением услуг облачного хостинга даже игровые проекты с маленьким бюджетом могут позволить себе разместить сервер в облаке. Несмотря на то что эта услуга оказывается не бесплатно, ее стоимость несравнимо ниже стоимости аренды физического сервера в вычислительном центре. Подробнее о пользовании услугами облачного хостинга для размещения выделенных серверов рассказывается в главе 13.

Однако если в вашем бюджете не предусмотрено средств на оплату услуг хостинга или вы просто решили дать игрокам возможность запускать собственные серверы, решение проблемы значительно усложняется. Один из ограниченных подходов заключается в организации соединения «точка-точка» между клиентами. Это увеличивает сложность программной реализации и повышает требования к пропускной способности, но позволяет выполнять некоторые проверки данных, поступающих с сервера.

Как это работает? Рассмотрим гипотетическую многопользовательскую игру «Вышибалы». Допустим, что клиент Б бросает мяч в клиента А. В стандартной модели «клиент-сервер» эта информация сначала будет отправлена клиентом Б на сервер, а затем с сервера — клиенту А. Чтобы добавить еще один уровень проверки, клиент Б мог бы в момент броска отправить пакет всем остальным клиентам, сообщая, что мяч брошен. Тогда клиент А, получив от сервера пакет с информацией о броске, сможет проверить его законность, сопоставив с пакетом, который он должен был получить от клиента Б.

К сожалению, нет никаких гарантий, что такая система проверки сервера всегда будет давать положительные результаты. Во-первых, то, что сервер доступен для каждого клиента, еще не значит, что каждый клиент будет доступен всем остальным клиентам. В частности, это характерно для конфигураций сетей с NAT, брандмауэрами и т. д. Во-вторых, даже если все клиенты будут доступны друг для друга, нет никаких гарантий, что пакеты, передаваемые между узлами, будут достигать адресатов быстрее, чем пакеты с сервера. То есть если клиент А должен принять решение о допустимости информации, полученной от сервера, он сможет сделать это только после получения соответствующего пакета от клиента Б. Это означает, что клиент А должен будет либо ждать пакет от клиента Б и тем самым задерживать обновление состояния игры, либо отказаться от проверки и поверить серверу на слово.

Программное выявление мошенничества

Оба подхода — противодействие атакам вида «незаконный посредник» и проверка ввода — являются по своей природе оборонительными мерами. Для противодействия атакам вида «незаконный посредник» можно шифровать данные, чтобы их нельзя было прочитать. Для проверки ввода добавляется дополнительный код, препятствующий выполнению недопустимых команд. Однако существует более агрессивный способ противодействия игрокам, пытающимся смошенничать.

В подходе с *программным выявлением мошенничества* используется программный код, являющийся частью игры, или внешний процесс, активно контролирующий целостность игры. Большинство способов мошенничества основано на использовании мошеннического программного обеспечения, выполняющегося на том же компьютере, что и сама игра. Одни из этих программ подключаются к игровому процессу, другие вторгаются в память игрового процесса, третьи — обычно сторонние приложения — предназначены для автоматизации действий, а некоторые даже подменяют файлы с данными, используемые игрой. Действия всех этих разновидностей мошеннических программ можно выявлять с помощью приема программного выявления мошенничества — мощного средства борьбы с мошенниками.

Кроме того, программное выявление мошенничества способно обнаруживать мошеннические действия, которые иными способами не обнаруживаются. Возьмем для примера стратегию реального времени с топологией «точка-точка», использующую механизм детерминированного соответствия. Большинство таких стратегий реализуют «туман войны», скрывающий ландшафт игрового мира в участках, не посещавшихся игровыми модулями игрока. Однако, как рассказывалось в главе 6, в модели «точка-точка» каждый узел обладает полной информацией о состоянии игры. То есть каждый узел хранит в памяти всю информацию о местоположении всех игровых модулей. Это означает, что «туман войны» целиком и полностью реализуется локальным процессом и потому может быть ликвидирован с помощью мошеннической программы. Такого рода мошенничество часто называют *взломом карты* (map hacking), и хотя оно пользуется большой популярностью в основном в стратегиях реального времени, любая игра, реализующая «туман войны», восприимчива к взлому карты. Другие узлы едва ли смогут обнаружить взлом карты — они будут получать самые обычные данные, не содержащие ничего подозрительного. Но реализация программного выявления мошенничества с успехом обнаруживает взлом карты.

Другая популярная разновидность мошенничества — использование *ботов*, которые либо играют в игру вместо игрока, либо помогают игроку в выполнении некоторых действий. Например, боты много лет широко использовались в массовых онлайн-играх игроками, стремящимися повысить свой уровень или получить какие-то приобретения в игре, пока они спят или занимаются чем-то другим. В шутерах от первого лица некоторые игроки использовали ботов, помогающих в прицеливании, чтобы повысить точность своей стрельбы. Обе разновидности ботов могут угрожать целостности игры, и обе обнаруживаются программной реализацией выявления мошенничества.

В конечном счете любая многопользовательская игра, разработчики которой стремятся сплотить вокруг нее здоровое сообщество, должна предусматривать программное выявление мошенничества. В настоящее время существует несколько таких решений. Некоторые из них запатентованы и используются только определенными компаниями — производителями игр, другие доступны бесплатно или за деньги. В оставшейся части этого раздела мы обсудим два решения: Valve Anti-Cheat и Warden. По очевидным причинам объем публикуемой информации о платформах программного выявления мошенничества весьма ограничен, поэтому они будут описаны лишь в общих чертах. Если вы решите реализовать собственное решение, будьте готовы покопаться в низкоуровневом машинном коде и заняться реверс-инжинирингом. Кроме того, даже самые лучшие платформы программного выявления мошенничества можно обмануть. Поэтому мы рекомендуем постоянно обновлять реализацию выявления мошенничества, чтобы быть впереди тех, кто пишет мошеннический код.

Valve Anti-Cheat

Valve Anti-Cheat (VAC) — это платформа программного выявления мошенничества, предназначенная для использования в играх, построенных на основе пакета Steamworks SDK. Подробное описание этого пакета приводится в главе 12 «Игровые службы», а сейчас просто познакомимся с VAC. Если говорить в общих чертах, Valve Anti-Cheat поддерживает список заблокированных пользователей для каждой игры. Когда заблокированный пользователь пытается подключиться к серверу, где используется VAC, он получает отказ в доступе. Некоторые игры даже поддерживают общие списки — например, если игрок был заблокирован в одной игре, он также будет заблокирован в других играх. Это служит мощным сдерживающим фактором.

На верхнем уровне VAC выявляет факты мошенничества во время выполнения, производя поиск известных мошеннических программ. VAC использует несколько методов поиска таких программ, но среди них есть как минимум один, осуществляющий сканирование памяти игрового процесса. Если обнаружится, что игрок пользуется мошеннической программой, он не блокируется немедленно, потому что немедленное блокирование может спугнуть остальных мошенников. Вместо этого VAC просто создает список заблокированных пользователей, чтобы заблокировать их в некоторый момент в будущем. Это дает возможность выявить как можно больше мошенников и потом заблокировать их всех одним махом. Такая практика отложенного блокирования называется игроками *волной банов* (ban wave), и она широко используется многими платформами программного выявления мошенничества.

Имеется также родственный механизм под названием *чистый сервер* (pure server), реализованный на движке Valve Source (и потому может использоваться только в играх на его основе). Чистый сервер проверяет данные в момент подключения пользователя. Сервер сверяет контрольные суммы всех файлов, которые должны находиться на стороне клиента. При подключении к игре клиент должен послать эти контрольные суммы серверу, и если они не совпадают с контрольными, клиенту отказывается в соединении. Эта процедура также выполняется при смене карт, когда производится переход на другой уровень. Для игр, допускающих воз-

можность изменения внешнего вида персонажей, поддерживается белый список файлов и каталогов, которые не должны проверяться. Несмотря на то что эта система ориентирована исключительно на движок Source, нечто подобное вы можете реализовать в своей игре.

Warden

Warden — это платформа программного выявления мошенничества, созданная и используемая в своих играх компанией Blizzard Entertainment. Функции Warden не так очевидны, как VAC. Однако они очень похожи на функции VAC: в процессе игры Warden сканирует память компьютера в поисках известных мошеннических программ. В случае обнаружения любой такой программы информация передается на сервер Warden, и пользователь блокируется в очередную волну банов.

Одной из сильных сторон Warden является возможность обновления непосредственно в процессе игры. Это дает важное тактическое преимущество — обычно мошенники хорошо осведомлены, что обманные действия не следует производить сразу после выпуска обновления к игре, поскольку высока вероятность, что данный вид мошенничества уже не работает, а даже если работает, почти наверняка обнаруживается. Но если обновления применяются динамически, существует возможность застать врасплох мошенников, не заметивших обновления Warden. С другой стороны, авторы некоторых мошеннических программ утверждают, что их программное обеспечение способно определить момент обновления Warden и выгрузить себя до того, как Warden завершит обновление.

Защита сервера

Еще одним важным аспектом безопасности сетевых игр является защита сервера. Она особенно актуальна для игр, использующих выделенный сервер для хранения общего игрового мира, хотя для атаки может оказаться уязвимым любой игровой сервер. Поэтому вы должны запланировать ответные действия для определенных типов атак, а также разработать тактику для непредвиденных ситуаций, когда вы не можете определить тип атаки.

Распределенная атака «отказ в обслуживании»

Цель распределенной атаки *отказ в обслуживании* (Distributed Denial-of-Service, DDoS) — «затопить» сервер запросами, чтобы он не смог нормально функционировать, и в конечном счете сделать его недостижимым или непригодным к использованию для обычных пользователей. Суть такой атаки заключается в передаче серверу достаточно большого объема данных с целью вызвать перенасыщение сетевого соединения или настолько высокие расходы вычислительной мощности на обработку, что сервер не сможет своевременно обрабатывать фактические запросы. Практически каждый достаточно крупный игровой сервер или игровая служба подвергались в какой-то момент DDoS-атаке.

Если вы используете собственное аппаратное обеспечение для развертывания игровых серверов, противостоять DDoS-атакам может быть сложно. Вам потребуется организовать тесное взаимодействие со своим поставщиком услуг Интернета, а также, возможно, обновить аппаратное обеспечение и организовать распределение трафика между несколькими серверами. С другой стороны, если вы пользуетесь услугами облачного хостинга, как описывается в главе 13, некоторые меры по предотвращению DDoS-атак предпринимаются поставщиком услуг хостинга. Все основные платформы облачного хостинга уже имеют определенный уровень защиты от DDoS, а кроме того, за дополнительную плату предоставляются специализированные услуги по противодействию DDoS-атакам. Но как бы то ни было, никогда не следует надеяться, что поставщик услуг облачного хостинга полностью предотвратит возможность DDoS, — правильным будет потратить какое-то время на выработку и тестирование различных защитных стратегий.

Неверные данные

Следует также учитывать, что злонамеренные пользователи могут посылать серверу пакеты с неправильно сформированными или ложными данными. Цели у злоумышленников могут быть разными, но самая простая из них — «обрушить» сервер, вызвать его аварийную остановку. Самые коварные пользователи могут попытаться заставить сервер выполнить вредоносный программный код, вызвав ошибку переполнения буфера или что-либо подобное.

Один из лучших способов защитить игру от неверных данных — использовать разновидность автоматизированного тестирования, получившую название *нечеткое тестирование* (fuzz testing). Обычно нечеткое тестирование используется для вскрытия ошибок, которые стандартно не обнаруживаются в ходе модульного тестирования или проверки качества программного обеспечения. В случае с сетевой игрой нечеткое тестирование можно было бы использовать для передачи серверу больших объемов неструктурированных данных с целью посмотреть, вызовет ли это аварийную остановку сервера, и исправить любые ошибки, вскрытые в ходе такого тестирования.

Чтобы выявить как можно больше ошибок, рекомендуется использовать два вида данных: полностью случайные и более структурированные, например пакеты с ожидаемыми сигнатурами, даже если переносимые ими данные являются совершенно случайными и неструктурированными. Выполнив множество итераций нечеткого тестирования и исправив выявленные при этом ошибки, вы снизите уязвимость игры для неверных данных.

Атаки временным анализом

Любой код, сравнивающий полученную сигнатуру с ожидаемой, теоретически чувствителен к *атакам по времени* (timing attack). Выполняя атаку этого вида, атакующий пытается определить, какая реализация алгоритма хеширования или шифрования используется сервером, анализируя время, затрачиваемое на отклонение недопустимых данных.

Допустим, что игра должна сравнить два массива по восемь 32-разрядных целых чисел в каждом. Один массив, `a`, представляет ожидаемый сертификат. Другой массив, `b`, представляет сертификат, присланный пользователем. Первое, что тут же приходит в голову, написать такую функцию:

```
bool Compare(int a[8], int b[8])
{
 for (int i = 0; i < 8; ++i)
 {
  if (a[i] != b[i])
  {
   return false;
  }
 }
 return true;
}
```

Инструкция `return false` выглядит здесь как вполне безвредная оптимизация производительности: если какой-то элемент не совпадает с ожидаемым, нет никаких причин продолжать сравнивать остальные элементы массивов. Однако из-за этого преждевременного возврата управления данный код уязвим для атаки по времени. При неправильном значении в `b[0]` функция `Compare` вернет управление быстрее, чем при правильном. То есть если пользователь опробует все возможные значения для `b[0]`, он сможет выявить правильное значение, анализируя скорость, с которой `Compare` возвращает управление. Этот процесс можно продолжить для каждого индекса, и в конечном счете пользователь сможет узнать весь сертификат.

Решение этой проблемы: переписать функцию `Compare` так, чтобы она всегда тратила на проверку одно и то же количество времени, независимо от того, какой элемент не совпадает с ожидаемым, `b[0]` или `b[7]`. С этой целью можно воспользоваться поразрядной операцией ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR), которая возвращает ноль, если два значения эквивалентны. То есть для каждой пары элементов из массивов `a` и `b` можно выполнить поразрядную операцию ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR) и сложить результаты поразрядной операцией ИЛИ (OR), как показано ниже:

```
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;
}
```

Вторжения

Еще одну большую проблему для безопасности сервера представляют попытки *вторжения* злонамеренных пользователей, например, в общий игровой мир. Цели такого вторжения могут быть самыми разными: похищение пользовательских

данных, номеров кредитных карт и паролей. Или, что еще хуже, атакующий может попытаться стереть базу данных игры, прекратив тем самым само ее существование. Из всех видов атак вторжения могут иметь самые разрушительные последствия, и потому относиться к ним следует с особой серьезностью.

Существует несколько превентивных мер для ограничения возможностей вторжения. Прежде всего, следует своевременно обновлять все программное обеспечение на серверах, включая операционную систему, базы данных, любые средства автоматизации, веб-приложения и т. д. Причина в том, что старые версии зачастую содержат критические уязвимости, исправленные в новых версиях. Своевременное обновление поможет ограничить круг возможностей для проникновения злоумышленников внутрь игрового сервера. Именно так уменьшение числа служб, действующих на сервере, уменьшит число вероятных точек проникновения.

То же относится к компьютерам разработчиков. Часто вторжения начинаются с проникновения в персональные компьютеры, имеющие доступ к центральному серверу, и использования этих персональных компьютеров в качестве плацдарма для вторжения в серверную систему. Такие атаки называют *направленным фишингом* (spear phishing attack). То есть как минимум сама операционная система и все программы, имеющие доступ в Интернет или в сеть, такие как веб-браузеры на компьютерах разработчиков, всегда должны своевременно обновляться. Другой способ противостоять вторжению на сервер через персональные компьютеры — ограничение доступности критически важного сервера и хранилищ данных для персональных компьютеров. С этой целью можно ввести на серверах обязательную двухфакторную проверку подлинности, чтобы простого знания пароля было недостаточно для получения доступа.

Но какие бы усилия для предотвращения вторжения ни предпринимались, всегда следует считать свой сервер уязвимым для опытного взломщика. Максимально обезопасьте любые уязвимые данные, хранящиеся на сервере. Это поможет уменьшить тяжесть последствий для вашей игры и игроков в случае, если злоумышленнику все же удастся проникнуть на ваш сервер. Например, пароли пользователей никогда не должны храниться в открытом виде, иначе любой получивший доступ к базе данных моментально сможет узнать пароли всех пользователей, а это недопустимо, особенно если учесть, что многие пользователи используют один и тот же пароль для множества учетных записей. Пароли всегда должны хешироваться с использованием соответствующего алгоритма, такого как Blowfish-алгоритм bcrypt. Не используйте более простые алгоритмы хеширования, такие как SHA-256, MD5 или DES, для паролей, потому что все эти устаревшие системы шифрования легко взламываются на современных компьютерах. По аналогии с паролями, шифрованию должна также подвергаться вся платежная информация, такая как номера кредитных карт, с соблюдением соответствующих отраслевых стандартов и правил.

Как свидетельствуют многочисленные публикации об утечках секретной информации, появившиеся в последние годы, часто самой большой угрозой безопасности сервера являются не внешние пользователи, а нечистые на руку или оскорбленные сотрудники. Такой сотрудник может попытаться похитить или опубликовать данные, которые не должны быть ему доступны. Для борьбы с подобными явлениями важно иметь надежную систему журналирования и ревизии. Это может действовать

и как средство устрашения, и как инструмент доказательства преступных действий, если действительно что-то произойдет.

Наконец, обязательно организуйте регулярное резервное копирование любых важных данных на физические носители, находящиеся за пределами сайта. В этом случае даже если из-за действий злоумышленников или по каким-то другим причинам вся база данных будет утрачена, вы сможете восстановить последнюю сохраненную версию данных. Восстановление из резервной копии, конечно же, не самый лучший выход, но это намного лучше, чем полная потеря всех игровых данных.

В заключение

Большинство разработчиков многопользовательских игр должны позаботиться о безопасности. В первую очередь необходимо подумать о безопасности передаваемых данных. Так как пакеты могут перехватываться злоумышленниками, важно шифровать всю уязвимую информацию, такую как пароли и номера кредитных карт. Для этой цели лучше всего подходят методы, основанные на шифровании с открытым ключом, такие как RSA. В отношении данных с состоянием игры желательно максимально уменьшить объем передачи таких данных. Это, в частности, поможет сузить круг возможностей для мошенничества в клиент-серверных играх, потому что в распоряжении клиента будет меньше информации, которую он сможет подделывать.

Проверка ввода также поможет гарантировать, что ни один пользователь не сможет выполнить недопустимое действие. Но недопустимый ввод не всегда связан с мошенничеством: в клиент-серверной игре может возникнуть ситуация, в которой клиент просто не успел получить последние обновления, когда отдавал команду, ставшую недопустимой. С другой стороны, важно обеспечить проверку всех команд, передаваемых по сети. Сервер может проверять ввод клиентов, а узлы могут проверять ввод других узлов. В отношении проверки данных, поступающих с сервера, самый действенный способ — не позволять игрокам запускать собственные серверы.

Несмотря на более агрессивную природу, программное выявление мошенничества может оказаться лучшим средством борьбы с мошенничеством в игре. Типичные инструменты выявления мошенничества сканируют память компьютера, где выполняется игра, в поисках любых известных мошеннических программ. Если такая программа обнаруживается, запустивший ее пользователь блокируется, обычно в какой-то момент в будущем, во время очередной волны банов.

Наконец, важно защищать серверы от разнообразных атак. Целью распределенных атак типа «отказ в обслуживании» является «затопление» сервера запросами, а противостоять им можно, воспользовавшись услугами облачного хостинга. Повысить устойчивость серверного кода к неверным данным в пакетах поможет метод нечеткого тестирования. Наконец, очень важно своевременно обновлять программное обеспечение и шифровать уязвимые данные, хранящиеся на сервере, чтобы нивелировать последствия и уменьшить масштаб разрушений в случае вторжения злоумышленника.

Вопросы для повторения

1. Опишите два разных способа выполнения атаки вида «незаконный посредник».
2. Что такое шифрование с открытым ключом? Как это может пригодиться для защиты от атак вида «незаконный посредник»?
3. Приведите пример, когда проверка ввода может привести к ложному срабатыванию, то есть когда ввод пользователя будет ошибочно расценен как попытка мошенничества, хотя в действительности он таковым не является.
4. Как можно организовать проверку данных, поступающих с сервера, чтобы дать игрокам возможность запускать свои игровые серверы?
5. Почему взлом карты невозможно определить в играх с механизмом детерминированного соответствия без использования программных средств выявления мошенничества?
6. Опишите коротко принцип действия системы Valve Anti-Cheat, препятствующей мошенничеству со стороны игроков.
7. Опишите два разных способа защиты серверов от потенциальных вторжений.

Для дополнительного чтения

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. Доступно по адресу: *https://support.steampowered.com/kb_article.php?ref=7849-Radz-6869*. Проверено 28 января 2016.

11 Игровые движки

Крупные студии, занимающиеся созданием игр, обычно разрабатывают и используют собственные игровые движки, тогда как небольшие компании чаще пользуются готовыми движками. Такой вариант оказывается намного дешевле и эффективнее для небольших студий, выпускающих сетевые игры. В этом случае программисты будут писать сетевой код более высокого уровня, чем тот, который приведен в большей части этой книги.

В данной главе рассматриваются два очень популярных современных игровых движка — Unreal 4 и Unity — и демонстрируется, как с их использованием реализовать сетевые взаимодействия и поддержку многопользовательского режима.

Unreal Engine 4

Игровой движок Unreal Engine в той или иной форме существует с 1998 года, когда вышла видеоигра «Unreal». Однако за прошедшие годы движок сильно изменился. В этом разделе обсуждается версия Unreal Engine 4, выпущенная в 2014 году. В остальной главе под именем «Unreal» мы будем подразумевать игровой движок, а не видеоигру с тем же названием. Разработчик, использующий движок Unreal, не должен волноваться о низкоуровневых тонкостях сетевых взаимодействий. Он может полностью сосредоточиться на высокоуровневой реализации игрового сценария и лишь проверять, правильно ли она работает в сетевом окружении. Это напоминает уровень модели игры в сетевой модели «Tribes».

Поэтому в данном разделе будут рассматриваться в основном высокоуровневые аспекты поддержки сетевых взаимодействий в движке Unreal Engine 4. Однако ради полноты обсуждения имеет смысл упомянуть некоторые низкоуровневые детали и их соотношение с темами, рассматривавшимися в главах с первой по десятую. Читатели, желающие поближе познакомиться с низкоуровневыми тонкостями поддержки сетей в Unreal Engine, могут также бесплатно создать учетную запись разработчика на сайте *www.unrealengine.com* и получить полный доступ к исходному коду.

Сокеты и простые сетевые взаимодействия

Для поддержки как можно более широкого круга платформ движок Unreal должен абстрагировать тонкости реализации сокетов. Классы с интерфейсом `ISocketSubsystem` содержат реализации для разных платформ, поддерживаемых движком Unreal. Это своего рода аналог реализации сокетов Беркли, представленной в главе 3. Как уже говорилось, программный интерфейс сокетов в Windows немного отличается от программного интерфейса в Mac или в Linux, поэтому подсистема сокетов в Unreal должна учитывать эти различия.

Подсистема сокетов отвечает за создание сокетов и адресов. Ее функция `Create` возвращает указатель на экземпляр класса `FSocket`, который затем может использоваться для передачи и приема данных с помощью стандартных функций, таких как `Send`, `Recv` и т. д. В отличие от кода, приводимого в главе 3, функциональность сокетов TCP и UDP не разделена на разные классы.

Имеется также класс `UNetDriver`, реализующий прием, фильтрацию, обработку и отправку пакетов. Его можно считать аналогом класса `NetworkManager`, реализованного в главе 6, но на более высоком уровне. Так же как в случае с подсистемой сокетов, существуют разные реализации, основанные на разных транспортах, будь то IP или транспорт игровой службы, такой как Steam, о которой рассказывается в главе 12 «Игровые службы».

В движке имеется довольно много низкоуровневого кода, реализующего передачу сообщений. Существует также большое число классов для обмена сообщениями, не зависящих от типа транспорта. Они имеют достаточно сложную реализацию, но, если вам интересно, обращайтесь за дополнительной информацией к документации Unreal: *https://docs.unrealengine.com/latest/INT/API/Runtime/Messaging/index.html*.

Игровые объекты и топология

В Unreal используется несколько весьма специфических терминов для обозначения ключевых игровых классов, поэтому прежде чем двигаться дальше, обсудим эту терминологию. Основным базовым классом игровых объектов является класс `Actor`. Каждый объект в игровом мире, статический или динамический, видимый или нет, является экземпляром одного из подклассов `Actor`. Одним из важнейших подклассов `Actor` является класс `Pawn`, который реализует возможность управления им. В частности, класс `Pawn` имеет переменную-член, ссылающуюся на экземпляр класса `Controller`. Класс `Controller` также наследует класс `Actor`, то есть `Controller` также является игровым объектом, требующим обновления. Контроллер может быть экземпляром `PlayerController` или `AIController`, в зависимости от того, кто управляет экземпляром `Pawn`. На рис. 11.1 изображена малая часть иерархии классов движка Unreal.

Чтобы понять, как эти классы взаимодействуют друг с другом, рассмотрим простой пример однопользовательской игры «Вышибалы». Допустим, чтобы бросить мяч, игрок должен нажать клавишу «пробел». Событие ввода пробела передается экземпляру `PlayerController`. Затем экземпляр `PlayerController` извещает экземпляр `PlayerPawn`, что тот должен бросить мяч. В результате `PlayerPawn` создает экземпляр

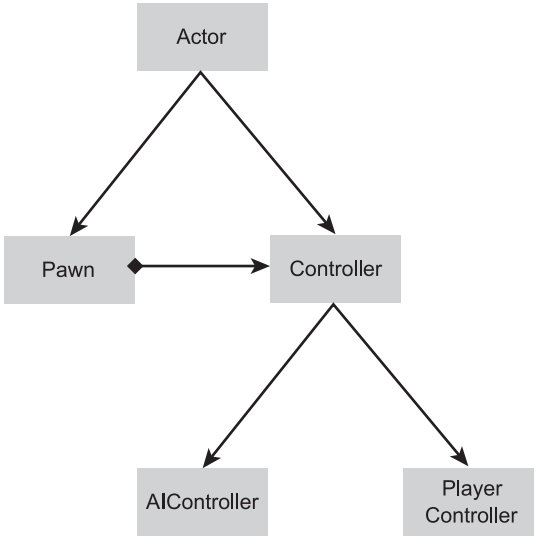


Рис. 11.1. Фрагмент иерархии классов движка Unreal

класса `DodgeBall`, также являющегося подклассом `Actor`. В действительности цепь событий, происходящих за кулисами движка, значительно длиннее, но и такого упрощенного представления вполне достаточно, чтобы понять, как эти классы взаимодействуют друг с другом.

Для сетевых игр Unreal поддерживает только модель «клиент-сервер». Сервер может действовать в двух разных режимах: выделенный сервер и локальный сервер. В режиме *выделенного сервера* (dedicated server) сервер выполняется как процесс, отделенный от всех клиентов. Обычно выделенный сервер выполняется на отдельном компьютере, но это не является обязательным требованием. В режиме *локального сервера* (listen server) один из экземпляров игры принимает на себя функции сервера и клиента одновременно. Существуют некоторые тонкости, отличающие игру с выделенным сервером от игры с локальным сервером, но они не будут здесь рассматриваться.

Репликация акторов

Учитывая, что Unreal использует модель «клиент-сервер», он должен реализовать возможность передачи обновлений с сервера всем акторам на клиентах. Как нетрудно догадаться, эта функция называется *репликацией акторов*. Unreal стремится сократить число акторов, требующих репликации. Так же как в модели «Tribes», Unreal определяет множество акторов, релевантных для каждого клиента. Кроме того, если актор может быть релевантным только для одного клиента, он может быть создан на этом клиенте, а не на сервере. Такие акторы применяются, например, для создания эффекта разлетающихся осколков. Кроме того, поддерживается возможность управления релевантностью акторов при помощи нескольких флагов.

Например, флаг `bAlwaysRelevant` существенно увеличивает вероятность признания актора релевантным (в противоположность своему имени, данный флаг *не гарантирует*, что актор всегда (always) будет считаться релевантным).

За релевантностью следует еще одно важное понятие — *роли*. В сетевой многопользовательской игре может одновременно действовать несколько отдельных экземпляров игры. Каждый из экземпляров может запросить свою роль в отношении каждого актора, чтобы определить, кто владеет им. Важно понимать, что в ответ на запросы о роли для конкретного актора разным экземплярам игры могут возвращаться *разные значения*. Вернемся к примеру игры «Вышибалы». В сетевой многопользовательской версии игры мяч может порождаться сервером. То есть если сервер запросит свою роль в отношении мяча, он может получить в ответ роль «authority» (управляющий), указывающую, что он наделен полномочиями управления актором мяча. При этом все остальные клиенты получают в ответ роль «simulated proxy» (моделирующий агент), сообщающую, что клиент может лишь моделировать действия данного актора, но не управлять им. Ниже перечислены три возможные роли:

- **Authority** (управляющий). Экземпляр игры наделен полномочиями управления актором.
- **Simulated proxy** (моделирующий агент). Для клиента эта роль означает, что актором владеет и распоряжается сервер. Клиент с ролью моделирующего агента может моделировать лишь некоторые аспекты актора, например движение.
- **Autonomous proxy** (автономный моделирующий агент). Роль автономного моделирующего агента очень похожа на роль простого моделирующего агента, но предполагает, что моделируемый актор может принимать события ввода непосредственно от текущего экземпляра игры, то есть в процессе моделирования актора должен учитываться ввод игрока.

Это не значит, что в многопользовательской игре сервер всегда является полноправным владельцем всех акторов. В некоторых случаях клиенты могут порождать свои акторы, например, для создания эффекта разлетающихся осколков, и для таких акторов они будут получать роль «authority», при этом сервер даже не будет подозревать об их существовании.

Однако каждый актор, для которого сервер играет роль «authority», будет передаваться всем клиентам при условии релевантности. Внутри акторов можно указать, какие свойства должны или не должны передаваться. Благодаря этому экономится полоса пропускания, поскольку передаются только свойства, необходимые для моделирования актора на стороне клиента. Репликация акторов в Unreal выполняется *только* в одном направлении — от сервера к клиенту. Клиент не может создать актор и затем передать его серверу (или другим клиентам).

Настройка репликации не ограничивается одним только определением копируемых свойств. К примеру, можно настроить репликацию свойства по условию. Также можно указать функцию, которая должна вызываться на клиенте при получении определенного свойства от сервера. Так как игровой код в Unreal Engine 4 пишется на языке C++, движок использует сложный набор макросов для определения разнообразных параметров репликации. Соответственно, добавляя

переменную-член в определение класса, можно при помощи макросов определить для нее параметры репликации. В Unreal имеется также довольно мощная система сценариев на основе диаграмм с названием *Blueprint*. Следует отметить, что большая часть функций поддержки многопользовательского режима также доступна через систему сценариев.

Кроме того, в Unreal уже реализовано прогнозирование движения акторов на стороне клиента. В частности, если в акторе установить флаг `bReplicateMovement`, он будет копироваться и прогнозировать свое местоположение на клиенте, опираясь на информацию о скорости движения. При необходимости можно также переопределить метод, осуществляющий прогнозирование. Однако для большинства игр реализация по умолчанию дает очень неплохие результаты.

Удаленный вызов процедур

Как рассказывалось в главе 5 «Репликация объектов», удаленный вызов процедур (Remote Procedure Calls, RPC) играет важную роль в репликации. Поэтому неудивительно, что в Unreal имеется весьма мощная система поддержки RPC, реализующая три типа вызовов: на сервере, на клиенте и широковещательные.

Серверная функция — это функция, вызываемая клиентом и выполняющаяся на сервере, с одной большой оговоркой: сервер не позволяет любому клиенту вызывать функции в любых акторах, потому что помимо всех прочих неприятностей такая открытость способствовала бы увеличению вероятности мошенничества. Вместо этого только клиент, являющийся *владельцем* актора, может вызвать серверную функцию в этом акторе. Обратите внимание, что в данном случае слово «владелец» имеет иной смысл, нежели роль «authority» для экземпляра игры. Под владельцем здесь подразумевается экземпляр `PlayerController`, связанный с актором. Например, если контроллер A (типа `PlayerController`) управляет игровым объектом A (типа `PlayerPawn`), то клиент, управляющий контроллером A, считается владельцем объекта A. Если вернуться к примеру игры «Вышибалы», это означает, что только клиент A может вызвать серверную функцию `ThrowDodgeBall` объекта A, — любые другие попытки со стороны клиента A вызвать `ThrowDodgeBall` в других объектах `PlayerPawn` будут просто игнорироваться.

Клиентская функция — противоположность серверной функции. Когда сервер вызывает клиентскую функцию, вызов передается клиенту, являющемуся владельцем актора. Например, если в игре «Вышибалы» сервер определяет, что мяч попал в игрока B, он может вызвать клиентскую функцию у игрока B, чтобы тот вывел на экран сообщение: «Выбит!».

Как можно заключить из названия, *широковещательная функция* будет вызвана в множестве экземпляров игры. В частности, широковещательная функция вызывается сервером, но выполняется не только на клиентах, но и на самом сервере. Широковещательные функции используются для извещения всех клиентов о наступлении некоторого события: например, широковещательную функцию сервер мог бы вызвать, чтобы каждый клиент создал локальный актор, воспроизводящий эффект разлетающихся осколков.

Все вместе эти три типа удаленных вызовов процедур открывают массу возможностей. Кроме того, движок Unreal предоставляет на выбор два механизма RPC: надежный и ненадежный. То есть функции RPC для обработки низкоприоритетных событий можно отметить как ненадежные и избежать снижения производительности в случае потери пакетов.

Unity

Первая версия игрового движка Unity была выпущена в 2005 году. За последние несколько лет он приобрел большую популярность у многих разработчиков игр. Так же как Unreal, движок Unity обладает встроенными средствами синхронизации и RPC, хотя и несколько отличающимися от используемых в Unreal. В состав версии Unity 5.1 вошла новая сетевая библиотека UNET, поэтому в данном разделе основное внимание будет уделено этой новейшей библиотеке. В UNET реализовано два разных программных интерфейса (API): высокоуровневый API, удовлетворяющий потребностям большинства сетевых игр, и низкоуровневый транспортный API, который можно использовать для реализации нестандартных взаимодействий через Интернет. В этом разделе будет рассматриваться высокоуровневый API.

Несмотря на то что основная часть игрового движка Unity написана на C++, разработчики Unity не дают доступа к коду на C++. Разработчики, использующие Unity, основную массу программного кода пишут на C#, однако есть возможность использовать Unity из сценариев на JavaScript. Большинство опытных разработчиков отдают предпочтение C#. Программированию игровой логики на C# вместо C++ присущи свои достоинства и недостатки, но к нашей задаче это не имеет отношения.

API транспортного уровня

Программный интерфейс транспортного уровня в UNET служит оберткой вокруг платформенно-зависимых сокетов. Как можно догадаться, существуют функции для создания соединений с другими узлами сети, через которые можно посылать и принимать данные. При создании соединения требуется выбрать между надежным и ненадежным соединениями. Но вместо выбора между UDP- и TCP-соединением нужно указать, как это соединение будет использоваться. При создании коммуникационного канала укажите одно из нескольких значений из перечисления `QosType`:

- `Unreliable`. Сообщения отправляются без гарантий доставки.
- `UnreliableSequenced`. Доставка сообщений не гарантируется, и сообщения, полученные не по порядку, отбрасываются. Эта разновидность соединения может использоваться для обмена голосовыми сообщениями.
- `Reliable`. Доставка сообщений гарантируется, если только соединение не будет разорвано.
- `ReliableFragmented`. Сообщение может быть разбито на несколько пакетов. Эта разновидность соединения может использоваться для передачи по сети больших файлов, так как предусматривает сборку сообщений на стороне получателя.

Установить соединение можно вызовом функции `NetworkTransport.Connect`. Она вернет числовой идентификатор (ID) соединения, который затем используется как параметр в вызовах других функций `NetworkTransport`, таких как `Send`, `Receive` и `Disconnect`. Функция `Receive` возвращает значение типа `NetworkEventType`, включающее принятые данные или событие, такое как событие разъединения.

Игровые объекты и топология

Самым существенным отличием Unity от Unreal является порядок создания игровых объектов. Если в Unreal используется относительно монолитная иерархия игровых объектов и акторов, то в Unity применяется модульный подход. Класс `GameObject` в Unity является, по сути, лишь контейнером для экземпляров класса `Component`. Все игровые функции делегируются компонентам, содержащимся в данном экземпляре `GameObject`. Это позволяет более четко разграничить разные аспекты поведения игровых объектов, хотя иногда может осложнить программирование, особенно при наличии зависимостей между несколькими компонентами. Обычно `GameObject` включает один или несколько компонентов, наследующих `MonoBehaviour`, которые реализуют различные функциональные возможности для `GameObject`. Так, например, вместо класса `PlayerCat`, непосредственно наследующего `GameObject`, вы должны определить компонент `PlayerCat`, наследующий `MonoBehaviour`, и присоединить к любому игровому объекту, который должен действовать, как `PlayerCat`.

Для хранения состояния игры в сетевом API движка Unity используется класс `NetworkManager`. Экземпляр этого класса может действовать в трех режимах: как автономный клиент, как автономный (выделенный) сервер или как гибридный узел, совмещающий функции клиента и сервера. Это означает, что фактически Unity поддерживает те же самые режимы выделенного и локального сервера, которые поддерживаются движком Unreal.

Создание объектов и репликация

Поскольку в Unity используется топология «клиент-сервер», процедура создания объектов в сетевой игре на основе Unity значительно отличается от аналогичной процедуры в однопользовательской игре. В частности, когда вызовом функции `NetworkServer.Spawn` на сервере создается игровой объект, это означает, что данный объект будет контролироваться сервером по сетевому идентификатору экземпляра. Кроме того, игровой объект, созданный таким способом, должен быть создан на всех клиентах и копироваться при обновлении. Чтобы клиент смог создать правильный игровой объект, для него должна быть зарегистрирована соответствующая *заготовка* (prefab). Заготовка (prefab) в Unity представляет собой коллекцию компонентов, данных и сценариев, используемых игровым объектом, — в их число могут входить, например, трехмерные модели, звуковые эффекты и сценарии, реализующие поведение. Регистрация заготовки на стороне клиента гарантирует готовность всех данных на стороне клиента к моменту, когда сервер сообщит клиенту, что тот должен создать экземпляр данного игрового объекта.

После создания объекта на сервере свойства его компонентов можно передать клиентам несколькими способами. Однако для этого требуется, чтобы компоненты наследовали класс `NetworkBehaviour`, а не обычный `MonoBehaviour`. Самый простой способ организовать репликацию переменных-членов — снабдить каждую переменную атрибутом `[SyncVar]`. Этот способ подходит для репликации встроенных типов, а также типов Unity, таких как `Vector3`. Любые переменные с атрибутом `SyncVar` будут автоматически передаваться клиентам при изменении их значений — от вас не требуется отмечать их как изменившиеся. Но имейте в виду, что атрибут `SyncVar` можно применять также к пользовательским структурам, и в этом случае содержимое структуры будет копироваться как одно целое. То есть если определена структура с 10 членами, но изменился только один, в сеть будут скопированы все 10 членов, а это может отрицательно сказаться на пропускной способности.

На тот случай, если потребуется более точное управление репликацией переменных, следует переопределить функции-члены `OnSerialize` и `OnDeserialize` и вручную читать и записывать переменные, подлежащие синхронизации. Такое решение позволяет организовать нестандартный способ репликации, но оно несовместимо с `SyncVar`, поэтому вам придется выбирать тот или иной подход.

Вызов удаленных процедур

Движок Unity также поддерживает вызов удаленных процедур (Remote Procedure Calls, RPC), но используемая в нем терминология немного отличается от терминологии, приведенной в этой книге. В Unity под *командой* подразумевается передача операции от клиента серверу. Команды могут посылаться только объектам, которыми управляет игрок. Вызов *клиентской функции RPC*, напротив, — передача операции от сервера клиенту. По аналогии с атрибутом `SyncVar` функции RPC этого типа поддерживаются только для подклассов `NetworkBehaviour`.

Объявление функций, поддерживающих удаленный вызов, во многом напоминает объявление синхронизируемых переменных. Чтобы объявить функцию командой, ее следует снабдить атрибутом `[Command]`, а имя функции должно начинаться с приставки `Cmd`, например `CmdFireWeapon`. Аналогично, клиентскую функцию RPC следует снабдить атрибутом `[ClientRpc]`, а ее имя должно начинаться с приставки `Rpc`. Обе разновидности функций могут вызываться как обычные функции C#, с той лишь разницей, что автоматически будут созданы данные для передачи по сети и фактический вызов будет выполнен удаленно.

Координация

Кроме того, библиотека UNET предоставляет некоторые функции координации, обычно выполняемые игровыми службами, о которых рассказывается в главе 12 «Игровые службы». В отличие от движка Unreal, где имеются лишь обертки для игровых служб на рассматриваемой платформе, в Unity предусмотрена возможность запрашивать список активных игровых сеансов и выбирать нужный сеанс. После выбора сеанса можно присоединиться к игре. Эта функциональность добавляется в подкласс `MonoBehaviour` при помощи экземпляра

класса NetworkMatch, который затем будет вызывать обработчики OnMatchCreate, OnMatchList и OnMatchJoined.

В заключение

Для небольших студий, занимающихся созданием компьютерных игр, выбор готового движка может оказаться самым разумным решением. В этом случае программисты будут писать сетевой код более высокого уровня, чем тот, который приведен в большей части этой книги. Вместо того чтобы заботиться об операциях с сокетами и о сериализации данных, программисты должны будут изучить функции выбранного движка, необходимые для реализации сетевой игры.

Игровой движок Unreal Engine существует уже почти 20 лет. Четвертая версия движка, выпущенная в 2014 году, доступна с полными исходными текстами на C++. Несмотря на то что в движке существует множество платформенно-зависимых оберток для таких функциональных элементов, как сокеты и адреса, предполагается, что разработчикам не придется напрямую использовать эти классы.

Сетевая модель движка Unreal поддерживает топологию «клиент-сервер» с выделенным или локальным сервером. Версия игрового объекта Actor образует в Unreal иерархию, включающую множество разных подклассов. Важным аспектом этой иерархии является идея сетевой роли. Роль «authority» (управляющий) означает, что экземпляр игры управляет данным объектом, тогда как роли «simulated proxy» (моделирующий агент) и «autonomous proxy» (автономный моделирующий агент) используются, если клиент просто отражает объект, действующий на сервере. Класс Actor имеет встроенную поддержку репликации объектов. Репликация действий, таких как перемещение, осуществляется простой установкой флага типа Boolean, при этом нестандартные параметры также можно пометить как подлежащие репликации. Кроме того, поддерживаются разные способы удаленного вызова процедур.

Первая версия движка Unity была выпущена в 2005 году, и за последние несколько лет он приобрел большую популярность. Разработчики, использующие Unity, основную массу программного кода пишут на C#. В состав версии Unity 5.1 вошла новая сетевая библиотека UNET, предоставляющая замечательный высокоуровневый программный интерфейс к сетевым операциям. Впрочем, в ней также реализован и низкоуровневый транспортный слой.

Транспортный уровень скрывает функции создания сокетов и вместо этого предоставляет разработчикам возможность организовать передачу данных в нескольких режимах, включая надежный и ненадежный, но большинство игр на основе Unity не используют транспортный уровень непосредственно. Вместо этого создатели игр обычно работают с высокоуровневым API, который, по аналогии с Unreal, поддерживает работу в режиме выделенного и локального сервера. Все компоненты, требующие сетевой поддержки, должны наследовать класс NetworkBehaviour, добавляющий возможность репликации. Организация репликации осуществляется с помощью атрибута [SyncVar] или путем подстановки нестандартных функций сериализации. Аналогичный подход используется для удаленного вызова процедур, как на стороне сервера, так и на стороне клиента. Наконец, Unity обладает упро-

щенной поддержкой координации, которую можно использовать как легковесную замену полноценным игровым службам.

Вопросы для повторения

1. Оба движка, Unreal и Unity, обладают встроенной поддержкой только топологии «клиент-сервер» и не поддерживают топологию «точка-точка». Как вы думаете, чем объясняется такое решение?
2. Какие роли могут иметь акторы в сетевой игре на основе Unreal и в чем заключается их важность?
3. Опишите разные случаи использования механизма удаленного вызова процедур в Unreal.
4. Опишите, как действует модель игровых объектов и компонентов в Unity. Какие преимущества и недостатки такой модели вы могли бы назвать?
5. Как в Unity реализована синхронизация переменных и вызов удаленных процедур?

Для дополнительного чтения

Epic Games. «Networking & Multiplayer». Unreal Engine. *https://docs.unrealengine.com/latest/INT/Gameplay/Networking/*. Проверено 28 января 2016.

Unity Technologies. «Multiplayer and Networking». Unity Manual. *http://docs.unity3d.com/ru/current/Manual/UNet.html*. Проверено 28 января 2016.

12 Игровые службы

В наши дни многие игроки имеют учетные записи в игровых службах, таких как Steam, Xbox Live или PlayStation Network. Эти службы предоставляют массу возможностей игрокам и играм, включая координацию, хранение статистики, наград, рекордов и состояния игры в облаке и многое другое. Поскольку эти игровые службы получили широкое распространение, игроки ожидают, что все игры, даже однопользовательские, будут интегрированы с одной из таких служб. В этой главе мы поговорим о том, как интегрировать такие службы в вашу игру.

Выбор игровой службы

При таком богатстве выбора стоит задуматься, какую игровую службу интегрировать в игру. Иногда выбор оказывается предопределен платформой, на которой должна действовать игра. Например, все игры для Xbox One должны интегрировать службу Xbox Live, потому что просто невозможно в игру для Xbox One интегрировать службу PlayStation Network. Однако игры для PC, Mac и Linux оставляют возможность выбора. Вне всяких сомнений, самой популярной службой для этих платформ на сегодняшний день является служба Steam, поддерживаемая компанией Valve Software. Она существует уже более 10 лет и имеет базу, включающую тысячи игр. Учитывая, что игра «RoboCat RTS» написана нами для платформы PC/Mac, будет вполне разумно интегрировать в нее поддержку именно службы Steam.

Прежде чем интегрировать службу Steam в игру, необходимо выполнить несколько условий. Сначала надо принять лицензионное соглашение «Steamworks SDK Access Agreement». Это соглашение доступно по адресу: *https://partner.steamgames.com/documentation/sdk_access_agreement*. Далее следует зарегистрироваться как партнер Steamworks, для чего потребуется подписать соглашение о неразглашении, а также предоставить соответствующую информацию. Наконец, нужно получить идентификатор приложения для своей игры. Идентификатор приложения выдается только после того, как будет подписано соглашение о партнерстве с Steamworks и ваша игра получит «зеленый свет» в Steam.

Однако уже после первого шага — принятия соглашения «Steamworks SDK Access Agreement» — вы получите доступ к файлам SDK, документации и примеру игрового проекта (с названием «SpaceWar»), уже имеющему собственный идентификатор приложения. Для демонстрационных целей примеры кода в этой главе используют идентификатор приложения для «SpaceWar». Этого более чем достаточно, чтобы понять, как интегрировать Steamworks в игру после выполнения всех остальных шагов и получения своего уникального идентификатора.

Основные настройки

Прежде чем писать какой-либо код, имеющий отношение к игровой службе, посмотрим, как лучше интегрировать его в игру. Самый простой путь — вставить вызовы игровой службы везде, где это необходимо. В нашем случае можно непосредственно вызывать функции Steamworks SDK во всех файлах, где требуются услуги игровой службы. Однако такой подход не следует рекомендовать к использованию по двум причинам. Во-первых, для этого потребуется, чтобы все разработчики в коллективе имели определенное представление о Steamworks, потому что код, использующий эту службу, будет разбросан по всей игре. Во-вторых, что особенно важно, это усложнит интеграцию с другими игровыми службами. Данная проблема особенно актуальна для кроссплатформенных игр, потому что, как уже говорилось, платформы накладывают свои ограничения на использование игровых служб. Поэтому даже при том условии, что «RoboCat RTS» пока существует в версии только для PC и Mac, если позднее возникнет желание перенести ее на PlayStation 4, понадобится сделать переход с Steamworks на PlayStation Network как можно более гладким. А поскольку код, вызывающий Steamworks, будет разбросан по всей игре, это станет серьезным препятствием на пути к поставленной цели.

Описанные проблемы подводят нас к главному архитектурному решению этой главы, определяющему порядок интеграции игровых служб. Определения в заголовочном файле `GamerServices.h` не связаны ни с какими функциями или объектами Steamworks, поэтому нет необходимости подключать заголовочный файл `steam_api.h`. Добиться этого помогло применение *указателя на реализацию* — идиомы программирования на C++, используемой для сокрытия особенностей реализации классов. Чтобы объявить указатель на реализацию, необходимо определить класс, включающий опережающее объявление класса реализации, и указатель на него. Благодаря этому детали реализации класса отделяются от его объявления. Базовые компоненты указателя на реализацию в классе `GamerServices` представлены в листинге 12.1. Обратите внимание, что класс использует `unique_ptr` вместо обычного указателя, как это рекомендуется в современном C++.

Листинг 12.1. Указатель на реализацию в GamerServices.h

```
class GamerServices
{
public:
  // множество других объявлений опущено
```

```
  //...

  // опережающее объявление
  struct Impl;
private:
  // указатель на реализацию
  std::unique_ptr<Impl> mImpl;
};
```

Важно отметить, что сам класс реализации никогда не объявляется полностью в заголовочном файле. Вместо этого детали класса реализации объявляются в объектном файле — в данном случае в файле `GamerServicesSteam.cpp`, и там же выполняется инициализация указателя `mImpl`. Это означает, что вызовы любых функций Steamworks API будут выполняться только в этом файле C++. Если позднее понадобится интегрировать службу Xbox Live, достаточно будет создать другую реализацию класса `GamerServices` в `GamerServicesXbox.cpp` и добавить этот новый файл в проект вместо реализации Steam, при этом, в теории, не менять никакой другой код.

Указатель на реализацию позволяет абстрагироваться от платформенно-зависимых деталей, но этот прием порождает проблему снижения производительности, особенно важную для игр. При использовании указателя на реализацию для вызовов подавляющего большинства функций-членов потребуется выполнять дополнительную, достаточно дорогостоящую операцию разыменования указателя. Для класса, производящего большое число вызовов функций-членов, такого как устройство отображения, падение производительности может оказаться весьма заметным. Однако в объекте `GamerServices` не придется делать много вызовов в каждом кадре. Поэтому здесь можно пожертвовать производительностью в угоду гибкости.

Следует также отметить, что функциональность, доступная в объекте `GamerServices`, — лишь малая часть возможностей Steamworks. Это объясняется тем, что в объект включены обертки только для функциональности, необходимой в игре «RoboCat RTS»; при желании, конечно же, возможности объекта можно расширить. Однако если вы решите существенно расширить возможности, мы рекомендуем разделить код интеграции с игровыми службами, поместив его в несколько файлов. Например, вместо того чтобы хранить определения всех сетевых функций непосредственно в `GamerServices`, можно создать класс `GamerServiceSocket`, реализующий функциональность, напоминающую `TCPSocket` или `UDPSocket`.

Инициализация, запуск и остановка

Инициализация Steamworks производится вызовом `SteamAPI_Init`. Эта функция не имеет входных параметров и возвращает логическое значение, сообщающее об успехе инициализации. Вызов функции выполняется в `GamerServices::StaticInit`. Примечательно, что инициализация игровых служб происходит в `Engine::StaticInit` *до* инициализации механизма отображения. Объясняется это тем, что одной из особенностей Steam является *оверлей*. Оверлей позволяет игрокам, например,

обмениваться сообщениями в чате или использовать веб-браузер, не покидая игры. Работа оверлея основывается на перехвате вызовов функций OpenGL. Поэтому для корректной работы оверлея `SteamAPI_Init` должна вызываться до инициализации механизма отображения. В случае успеха она заполнит группу глобальных указателей на интерфейс, которые затем будут доступны через глобальные функции, такие как `SteamUser`, `SteamUtils` и `SteamFriends`.

Обычно игра, интегрированная со службой Steam, запускается с помощью клиента Steam. Именно так Steamworks узнает идентификатор приложения для игры, которая должна быть запущена. Однако в процессе разработки редко кто запускает игру с помощью клиента Steam — чаще вам придется запускать ее под управлением отладчика или как автономный исполняемый файл. Чтобы механизм Steamworks мог узнать идентификатор приложения во время разработки, в каталог с исполняемым файлом следует поместить файл `steam_appid.txt` с этим идентификатором. Однако даже при том, что это избавляет от необходимости запускать игру посредством клиента Steam, экземпляр клиента все равно должен быть запущен и зарегистрирован в службе с учетными данными пользователя. Если у вас нет клиента Steam, его можно получить на веб-сайте Steam: *http://store.steampowered.com/about/*.

Кроме того, чтобы проверить игру нескольких пользователей в Steam друг против друга, необходимо создать несколько тестовых учетных записей. Тестирование на локальном компьютере выглядит значительно сложнее, чем тестирование версии игры из главы 6, потому что на одном компьютере нельзя запустить несколько экземпляров Steam. Поэтому для тестирования многопользовательского режима вам придется использовать несколько компьютеров или запустить несколько экземпляров виртуальной машины.

Так как Steamworks часто приходится взаимодействовать с удаленным сервером, многие функции этого механизма действуют асинхронно. Чтобы известить приложение о завершении асинхронного вызова, Steamworks использует обратные вызовы. Для гарантированного выполнения этих обратных вызовов игра должна регулярно вызывать `SteamAPI_RunCallbacks`. Рекомендуется вызывать эту функцию один раз в каждом кадре, именно для этого данный вызов помещен в функцию `GamerServices::Update`, которая вызывается один раз в каждом кадре из `Engine::DoFrame`.

Остановка механизма Steamworks выполняется так же просто, как его инициализация, посредством функции `SteamAPI_Shutdown`. Она вызывается в деструкторе `GamerServices`.

Игры с топологией «клиент-сервер», кроме того, должны инициализировать/останавливать игровой сервер при помощи функций `SteamGameServer_Init` и `SteamGameServer_Shutdown`. Чтобы получить доступ к этим функциям, необходимо подключить заголовочный файл `steam_gameserver.h`. Выделенные серверы могут действовать в анонимном режиме, не требуя от пользователей сообщать свои учетные данные. Но так как игра «RoboCat RTS» использует только взаимодействия «точка-точка», код для этой главы не использует никаких функций управления игровым сервером.

Имена и идентификаторы пользователей

В версии «RoboCat RTS», обсуждавшейся в главе 6, идентификаторы игроков хранились как 32-разрядные целые числа без знака. Как вы наверняка помните, в той версии игры идентификаторы назначались игрокам ведущим узлом. При использовании игровой службы каждый игрок уже имеет уникальный идентификатор, присвоенный ему службой, поэтому не имеет смысла присваивать им какие-то другие уникальные идентификаторы. В случае Steamworks уникальные идентификаторы представлены экземплярами класса `CSteamID`. Однако модульная организация класса `GamerServices` потеряла бы смысл, если бы всюду использовались объекты `CSteamID`. К счастью, идентификаторы `CSteamID` можно преобразовывать в 64-разрядные целые числа и создавать их из таких чисел.

Поэтому чтобы обеспечить соответствие между идентификаторами Steam и идентификаторами игроков, необходимо изменить тип всех переменных, предназначенных для хранения идентификаторов, объявив их как `uint64_t`. Кроме того, теперь для присвоения идентификатора игроку экземпляр `NetworkManager` на ведущем узле должен запросить идентификатор у объекта `GamerServices`, в частности вызвать функцию `GetLocalPlayerId`, представленную в листинге 12.2.

Листинг 12.2. Основные функции для получения имен и идентификаторов пользователей

```
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));
}
```

В листинге 12.2 также представлены похожие тонкие обертки, позволяющие получить имя локального или удаленного игрока. Вместо того чтобы вынуждать игроков вводить свои имена *name*, как в старой версии «RoboCat», можно использовать имя, указанное игроком в Steam.

Обратите внимание, что, несмотря на использование 64-разрядных чисел в качестве идентификаторов пользователей в Steamworks, нет никаких гарантий, что аналогичные идентификаторы будут использоваться во всех игровых службах. Например, другие службы могут использовать для идентификации пользователей 128-разрядные универсальные уникальные идентификаторы (Universally Unique

Identifier, UUID). Чтобы решить эту проблему, можно добавить еще один уровень абстракции: например, создать класс `GamerServiceID`, который обертывает внутреннее представление, используемое игровой службой для идентификации.

Вступление в игру и координация

Ранняя версия «RoboCat RTS» включала нетривиальный объем кода, реализующего сбор новых игроков в предыгровом фойе (pregame lobby), где они ожидают вступления в игру. Каждый новый узел должен сначала отправить пакет приветствия ведущему узлу, дождаться ответного приветствия и, наконец, представиться всем остальным узлам, участвующим в игре. В примере для этой главы весь код, реализующий процедуру вступления в игру, был удален. Служба Steam, подобно большинству других игровых служб, реализует собственную процедуру вступления в игру. Поэтому мы рекомендуем использовать средства Steam, особенно если учесть, что они обладают более широким функционалом, чем аналогичная процедура, реализованная в «RoboCat».

В общем случае процесс вступления в многопользовательскую игру с помощью Steam выглядит примерно так:

1. Игра отыскивает подходящее фойе, опираясь на пользовательские параметры. В число таких параметров может входить режим игры и даже уровень мастерства (если игра предусматривает деление игроков по уровням).
2. Если найдено одно или несколько подходящих фойе, игра автоматически выбирает одно из них или предоставляет пользователю возможность выбора из списка. Если подходящее фойе не найдено, игра может создать его. В любом случае, после выбора или создания фойе игрок вступает в него.
3. Находясь в фойе, можно дополнительно настроить некоторые параметры предстоящей игры, такие как выбор персонажей, карты и т. д. В этот момент в фойе могут зайти другие игроки. Кроме того, игроки в одном фойе могут обмениваться сообщениями в чате.
4. Как только подготовка игры к запуску будет закончена, игроки присоединятся к ней и покинут фойе. Обычно при этом выполняется подключение к игровому серверу (выделенному или размещенному на компьютере одного из игроков). В игре «RoboCat RTS» сервер отсутствует, поэтому перед выходом из фойе игроки устанавливают соединения друг с другом.

Так как в «RoboCat» отсутствуют какие-либо меню или средства выбора режимов, игра начинает поиск фойе почти сразу после инициализации Steamworks. Поиск осуществляет функция `LobbySearchAsync`, представленная в листинге 12.3. В поиске используется единственный фильтр — название игры, который гарантирует обнаружение фойе только для игры «RoboCat». Но при желании можно применить любые другие фильтры, вызвав соответствующие функции установки фильтров перед вызовом `RequestLobbyList`. Обратите внимание, что `LobbySearchAsync` требует вернуть только один результат, потому что игра предусматривает автоматический вход в первое найденное фойе.

Листинг 12.3. Поиск фойе

```
const char* kGameName = "robocatrts";

void GamerServices::LobbySearchAsync()
{
  // гарантировать выбор фойе игры Robo Cat RTS!
  SteamMatchmaking()->AddRequestLobbyListStringFilter("game",
    kGameName, k_ELobbyComparisonEqual);

  // потребовать единственный результат
  SteamMatchmaking()->AddRequestLobbyListResultCountFilter(1);

  SteamAPICall_t call = SteamMatchmaking()->RequestLobbyList();
  mImpl->mLobbyMatchListResult.Set(call, mImpl.get(),
    &Impl::OnLobbyMatchListCallback);
}
```

Использование структуры `SteamAPICall_t` в `LobbySearchAsync` требует дополнительных пояснений. В Steamworks SDK все асинхронные вызовы возвращают структуру `SteamAPICall_t`, которая фактически является дескриптором асинхронного вызова. После получения дескриптора нужно сообщить механизму Steamworks, какая функция должна быть вызвана по завершении асинхронного вызова. Эта связь между дескриптором асинхронного вызова и функцией обратного вызова устанавливается с помощью экземпляра `CCallResult`. В данном случае этот экземпляр хранится в поле `mLobbyMatchListResult` класса реализации. Это поле и функция `OnLobbyMatchListCallback` определены внутри `GamerServices::Impl`:

```
// Результат вызова, когда приобретается список фойе
CCallResult<Impl, LobbyMatchList_t> mLobbyMatchListResult;
void OnLobbyMatchListCallback(LobbyMatchList_t* inCallback, bool inIOFailure);
```

В данном конкретном случае реализация `OnLobbyMatchListCallback`, представленная в листинге 12.4, имеет пару особенностей, на которых следует остановиться. Обратите внимание на логический параметр `inIOFailure`. Его имеют все функции обратного вызова. Значение `true` в этом параметре говорит, что во время асинхронной операции произошла ошибка и функция должна немедленно завершиться. Если обнаружено имеющееся фойе, функция попытается войти в него. В противном случае она создаст новое фойе. В обоих случаях вызывается еще одна асинхронная функция, поэтому здесь упоминаются дополнительно две функции обратного вызова: `OnLobbyEnteredCallback` и `OnLobbyCreateCallback`. Их реализации можно найти в примерах кода. Самое важное в этих функциях, на что следует обратить внимание: когда игрок входит в фойе, `NetworkManager` извещается об этом событии вызовом функции `EnterLobby`.

Листинг 12.4. Функция, вызываемая по завершении поиска фойе

```
void GamerServices::Impl::OnLobbyMatchListCallback(LobbyMatchList_t* inCallback,
bool inIOFailure)
{
  if(inIOFailure) {return;}
```

```
  // если фойе найдено, войти, иначе – создать его
  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);
  }
}
```

Функция `NetworkManager::EnterLobby` ничем особенным не примечательна, кроме того, что вызывает другую функцию в `NetworkManager` с именем `UpdateLobbyPlayers`. Функция `UpdateLobbyPlayers` вызывается, когда игрок впервые входит в фойе и когда другой игрок входит или покидает фойе. Благодаря этому `NetworkManager` всегда имеет актуальный список игроков, находящихся в фойе. Это важно, поскольку после удаления пакетов представления данная функция остается единственным средством узнать об изменении списка игроков в фойе.

Чтобы гарантировать вызов `UpdateLobbyPlayers` всегда, когда изменяется состав игроков в фойе, необходимо обеспечить вызов универсальной функции. Разница между универсальными функциями обратного вызова и результатами вызовов в том, что результаты вызовов связаны с конкретными асинхронными вызовами, тогда как универсальные функции обратного вызова — нет. То есть универсальные функции обратного вызова можно рассматривать как способ подписки на уведомления о конкретных событиях. Всякий раз, когда пользователь покидает фойе или входит в него, происходит обратный вызов. Объявление универсальных функций обратного вызова выполняется с помощью макроса `STEAM_CALLBACK` внутри класса, откликающегося на обратный вызов. В данном случае — это класс реализации, а применение макроса выглядит так:

```
// Обратный вызов, когда пользователь покидает/входит в фойе
STEAM_CALLBACK(Impl, OnLobbyChatUpdate, LobbyChatUpdate_t,
        mChatDataUpdateCallback);
```

Этот макрос упрощает объявление имен функций обратного вызова и соответствующих им переменных-членов. Переменные-члены должны быть перечислены в списке инициализации `GameServices::Impl`, как показано ниже:

```
mChatDataUpdateCallback(this, &Impl::OnLobbyChatUpdate),
```

Реализация `OnLobbyChatUpdate` вызывает функцию `UpdateLobbyPlayers` экземпляра `NetworkManager`. Таким способом гарантируется вызов `UpdateLobbyPlayers` каждый раз, когда игрок входит или покидает фойе. Так как функции `UpdateLobbyPlayers` необходимо получать массив с идентификаторами и именами игроков в игре, класс `GamerServices` предоставляет функцию `GetLobbyPlayerMap`, представленную в листинге 12.5.

Листинг 12.5. Создание массива всех игроков в фойе

```
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()));
    }
  }
}
```

На случай, если понадобится поддержка обмена текстовыми сообщениями между игроками в фойе, Steamworks предоставляет функцию `SetLobbyChatMsg`, осуществляющую передачу сообщений. В паре к ней предоставляется обратный вызов `LobbyChatMsg_t`, который необходимо зарегистрировать, чтобы получать уведомления о получении новых сообщений. Так как в «RoboCat» отсутствует интерфейс для обмена сообщениями, эта функциональность не была включена в класс `GamerServices`. Однако чтобы добавить необходимые функции-обертки, потребуется совсем немного времени.

Когда все будет готово к запуску, в игре с топологией «клиент-сервер» следует вызвать функцию `SetLobbyGameServer`, связывающую конкретный сервер с фойе. Связь можно установить по IP-адресу (для выделенного сервера) или по идентификатору Steam (для сервера, запущенного на компьютере игрока). Это приведет к обратному вызову `LobbyGameCreated_t` у всех игроков, который сообщит им, что пришло время подключиться к серверу.

Но так как «RoboCat RTS» — игра с топологией «точка-точка», данная функциональность в ней не используется. Вместо этого, когда все будет готово к запуску, игра выполняет три шага. Сначала фойе закрывается для приема новых игроков, и никакие другие игроки не могут войти в него. Затем осуществляется обмен информацией между узлами с целью синхронизировать начало игры. Наконец, сразу после запуска игры, все игроки покидают фойе. Когда последний игрок оставит фойе, оно автоматически уничтожается. Функции, закрывающие фойе и освобождающие его, объявлены в `GamerServices` с именами `SetLobbyReady` и `LeaveLobby`. Эти функции являются очень тонкими обертками, вызывающими соответствующие функции Steamworks.

Сетевые взаимодействия

Многие игровые службы также предоставляют обертки для сетевых взаимодействий между двумя пользователями, подключенными к службе. Steamworks, например, предоставляет группу функций для передачи пакетов другим игрокам. Класс `GamerServices` включает обертки для некоторых из них, как показано в листинге 12.6.

Листинг 12.6. Сетевые взаимодействия «точка-точка» с помощью 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;
}
```

Обратите внимание, что ни в одной из этих сетевых функций не упоминаются IP-адреса сокетов. Это объясняется тем, что Steamworks позволяет посылать пакеты конкретным пользователям только по их идентификаторам, а не по IP-адресам. На то есть две причины. Во-первых, такое решение дает дополнительную защиту, потому что IP-адреса остаются неизвестными другим пользователям. Во-вторых, что, пожалуй, самое важное, это позволяет Steam независимо осуществлять преобразование сетевых адресов. Как уже говорилось в главе 6, одна из проблем, связанных с прямыми ссылками на адреса сокетов, заключается в том, что адрес может принадлежать другой сети. Однако при использовании сетевых функций Steamworks эта проблема целиком решается самой службой Steam. Мы просто посылаем пакет определенному пользователю, а Steam пытается послать данные этому пользователю через NAT, если это возможно. Если маршрутизатор с NAT оказывается непреодолимым препятствием, Steam будет использовать сервер-ретранслятор. Это гарантирует достижимость всех пользователей, подключенных к Steam.

Дополнительно Steamworks поддерживает несколько разных режимов передачи. В игре «RoboCat RTS» нет второстепенной информации, поэтому все пакеты пересылаются через надежные соединения, как это можно заметить по параметру `k_EP2PSendReliable`. Этот режим позволяет пересылать блоки данных объемом до 1 Мбайт с автоматической фрагментацией пакетов и сборкой их на стороне получателя. Однако существует возможность использовать и UDP-подобные взаимодействия, для чего достаточно передать параметр `k_EP2PSendUnreliable`. Также поддерживаются режимы ненадежной передачи при наличии установленного соединения и надежной передачи, осуществляющей буферизацию с применением алгоритма Нейгла.

Когда передача пакета определенному пользователю выполняется с помощью `SendP2PPacket` впервые, на его доставку может потребоваться до нескольких секунд, потому что службе Steam нужно некоторое время, чтобы проложить маршрут между отправителем и получателем. Кроме того, когда получатель принимает пакет от нового пользователя, он также должен принять запрос на открытие сеанса от отправителя. Это делается с целью предотвратить получение нежелательных пакетов от определенного пользователя. Чтобы принять запрос на открытие сеанса, каждый раз, когда такой запрос поступает, вызывается функция обратного вызова. Аналогично, когда попытка открыть сеанс завершается неудачей, вызывается другая функция обратного вызова. Реализация обеих функций в игре «RoboCat» представлена в листинге 12.7.

Листинг 12.7. Функции обработки событий открытия сеанса в топологии «точка-точка»

```
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)
{
  // связь с игроком потеряна, сообщить об этом диспетчеру сети
  NetworkManager::sInstance->HandleConnectionReset(
    inCallback->m_steamIDRemote.ConvertToUint64());
}
```

Чтобы исправить ситуацию, когда передача первого пакета занимает некоторое время, процедура инициализации в игре «RoboCat» была немного изменена. Когда на ведущем узле все будет готово к запуску игры, игрок нажимает клавишу Enter. Однако вместо того, чтобы начать обратный отсчет, `NetworkManager` входит в новый режим «готовности». Находясь в этом режиме, он посылает пакет всем остальным узлам. В свою очередь каждый узел, получив пакет готовности к началу, рассылает свой пакет готовности всем остальным узлам. Благодаря этому все узлы открывают сеансы друг с другом до начала игры.

Когда ведущий узел получает пакеты готовности от всех остальных узлов, он входит в состояние «запуск» и рассылает пакеты запуска игры, как и прежде. Следует отметить, что без промежуточного состояния готовности узлы не откроют сеансы друг с другом до начала игры, и тогда на доставку пакетов с нулевым ходом может уйти до нескольких секунд, а это означает, что каждый игрок заметит задержку в начале игры.

Для включения нового сетевого кода обработка пакетов в `NetworkManager` была полностью переписана в этой версии «RoboCat». Теперь вместо класса `UDPSocket` для обработки пакетов используются функции, предоставляемые классом `GamerServices`.

Статистика игрока

Популярной особенностью игровых служб является возможность получения различных статистик. Благодаря ей можно посмотреть свои показатели или показатели своего друга, чтобы оценить успехи, которых удалось достичь в той или иной игре. Обычно существует возможность запросить статистику игрока у сервера, а также изменить ее и записать новые значения на сервер. Несмотря на существующую возможность читать статистики непосредственно с сервера и записывать их на сервер, мы рекомендуем кэшировать их в памяти на стороне игрока. Именно такой подход реализован в классе `GamerServices`.

В играх на основе Steamworks имена и типы значений статистик определяются для конкретного идентификатора приложения на сайте Steamworks. Поскольку в коде примера для этой главы используется идентификатор игры «SpaceWar», мы можем использовать только статистики, которые в свое время были определены для игры «SpaceWar». Но сама реализация поддержки статистики будет работать с любым набором статистик, для этого вам понадобится только изменить определения статистик.

Служба Steam поддерживает три типа статистик. Целочисленные и вещественные статистики хранят целые и вещественные числа. Третий тип статистик называется «average rate» (средняя скорость). Для статистик этого типа определяется размер скользящего окна для усреднения. При выполнении чтения этой статистики с сервера возвращается единственное вещественное значение. Но когда выполняется ее изменение, нужно указать значение и интервал времени, в течение которого это значение было достигнуто. Затем служба Steam автоматически вычисляет новое скользящее среднее. С таким подходом можно существенно влиять на статистики, такие как «число золотых монет в час», изменяя их с ростом мастерства игрока, даже при том что пользователь мог вступить в игру много часов назад.

При определении статистик для игры на сайте Steamworks, кроме всего прочего, необходимо заполнить строковое свойство «API Name». Все функции SDK, осуществляющие чтение или запись конкретных статистик, будут требовать передачи этой строки соответствующей статистике. С этой целью можно добавить в `GamerServices` функции для работы со статистиками, принимающие строку как параметр. Однако проблема в том, что для реализации такого подхода требуется помнить точные

имена API для всех статистик, к тому же всегда существует вероятность допустить банальную опечатку. Кроме того, поскольку статистики хранятся в локальном кэше, каждый запрос к локальному кэшу может потребовать выполнения определенного вида хеширования. Обе эти проблемы легко решаются использованием перечисления для определения всех возможных статистик.

Один из возможных вариантов: определить перечисление, а затем отдельно определить массив с именами API для каждого значения в перечислении. Но проблема в том, что при изменении определения статистики нужно не забыть изменить перечисление и массив строк. Кроме того, если какие-то фрагменты игры написаны на языке сценариев, придется просмотреть и эти фрагменты, потому что где-то в сценариях может потребоваться переопределить те же самые перечисления. Необходимость помнить о своевременном обновлении кода во всех трех местах может стать причиной ошибок и раздражения.

К счастью, существует интересный прием, основанный на особенностях работы препроцессора C++. Он называется *X-макрос* (X macro) и позволяет определять статистики в одном месте. Эти определения затем автоматически используются везде, где необходимо, и гарантируют синхронизацию. X-макрос полностью устраняет вероятность появления ошибок из-за изменения статистик, поддерживаемых игрой.

Чтобы реализовать X-макрос, сначала нужно создать файл с определениями всех элементов и любых дополнительных свойств этих элементов. В данном случае определения хранятся в отдельном файле `Stats.def`. Каждая статистика определяется двумя компонентами: ее именем и типом данных. Определения статистик могли бы выглядеть так, как показано ниже:

```
STAT(NumGames,INT)
STAT(FeetTraveled,FLOAT)
STAT(AverageSpeed,AVGRATE)
```

Далее в `GamerServices.h` добавляются два перечисления, связанные со статистиками. Одно из них, `StatType`, ничем не примечательно. Оно просто определяет три поддерживаемых типа статистик: `INT`, `FLOAT` и `AVGRATE`. Другое перечисление, `Stat`, намного сложнее, потому что использует X-макрос. Это перечисление приводится в листинге 12.8.

Листинг 12.8. Объявление перечисления Stat с использованием X-макроса

```
enum Stat
{
  #define STAT(a,b) Stat_##a,
  #include "Stats.def"
  #undef STAT
  MAX_STAT
};
```

Здесь сначала определяется макрос с именем `STAT`, принимающий два параметра. Обратите внимание, что это число соответствует числу параметров в каждой записи в файле `Stats.def`. В данном случае второй параметр полностью игнорируется макросом. Это объясняется тем, что для данного перечисления тип статистики не

имеет никакого значения. Далее используется оператор ## препроцессора, чтобы объединить символы содержимого первого параметра с приставкой Stat_. Затем подключается файл Stats.def — эта инструкция просто копирует содержимое файла Stats.def в объявление перечисления. Поскольку в этот момент макрос STAT уже определен, он замещается результатом его развертывания. Так, например, первый элемент перечисления будет определен как Stat_NumGames, потому что именно в эту последовательность символов будет развернуто макроопределение STAT(NumGames,INT).

В заключение аннулируется определение макроса STAT и определяется последний элемент перечисления MAX_STAT. То есть трюк с X-макросом не только помог определить все члены перечисления, соответствующие статистикам, перечисленным в Stats.def, но также добавил элемент, отражающий общее число статистик.

Вся прелесть применения X-макроса в том, что ту же идиому можно использовать всюду, где необходим список статистик. При таком подходе после изменения определений в Stats.def достаточно просто перекомпилировать код, чтобы запустить волшебство макросов и обновить весь код, зависящий от этих определений. Кроме того, так как содержимое Stats.def имеет очень простую структуру, его легко можно было бы проанализировать в сценариях, если они используются в игре.

Прием X-макрос используется также в объявлении массива статистик, в файле реализации. Сначала объявляется структура StatData, представляющая кэшированные значения отдельных статистик. Чтобы не усложнять реализацию, в StatData включены элементы для хранения всех трех статистик — целочисленной, вещественной и среднего значения скорости, как показано в листинге 12.9.

Листинг 12.9. Структура StatData

```
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)
  { }
};
```

Далее в классе GamerServices::Impl имеется массив-член, объявленный как

```
std::array<StatData, MAX_STAT> mStatArray;
```

Обратите внимание, что в определении массива присутствует значение MAX_STAT, которое изменяется автоматически и определяет число элементов в массиве.

Наконец, в списке инициализации конструктора GamerServices::Impl в игру вступает X-макрос. С его помощью конструируется каждый элемент StatData в массиве mStatArray, как показано в листинге 12.10.

Листинг 12.10. Инициализация mStatArray с помощью X-макроса

```
mStatArray({
  #define STAT(a,b) StatData(#a, StatType::##b),
  #include "Stats.def"
  #undef STAT
} ),
```

Этот второй X-макрос использует оба элемента из макроопределения STAT. Первый элемент с помощью оператора # преобразуется в строковый литерал, а второй — в соответствующий элемент перечисления StatType. Так, например, определение STAT(NumGames,INT) будет преобразовано в экземпляр StatData:

```
StatData("NumGames", StatType::INT),
```

Прием X-макрос также используется для определения списка наград и таблицы рекордов, поскольку они состоят из множества элементов, требующих синхронизации с определениями статистик. С другой стороны, несмотря на удобство, этим приемом не следует злоупотреблять, потому что он делает программный код трудно читаемым. И тем не менее это очень полезный инструмент, который должен быть в арсенале разработчика на случаи, подобные описанному выше.

После реализации X-макроса остальная часть кода для обработки статистик становится на свое место относительно просто. В GamerServices имеется защищенная функция RetrieveStatsAsync, которая вызывается во время инициализации объекта GamerServices. Когда сервер присылает статистику, Steamworks выполняет обратный вызов. Обе эти функции представлены в листинге 12.11. Обратите внимание, что в OnStatsReceived используются не «жестко зашитые» определения статистик, а информация из массива mStatsArray, который автоматически генерируется X-макросом. Кроме того, для нужд отладки код записывает в журнал значения статистик при первой загрузке.

Листинг 12.11. Извлечение статистик из сервера Steam

```
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)
  {
```

```
    // загрузить статистики
    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
      {
        // в ответ на запрос средней скорости возвращается
        // единственное вещественное значение
        SteamUserStats()->GetStat(stat.Name, &stat.FloatStat);
        LOG("Stat %s = %f", stat.Name, stat.FloatStat );
      }
    }

    // загрузить награды
    //...
  }
}
```

В классе `GamerServices` имеются также функции для извлечения и изменения значений статистик. Функция чтения возвращает копию значения из локального кэша. Функция записи изменяет значение в локальном кэше и посылает его на сервер. Таким способом гарантируется синхронизация статистик в локальном кэше и на сервере. Реализации `GetStatInt` и `AddToStat` для целочисленных значений представлены в листинге 12.12. Запись вещественных значений и средней скорости выполняется аналогично, однако, как упоминалось выше, когда изменяется средняя скорость, серверу посылается два значения.

Листинг 12.12. Функции GetStatInt и AddToStat

```
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)
{
  // Проверить готовность статистики
  //...
```

```
  StatData& stat = mImpl->mStatArray[inStat ];

  // Это целочисленная статистика?
  //...
  stat.IntStat += inInc;
  SteamUserStats()->SetStat(stat.Name, stat.IntStat );
}
```

В настоящее время «RoboCat RTS» использует статистики для слежения за числом уничтоженных вражеских котов, а также за числом своих потерянных котов. Код, обновляющий статистики, находится в `RoboCat.cpp`. Подобный подход, когда код обновления статистик разбросан повсюду, где он необходим, широко используется в играх, поддерживающих статистики.

Награды игрока

Другой популярной особенностью игровых служб является поддержка наград. После выполнения некоторых опасных заданий игроку присуждается награда. Награды могут присуждаться за однократные события, такие как уничтожение противника или победа в игре с определенным уровнем сложности. Из других наград можно назвать такие достижения, как накопленные значения статистик за некоторое время, например число побед в 100 матчах. Некоторые игроки настолько обожают награды, что стремятся получить их все.

В Steam награды обрабатываются точно так же, как статистики. Множество наград для конкретной игры определяется на сайте Steamworks, и по этой причине игра «RoboCat» может использовать только награды, которые были определены для «SpaceWar». По аналогии со статистиками реализация наград использует X-макросы. Определения наград находятся в файле `Achieve.def`, и на их основе конструируется соответствующее перечисление `Achievement`. Также имеются структура `AchieveData` и массив этих структур с именем `mAchieveArray`.

Функция `RequestCurrentStats` дополнительно запрашивает информацию о текущих наградах с сервера Steam. Соответственно, функция обратного вызова `OnStatsReceived` дополнительно выполняет кэширование наград. Достижения копируются в цикле, который вызывает функцию `GetAchievement` и передает ей логическое значение, чтобы указать, получена ли данная награда:

```
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);
}
```

Также имеется несколько простых оберток для определения состояния наград (получена или нет) и для фактического присуждения указанной награды. Как и в случае со статистиками, проверка состояния награды выполняется в локальном кэше, а функция присуждения изменяет признак в кэше и записывает его на сервер. Соответствующий код приводится в листинге 12.13.

Листинг 12.13. Проверка и присуждение наград

```
bool GamerServices::IsAchievementUnlocked(Achievement inAch)
{
  // Проверить готовность статистики
  //...
  return mImpl->mAchieveArray[inAch].Unlocked;
}

void GamerServices::UnlockAchievement(Achievement inAch)
{
  // Проверить готовность статистики
  //...
  AchieveData& ach = mImpl->mAchieveArray[inAch];

  // игнорировать уже присужденную награду
  if(ach.Unlocked) {return;}

  SteamUserStats()->SetAchievement(ach.Name);
  ach.Unlocked = true;
  LOG("Unlocking achievement %s", ach.Name);
}
```

Что касается присуждения наград: награду желательно присуждать сразу после того, как она заработана. Иначе игрок будет недоумевать, увидев, что условия присуждения награды выполнены, а награда не присвоена. С другой стороны, в многопользовательской игре может оказаться предпочтительнее ставить награды в очередь и присуждать их в конце матча. В этом случае игрок не будет введен в заблуждение сообщением на экране о присуждении награды.

Так как награды в «RoboCat RTS» выдаются исходя из числа уничтоженных в игре котов, соответствующий код, следящий за достижениями, был добавлен в функцию-член `TryAdvanceTurn` класса `NetworkManager`. Благодаря этому в конце каждого хода игра проверяет, заработал ли игрок награду.

Таблицы рекордов

Таблицы рекордов дают возможность определять места, занимаемые игроками, по различным аспектам игры, например по числу набранных очков или по времени, за которое был пройден некий уровень. Вообще говоря, занимаемое место можно определять в глобальной иерархии или относительно ваших друзей, также зарегистрированных в игровой службе. Таблицы рекордов в Steam создаются вручную, на веб-сайте Steamworks, или программно, с помощью функций SDK.

Так же как и в случае со статистиками и наградами, реализация `GamerServices` использует X-макрос для определения перечисления рекордов. В данном случае рекорды определяются в файле `Leaderboards.def`. Каждая запись в этом файле содержит название рекорда, признак порядка сортировки и описание значений, которые должны отображаться при просмотре таблицы в Steam.

Порядок получения рекордов несколько отличается от получения статистик или наград. Во-первых, в каждый момент времени можно получить только один рекорд. Когда рекорд будет найден, должен быть вызван код, извлекающий результаты.

То есть если в игре потребуется последовательно найти все рекорды, код, извлекающий результаты, должен запросить получение следующего рекорда, и этот процесс должен повторяться до тех пор, пока не будут найдены все рекорды. Как это делается, показано в листинге 12.14.

Листинг 12.14. Поиск всех рекордов

```
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;

    // загрузить следующий
    mCurrentLeaderFind++;
    if(mCurrentLeaderFind != MAX_LEADERBOARD)
    {
      GamerServices::sInstance->FindLeaderboardAsync(
        static_cast<Leaderboard>(mCurrentLeaderFind));
    }
    else
    {
      mAreLeadersReady = true;
    }
  }
}
```

Еще одно отличие заключается в том, что в процессе поиска рекордов не требуется загружать какие-либо записи из таблицы. Вместо этого сервер просто возвращает дескриптор рекорда. Если понадобится загрузить записи для отображения, следует вызвать функцию `DownloadLeaderboardEntries` из Steamworks SDK, передав ей дескриптор и параметры загрузки («все», «только друзья» и пр.). В результате этого позднее будет вызван обработчик с результатами, где можно организовать отображение таблицы рекордов. Аналогично выглядит процедура выгрузки рекордов с помощью функции `UploadLeaderboardScore`. Примеры использования этих двух функций можно найти в `GamerServicesSteam.cpp`.

Так как в игре «RoboCat» отсутствует пользовательский интерфейс для отображения рекордов, проверить работу этого механизма можно с помощью пары команд, предусмотренных для отладки. Нажатие клавиши F10 запустит процедуру выгрузки текущего значения счетчика уничтоженных врагов, а нажатие клавиши F11 загрузит список рекордов, в котором ваше значение счетчика будет находиться в середине. Попутно следует отметить, что нажатие F9 сбросит все достижения и статистики, связанные с идентификатором приложения (в данном случае с идентификатором игры «SpaceWar»).

Поддержка рекордов в Steam имеет одну интересную особенность: она позволяет пользователям выгружать вместе с рекордом дополнительную информацию. Например, рекордно быстрое преодоление уровня можно сопроводить скриншотом или видеозаписью, демонстрирующей прохождение. Или в игре, имитирующей гонки, может иметься фантом, которого пользователи могут загрузить и состязаться с ним. Эта поддержка открывает возможность сделать рекорды более интерактивными, чем простой список заработанных очков или баллов.

Другие службы

Эта глава охватывает множество разных аспектов Steamworks SDK, но еще больше остались нерассмотренными. Например, в SDK имеется поддержка облачных хранилищ, позволяющая игрокам синхронизировать сохраненное состояние игры между несколькими компьютерами. Предусмотрен пользовательский интерфейс для игры в режиме «большой картинки» (Big Picture Mode), спроектированный для пользователей с единственным контроллером. Имеется также поддержка микротранзакций и загружаемого содержимого (Downloadable Content, DLC).

На сегодняшний день в мире существует множество других игровых служб на выбор. Для пользователей устройств семейства PlayStation, таких как консоли PlayStation, PlayStation Vita и мобильные телефоны PlayStation, существует служба PlayStation Network. Для поддержки консолей Xbox была создана служба Xbox Live, которая также доступна для владельцев компьютеров с операционной системой Windows 10. В числе других служб можно назвать Apple Game Center для устройств на Mac/iOS и Googles Play Games Services для устройств на Android и iOS.

Некоторые игровые службы имеют особенности, характерные только для них. Например, Xbox Live поддерживает возможность сохранения групп игроков между разными играми и возможность начать новую игру всей группой. Также на консолях часто имеется стандартный пользовательский интерфейс, определяемый игровой службой. Например, выбор сохраненной локации на Xbox всегда должен производиться с использованием специализированного пользовательского интерфейса, который предоставляется через обращение к службе.

Спектр возможностей, предлагаемых игровыми службами, постоянно расширяется. Игроки ожидают, что эти возможности будут интегрированы в современные игры, поэтому, какую бы службу вы ни выбрали, подумайте, как лучше использовать ее для создания благоприятного впечатления у ваших игроков.

В заключение

Игровые службы дают игрокам широкий спектр возможностей. Некоторые службы привязаны к конкретной платформе, но для каждой платформы, такой как PC, существует множество вариантов выбора. Самой популярной, пожалуй, игровой службой для PC, Mac и Linux является Steam, и именно поэтому она использовалась для демонстрации приемов интеграции в этой главе.

Одно из важнейших решений, которые придется принимать перед интеграцией игровой службы, — определение способа организации программного кода, предназначенного для конкретной игровой службы. Это важно, потому что при последующем переносе игры на другую платформу может случиться так, что эта платформа не поддерживает первую игровую службу. Решить эту задачу можно при помощи идиомы указателя на реализацию.

Координация вступления в игру — важная функция, поддерживаемая большинством игровых служб. Она позволяет игрокам собраться вместе, чтобы начать игру. В играх на основе Steamworks игроки сначала находят фойе игры и входят в него. Когда все будет готовок к запуску игры, игроки соединяются с сервером (в топологии «клиент-сервер») или друг с другом (в топологии «точка-точка») и затем покидают фойе.

Игровые службы также часто предоставляют механизмы пересылки пакетов с данными другим пользователям. Это одновременно дает дополнительную защиту, освобождая игроков от необходимости раскрывать свои IP-адреса, и позволяет игровым службам самостоятельно выполнять операции по преодолению маршрутизаторов с NAT или ретрансляции пакетов. Сетевой код в игре «RoboCat RTS» был изменен так, чтобы отправка данных производилась исключительно средствами Steamworks SDK. Как дополнительное преимущество SDK предоставляет метод надежных взаимодействий. Поскольку доставка первого пакета пользователю может происходить с существенной задержкой из-за необходимости открывать сеанс, процедура начальной настройки в «RoboCat» была изменена, чтобы соединения между узлами устанавливались при переходе в состояние «готов» до того, как будет запущен обратный отсчет перед началом игры.

В числе других типичных особенностей игровых служб можно назвать поддержку статистики, наград и рекордов. Реализация статистик в классе `GamerServices` основана на объявлении всех возможных статистик во внешнем файле `Stats.def`. Эта информация затем используется во множестве мест с помощью X-макросов, благодаря чему гарантируется соответствие между перечислением и массивом, содержащим информацию о статистиках. Аналогичный подход использован в реализации наград и рекордов.

Вопросы для повторения

1. Опишите идиому «указатель на реализацию». Какие преимущества она дает? Какие недостатки имеет?
2. Какую роль играют обратные вызовы в Steamworks?

3. Опишите примерно, как действует процедура вступления в игру, используемая в Steamworks.
4. Какие преимущества с точки зрения сетевых взаимодействий дает игровая служба?
5. Опишите, как работает прием X-макрос. Какие преимущества и какие недостатки он имеет?
6. Реализуйте класс идентификатора пользователя `GamerServiceID` и используйте его как обертку для идентификатора Steam. Замените все ссылки на идентификатор игрока типа `uint64_t` ссылками на этот новый класс.
7. Реализуйте класс `GamerServicesSocket` в стиле класса `UDPSocket`, который будет использовать Steamworks SDK для отправки данных. Не забудьте предоставить возможность выбирать между надежными и ненадежными взаимодействиями. Измените `NetworkManager` так, чтобы он использовал этот новый класс.
8. Реализуйте меню для отображения статистик текущего пользователя, а затем реализуйте возможность просмотра списка рекордов.

Для дополнительного чтения

Apple, Inc. «Game Center for Developers». Apple Developer. *https://developer.apple.com/game-center/*. Проверено 28 января 2016.

Google. «Play Games Services». Google Developers. *https://developers.google.com/games/services/*. Проверено 28 января 2016.

Microsoft Corporation. «Developing Games — Xbox One and Windows 10». Microsoft Xbox. *http://www.xbox.com/en-us/Developers/*. Проверено 28 января 2016.

Sony Computer Entertainment America. «Develop». PlayStation Developer. *https://www.playstation.com/en-us/develop/*.[1] Проверено 28 января 2016.

Valve Software. «Steamworks». Steamworks. *https://partner.steamgames.com/*. Проверено 28 января 2016.

[1] Этот же сайт на русском языке: *https://www.playstation.com/ru-ru/develop/*, но содержит не все материалы, имеющиеся на оригинальном сайте. — *Примеч. пер.*

13 Облачный хостинг для выделенных серверов

Стремительное развитие и распространение облачных технологий позволило даже небольшим студиям приобретать облачный хостинг для своих выделенных серверов. Больше не нужно надеяться, что игроки с быстрыми подключениями к Интернету разместят серверы на своих компьютерах. В этой главе мы рассмотрим достоинства и недостатки, а также методы размещения игровых серверов в облаке.

Размещать или не размещать

В начале эпохи онлайн-игр размещение собственных выделенных серверов было непомерно сложной задачей, для решения которой требовалось приобрести и настроить огромное количество компьютерного «железа», организовать сетевую инфраструктуру и принять на работу технический персонал. Любое дело, связанное с использованием аппаратных средств, является весьма рискованным предприятием. Если вы переоценили количество потенциальных игроков, ваши стойки с серверами будут вхолостую обогревать окружающее пространство. А вот в случае недооценки игроки, оплатившие услуги, не смогут подключиться к игре из-за ограниченной вычислительной мощности или пропускной способности. И в результате, пока вы изо всех сил стараетесь приобрести новейшее оборудование, ваши игроки уходят от вас, пишут отрицательные отзывы и рекомендуют своим друзьям не играть в вашу игру.

Но ужас тех дней закончился. Изобилие предложений, гибкие цены, зависящие от арендуемой вычислительной мощности, предложения от гигантов услуг облачного хостинга, таких как Amazon, Microsoft и Google, — все это позволило игровым компаниям оперативно реагировать на приток или отток игроков. Сторонние службы, такие как Heroku и MongoLabs, еще больше упрощают развертывание, предоставляя услуги по управлению серверами и базами данных.

С исчезновением барьера входа на рынок каждый разработчик может рассмотреть возможность хостинга выделенных серверов независимо от величины компании, в которой он работает. Но, несмотря на отсутствие необходимости заранее выделять средства на приобретение серверов, услуги облачного хостинга имеют свои недостатки, о которых следует знать:

- **Сложность.** Содержание флотилии выделенных серверов намного более сложная задача, чем предоставление игрокам возможности самим развертывать игровые серверы. Даже при том, что облачный хостинг предлагает всю необходимую инфраструктуру и программное обеспечение для администрирования, вам все еще придется писать собственный код управления процессом и виртуальной машиной, как описывается ниже в этой главе. Кроме того, вы должны будете взаимодействовать с одним или несколькими поставщиками услуг облачного хостинга и своевременно адаптироваться под изменения в программных интерфейсах.
- **Стоимость.** Даже при том, что облачный хостинг значительно уменьшает авансовые затраты и затраты на перспективу, он все же не бесплатный. Увеличение интереса со стороны игроков может покрыть увеличившуюся стоимость, но так бывает не всегда.
- **Зависимость от третьих лиц.** Размещая свою игру на серверах Amazon или Microsoft, вы перекладываете все заботы об устранении простоев на плечи этих компаний. Несмотря на то что хостинговые компании предлагают подписать **соглашение об уровне обслуживания**, гарантирующее минимальную продолжительность работы, это едва ли утешит игроков, отдавших свои деньги, если все серверы внезапно станут недоступны.
- **Неожиданные изменения в аппаратном окружении.** Поставщики услуг хостинга обычно гарантируют предоставление аппаратных средств, соответствующих неким минимальным требованиям. Но это не мешает им менять аппаратуру без предупреждения при условии сохранения соответствия с упомянутыми минимальными требованиями. Если вдруг они включат в работу какую-то необычную аппаратную конфигурацию, которую вы не тестировали, это может вызвать проблемы.
- **Потеря причастности игроков.** На ранних этапах развития многопользовательских игр управление собственным игровым сервером было предметом гордости. Это давало игрокам возможность стать важной частью сообщества и рекламировать игры, которые они развертывали у себя. Даже сегодня эта культура все еще живет в огромном количестве серверов Minecraft, развернутых по всему миру. Неоспоримые преимущества причастности игроков утрачиваются, когда ответственность за работу серверов перемещается в облако.

Однако, несмотря на существенные недостатки, их перевешивают неоспоримые преимущества:

- **Надежность, масштабируемость, высокая пропускная способность серверов.** Исходящая пропускная способность закономерно занимает первое место, и нет никаких гарантий, что продвинутые игроки смогут нарастить мощность своих серверов, если вдруг кто-то еще захочет поиграть в вашу игру. Облачный

хостинг и хорошая программа управления сервером помогают регулировать вычислительную мощность сервера по мере необходимости.

- **Предотвращение мошенничества.** Если все серверы находятся под вашим контролем, вы сможете гарантировать установку на них неизмененных, законных версий игр. То есть все игроки будут действовать по одинаковым правилам, не зависящим от прихотей администраторов. Это обеспечит не только достоверность ранжирования и рекордов, но и постоянное продвижение игрока в игре, как, например, в «Call of Duty».
- **Ненавязчивая защита от копирования.** Зачастую игроки испытывают неприязнь к навязчивой защите от копирования и **управлению правами на цифровое содержимое** (Digital Rights Management, DRM). Однако в некоторых типах игр без DRM не обойтись, в первую очередь это касается тех игр, которые используют микротранзакции для получения дохода, как, например, «League of Legends». Ограничивая возможность размещения игры только хостингом компании, выделенные серверы фактически обеспечивают ненавязчивую форму DRM. Вам не придется распространять серверные исполняемые файлы среди игроков, что существенно усложнит запуск взломанных серверов, которые незаконно раскрывают цифровое содержимое. Это также позволит проверять учетные данные игроков.

Как разработчик многопользовательских игр вы, возможно, не лишены права выбора хостинга. Однако, учитывая большой вес мнения инженеров с разносторонними навыками, вы должны понимать все последствия вашего участия в таком выборе.

Важнейшие инструменты

Рассматривая возможность работы в новом окружении, в первую очередь обратите внимание на инструменты, созданные для этого окружения. Разработка серверного программного обеспечения — это быстро развивающаяся область со своим набором инструментов быстрой разработки. Существует множество языков, платформ и протоколов, позволяющих повысить эффективность труда разработчиков серверных компонентов. На момент написания этих строк наблюдалась определенная тенденция к использованию в службах REST API, формата JSON и Node.JS. Все это — гибкие и популярные инструменты разработки серверного программного обеспечения, и примеры в этой главе основаны на их использовании. Вы можете выбрать другие инструменты для разработки на сервере, размещенном в облаке, но основные идеи от этого не изменятся.

REST

Аббревиатура *REST* расшифровывается как *Representational State Transfer* (передача репрезентативного состояния). Интерфейс REST поддерживает парадигму, согласно которой все запросы к серверу должны быть самодостаточными и не должны полагаться на предыдущие или последующие запросы. http — протокол, лежащий

в основе Всемирной паутины, — является отличным примером воплощения этой идеи, и потому типичные REST API основаны на использовании http-запросов для хранения, извлечения и изменения данных на стороне сервера. Запросы посылаются с использованием наиболее типичных HTTP-методов GET и POST и менее типичных PUT, DELETE и PATCH. Несмотря на то что большинство авторов предлагают стандарты, точно определяющие структуру использования HTTP-запросов для квалификации интерфейса REST, многие инженеры в конечном итоге создают REST-подобные интерфейсы, прекрасно отвечающие потребностям пользователей, но не придерживающиеся в полной мере какого-либо набора требований REST. Строго говоря, интерфейсы REST должны использовать HTTP-методы в непротиворечивой манере: запросы GET — для извлечения данных, запросы POST — для создания новых компонентов данных, запросы PUT — для сохранения данных в определенном месте, запросы DELETE — для удаления данных и запросы PATCH — для непосредственного редактирования данных.

Одно из существенных достоинств интерфейсов REST в том, что обмен через них производится преимущественно текстовыми данными, то есть данными в читаемом виде, пригодными для анализа и отладки. Кроме того, они используют протокол HTTP, который, в свою очередь, использует транспортный протокол TCP, и потому они обеспечивают надежность взаимодействий. Самодостаточная природа запросов REST упрощает отладку и обосновывает выбор REST как основной модели API для современных облачных служб. Более подробную информацию о стиле интерфейсов REST и предложенных проектах стандартов REST можно найти в ресурсах, перечисленных в разделе «Для дополнительного чтения» в конце главы.

JSON

В конце 1990-х — начале 2000-х XML был объявлен универсальным форматом обмена данными, который изменит мир. И он начал изменять мир, но содержал слишком много угловых скобок, знаков «равно» и закрывающих тегов, чтобы удержаться на троне. В настоящее время все чаще используется JSON — новый универсальный формат обмена данными. Фактически формат JSON (расшифровывается как *JavaScript object notation* — форма записи объектов JavaScript) является подмножеством языка JavaScript. Объект, преобразованный в формат JSON, имеет именно тот вид, который необходим интерпретатору JavaScript, чтобы воссоздать его. Это текстовый формат, такой же легкочитаемый, как XML, но с меньшим количеством требований к оформлению и наличию закрывающих тегов, что делает его более простым для исследования и отладки. Кроме того, так как данные в этом формате являются допустимым кодом на языке JavaScript, их можно вставлять непосредственно в программы на JavaScript для отладки.

JSON прекрасно справляется с ролью формата данных для запросов REST. Указав HTTP-заголовок `Content-Type` со значением `application/json`, в запросах POST, PATCH или PUT можно передавать данные в формате JSON или возвращать их в ответ на запросы GET. JSON поддерживает все основные типы данных языка JavaScript, такие как логические, строковые, числовые значения и объекты.

Node.JS

Основанный на реализации Google V8 JavaScript, Node JS представляет собой открытый движок, предназначенный для реализации серверных служб на языке JavaScript. Выбор языка определялся желанием упростить разработку веб-сайтов с поддержкой AJAX, веб-интерфейс которых реализуется с применением JavaScript. Благодаря применению одного языка программирования на стороне сервера и клиента разработчики могут писать функции, которые легко переносятся между уровнями или совместно используются ими. Идея завоевала популярность, и вокруг проекта Node сплотилось обширное сообщество. Отчасти своим успехом движок обязан огромному числу открытых пакетов, созданных для него, их легко установить с помощью *диспетчера пакетов Node* (Node Package Manager, npm). Почти все известные службы с REST API включают обертки пакетов Node, упрощая взаимодействие с многочисленными поставщиками облачных услуг.

Сам движок Node реализует однопоточное окружение JavaScript, управляемое событиями. Цикл обработки событий действует в главном потоке выполнения — практически так же, как в видеоигре, — и передает входящие события соответствующим обработчикам. Эти обработчики, в свою очередь, выполняют продолжительные запросы к файловой системе или к внешним службам, таким как базы данных или серверы REST, которые выполняются как асинхронные задания в потоках выполнения, не связанных с JavaScript. Пока задания продолжают выполняться, управление возвращается в главный поток выполнения для обработки других входящих событий. Когда асинхронное задание завершается, оно посылает событие в главный поток выполнения, чтобы цикл событий смог вызвать соответствующий обработчик на JavaScript. Иначе говоря, Node предоставляет окружение, предотвращающее попадание в состояние «гонки» и вместе с тем обеспечивающее неблокирующее асинхронное поведение. Благодаря этим своим свойствам движок является главным кандидатом для реализации служб, обрабатывающих входящие запросы REST.

Node распространяется со встроенным сервером HTTP, но задачи декодирования входящих HTTP-запросов, заголовков и параметров и их маршрутизация между соответствующими функциями JavaScript обычно осуществляются одним из открытых пакетов Node, специализированных для этой цели. *Express JS* — один из таких пакетов, и именно он будет использоваться в примерах в этой главе. Ссылки на дополнительную информацию о пакете Express JS и движке Node JS можно найти в списке ресурсов в разделе «Для дополнительного чтения».

Обзор и терминология

С точки зрения игрока, запуск серверного процесса в облаке должен происходить незаметно для него. Когда игрок желает вступить в игру, клиент игрока запрашивает информацию об игре в конечной точке службы координации. Конечная точка выполняет поиск и, если не находит запрошенную игру, должна каким-то образом запустить новый сервер. После этого она возвращает клиенту IP-адрес и номер порта нового экземпляра сервера. Далее клиент автоматически подключается к указанному адресу, и игрок вступает в игру.

Может показаться заманчивым объединить процедуру координации с развертыванием выделенного сервера в один функциональный модуль. Это уменьшит объем кода и данных и даже немного увеличит производительность. Однако мы рекомендуем разделить их, хотя бы потому, что у вас может возникнуть потребность внедрить в свою систему выделенных серверов одно или несколько сторонних решений координации. Но даже если студия поддерживает свои собственные выделенные серверы, это еще не означает, что нельзя воспользоваться преимуществами сторонних решений координации, таких как Steam, Xbox Live или PlayStation Network. В некоторых случаях (в зависимости от платформы, для которой ведется разработка) это может оказаться обязательным условием. Поэтому мы рекомендуем держать модуль развертывания сервера отдельно от вашего модуля координации.

После того как система развертывания завершит запуск нового сервера, она должна зарегистрировать его в системе координации так, как это сделал бы игрок, разместивший игровой сервер у себя. После этого система координации сможет подключать игроков к экземплярам сервера, а ваша облачная система развертывания сможет сосредоточиться на регулировании производительности экземпляров игры по мере необходимости.

Экземпляр игрового сервера

Прежде чем продолжить обсуждение, давайте конкретизируем термин «сервер», имеющей в наше время слишком широкий спектр значений. Иногда под термином «сервер» подразумевается экземпляр класса в программном коде, моделирующий единственную истинную версию игрового мира и передающий его состояние клиентам. Иногда под ним подразумевается процесс, ожидающий входящих запросов на соединение и содержащий упомянутый выше экземпляр класса. Но в некоторых случаях под ним подразумевается физическое «железо», на котором выполняется некий процесс, как, например, во фразе: «подсчитать, сколько серверов можно разместить в этой стойке».

Чтобы избежать путаницы, термин *серверный экземпляр игры*, или просто *экземпляр игры*, в этой главе будет использоваться для обозначения сущности, моделирующей игровой мир и рассылающей его состояние клиентам. Идея заключается в том, чтобы обозначить единственную реальность, используемую группой игроков для совместной игры. Если ваша игра поддерживает возможность участия в битве до 16 игроков, тогда серверный экземпляр игры будет запускаться для моделирования битвы с 16 игроками. В «League of Legends» на уровне «Summoner's Rift» в игре состязаются команды 5 на 5. В терминах координации это *единственный матч* (битва).

Игровой серверный процесс

Экземпляр игры не может существовать в пустоте. Он живет в серверном игровом процессе, который обновляет его, управляет клиентами, взаимодействует с операционной системой и делает все остальное, что обычно делают процессы. Это — воплощение игры в операционной системе. Во всех предыдущих главах понятия игрового серверного процесса и экземпляра игры не отделялись друг от друга,

потому что между ними было прямое соответствие. Каждый серверный игровой процесс отвечал за обслуживание единственного экземпляра игры. Однако в мире хостинга выделенных серверов это соответствие меняется.

Правильно сконструированный код позволит единственному процессу управлять множеством экземпляров игры. При условии, что процесс обслуживает каждый экземпляр отдельно, присваивает каждому уникальный номер порта и не смешивает изменяемые данные между экземплярами, в одном процессе может благополучно сосуществовать множество игровых миров.

Обслуживание нескольких экземпляров одним процессом может оказаться эффективным решением для размещения множества игр, поскольку позволяет обеспечить совместное использование неизменяемых ресурсов большого объема, таких как геометрия столкновений, навигационные меши и данные для анимации. Когда экземпляры игры действуют в своих собственных процессах, каждый из них хранит свою копию этих данных, что может вызывать завышенные требования к объему памяти. Игры, поддерживающие одновременную работу нескольких экземпляров игры в одном процессе, также могут извлекать дополнительные выгоды из более полного управления планированием: выполняя итерации по экземплярам и обновлениям, они могут гарантировать более равномерное распределение времени между экземплярами. В случае с множеством процессов предоставить подобные гарантии практически невозможно, так как решение о том, когда и какие процессы запускать, принимает планировщик операционной системы. Это не является проблемой, но возможность более полного контроля иногда оказывается весьма кстати.

Существенные преимущества решения с поддержкой нескольких экземпляров могут показаться очень привлекательными, но имеющиеся недостатки такого подхода оказываются не менее существенными. Если произойдет авария в одном экземпляре, завершится весь процесс — со всеми действующими в нем экземплярами игры. Существует вероятность того, что отдельный экземпляр повредит общий, как предполагается неизменяемый, ресурс. Напротив, если каждый экземпляр игры действует в собственном выделенном процессе, поврежденный или аварийный экземпляр завершит только сам себя. Кроме того, процессы с единственными экземплярами игры проще поддаются обслуживанию и тестированию. Обычно инженерам, разрабатывающим серверный код, для тестирования и отладки достаточно единственного экземпляра игры. Если процесс поддерживает несколько экземпляров, а инженеры не запускают их, многие ветви в коде остаются неохваченными вниманием разработчиков. Хорошая группа тестеров с подробным и основательным планом тестирования исправит этот недостаток, но они не смогут заменить полный охват всех путей выполнения инженерами, ведущими разработку. По этим причинам на практике часто используется решение с одним экземпляром игры в серверном игровом процессе.

Игровой сервер — машина

Так же как экземпляр игры должен находиться в серверном игровом процессе, серверный игровой процесс должен выполняться на игровом сервере-машине, и так же как в одном процессе может действовать несколько экземпляров игры, на

одной машине может выполняться несколько процессов. Выбор числа процессов, действующих на одной машине, определяется требованиями к производительности конкретной игры. Для максимальной производительности можно запустить единственный процесс на машине. Это гарантирует, что все ресурсы, включая процессор, графический процессор и ОЗУ, будут отданы в полное распоряжение игровому процессу. Однако такое решение может оказаться слишком расточительным. На каждой машине действует также операционная система, а любая ОС является серьезным потребителем ресурсов.

Использование уникальной ОС для запуска единственного процесса, особенно если в этом процессе действует единственный экземпляр игры, может стать слишком дорогим удовольствием. К счастью, операционные системы поддерживают одновременную работу сразу нескольких процессов и защищают их память, препятствуя попыткам записи в неизменяемые ресурсы друг друга. В современной операционной системе весьма мало вероятно, что авария в одном процессе вызовет аварию в другом процессе на том же игровом сервере-машине. Поэтому с финансовой точки зрения правильнее запускать столько серверных игровых процессов на каждой серверной машине, сколько позволят требования к производительности. Затраты на настройку производительности сервера и увеличение ОЗУ могут многократно окупиться, если эти меры позволят запускать больше игровых процессов на одной и той же серверной машине.

ПОЧЕМУ ВИРТУАЛЬНЫЕ МАШИНЫ?

Может показаться странной необходимость упаковки выбранной вами операционной системы и игрового процесса в виртуальную машину только затем, чтобы получить услугу облачного хостинга. В действительности все объясняется просто: виртуальные машины — отличный способ для поставщиков услуг облачного хостинга распределить свои вычислительные мощности между клиентами. В Amazon один 16-ядерный компьютер может запускать четыре виртуальные машины с игрой «Call of Duty», каждая из которых требует 4 ядра. По запросу на снижение вычислительной мощности для «Call of Duty» в определенные периоды суток Amazon может остановить две из них, освобождая недоиспользуемые аппаратные ресурсы. Если от Electronic Arts приходит запрос запустить 8-ядерную машину «Sim City», Amazon может запустить ее на той же аппаратуре, что освободилась после остановки двух виртуальных машин «Call of Duty», и с максимальной выгодой использовать свои ресурсы.

Виртуальные машины также удобны для быстрого решения проблем с выходом аппаратуры из строя. Так как образы виртуальных машин содержат ОС и приложение как одно целое, поставщик услуг может быстро решить проблему, просто переместив виртуальные машины с одного физического сервера на другой.

Аппаратура

Серверной игровой машине в облаке не обязательно соответствует физическая аппаратная единица. Чаще *образами машин* являются *виртуальные машины* (Virtual Machine, VM), мощность которых может увеличиваться или уменьшаться по

желанию. Иногда виртуальная машина размещается в единственном экземпляре на физической машине, иногда ресурсы физической машины с 16 ядрами и более делят между собой несколько виртуальных машин. В зависимости от выбранного поставщика услуг облачного хостинга и вашего бюджета у вас может не оказаться вариантов выбора для размещения виртуальных машин. В случае невысокой арендной платы они почти наверняка будут делить между собой аппаратные ресурсы одной физической машины и могут приостанавливаться, когда не используются в течение некоторого времени. Это может сказаться на устойчивости рабочих характеристик. Более высокая арендная плата позволит вам указать желаемую аппаратную конфигурацию.

Локальный диспетчер серверных процессов

Системе облачного сервера необходим способ запускать серверные игровые процессы на игровых серверах-машинах и следить за ними. Серверные машины не могут просто запустить максимальное число игровых процессов во время загрузки и ожидать, что эти процессы будут работать все время, пока работает сама машина. Любой процесс может потерпеть аварию в любой момент, и с этого момента виртуальная машина будет недоиспользовать выделенные ей ресурсы. Кроме того, даже в самых тщательно спроектированных играх утечки памяти могут быть обнаружены непосредственно перед поставкой заказчику. Если дата поставки не может быть изменена, иногда приходится развертывать серверы, в которых через различные «отверстия» утекает до нескольких мегабайт. Чтобы небольшие утечки не накапливались и чтобы избежать проблем из-за некорректного завершения игры, принято останавливать и перезапускать серверные процессы в конце каждой битвы или матча, если это возможно.

Для корректного завершения серверного процесса виртуальной машине необходим способ создания его резервной копии. Ей также нужен способ настройки процесса исходя из типа игры, которую хотят начать игроки. А для этого надежной системе поддержки необходим механизм, посредством которого она сможет потребовать от данной серверной машины запустить серверный процесс, настроенный определенным образом. Возможно, ваша операционная система уже имеет встроенные средства для удаленного запуска процессов и мониторинга их состояния. Однако более кроссплатформенное и менее хрупкое решение заключается в создании *локального диспетчера серверных процессов* (Local Server Process Manager, LSPM).

Сам диспетчер LSPM — это процесс, назначением которого является прием команд извне, запуск серверных процессов согласно командам и мониторинг их состояния с целью постоянно иметь информацию о том, какие процессы выполняются на данной машине в текущий момент. В листинге 13.1 демонстрируются процедуры инициализации, запуска и завершения простого приложения node.js/express для управления локальными серверными процессами.

Листинг 13.1. Инициализация, запуск и завершение

```
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);

      // достигнуто максимальное число процессов?
      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)
  {
    // чтобы остановить процесс, нужно послать событие закрытия
    // и удалить процесс из списка
    process.child.kill();
    res.sendStatus(200);
  }
  else
  {
    res.sendStatus(404);
  }
});
```

Сначала диспетчер LSPM инициализирует некоторые глобальные переменные. `gProcesses` хранит карту всех процессов, управляемых диспетчером в настоящий момент, а `gProcessCount` — их число. Значения для переменных `gProcessPath` и `gMaxProcessCount` извлекаются из переменных окружения, поэтому они легко

настраиваются отдельно для каждой машины. `gMachineState` хранит состояние всей машины, как то: имеется ли место для запуска дополнительных процессов, полная ли она или находится на стадии остановки. Эта переменная хранит значения объекта `eMachineState`.

Диспетчер LSPM поддерживает создание новых процессов, посылая запрос POST конечной точке `/api/processes/`. В частности, если LSPM выполняется на локальной машине и прослушивает порт 3000, можно с помощью программы `curl` отправить веб-запрос и запустить новый процесс, настроенный на участие в игре четырех игроков, выполнив следующую команду:

```
curl -H "Content-Type: application/json" -X POST -d '{"params":{"maxPlayers":4}}'
http://127.0.0.1 :3000/api/processes
```

Как только диспетчер LSPM принимает этот запрос, он сначала проверяет, не находится ли он на этапе остановки и не запущено ли максимально возможное число процессов. Если все в порядке, диспетчер создает новый универсально уникальный идентификатор для ожидающего процесса и при помощи модуля `child_process` для Node JS запускает новый серверный игровой процесс. Через параметры командной строки он сообщает процессу уникальный идентификатор и параметры настройки, указанные запрашивающим.

Затем LSPM сохраняет запись о запущенном дочернем процессе в свою карту `gProcesses`. Переменная `state` здесь используется для хранения информации о состоянии процесса — запускается или выполняется. Переменная `lastHeartbeat` хранит момент времени, когда LSPM получил последнее известие от этого процесса, и к этой переменной мы еще вернемся в следующем разделе.

После регистрации процесса LSPM устанавливает обработчики событий, принимающие и регистрирующие любой вывод, поступающий от процесса. Также он устанавливает обработчик очень важного события `"close"`, который удаляет процесс из карты `gProcesses` и фиксирует все изменения в `gMachineState`.

Наконец, LSPM отвечает на запрос уникальным идентификатором процесса и информацией о количестве процессов, выполняющихся в настоящий момент. Не забывайте, что модель событий в Node является однопоточной, поэтому не стоит беспокоиться о состоянии гонки, изменяя переменную `gProcessCount` или карту `gProcesses` во время выполнения функции.

Получив копию уникального идентификатора процесса, запрашивающая программа сможет затем запросить сведения о процессе, послав GET-запрос конечной точке `/processes/:processUUID` (соответствующий код здесь не показан), или остановить его, послав POST-запрос конечной точке `/processes/:processUUID/kill`.

ВНИМАНИЕ В эксплуатационном окружении необходимо ограничить круг лиц, имеющих право запускать и останавливать серверы через диспетчер LSPM. Для этого можно определить «белый список» IP-адресов, с которых допускается отправка запросов диспетчеру, а затем отбрасывать все запросы, полученные не с этих адресов. Это не позволит злонамеренным игрокам посылать команды на запуск процессов диспетчеру LSPM. Как вариант, можно добавить в запрос маркер безопасности и проверять его наличие перед обработкой запроса. В любом случае нужно реализовать какую-то защиту, иначе ваша система поддержки постоянно будет находиться под угрозой разрушения.

Мониторинг процессов

После запуска процессов диспетчеру LSPM необходим способ их мониторинга. Эта задача решается за счет посылки процессами сигналов, сообщающих о работоспособности. Эти периодически посылаемые пакеты будут сообщать, что они продолжают действовать. Если в течение некоторого времени LSPM не получит сигнал от какого-то процесса, он предположит, что процесс остановился, завис, замедлился или в нем возникли другие недопустимые неполадки, и завершит этот процесс, как показано в листинге 13.2.

Листинг 13.2. Мониторинг процессов

```
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);
```

POST-запрос конечной точке `/processes/:processUUID/heartbeat` регистрируется как сигнал, сообщающий о работоспособности процесса с указанным идентификатором. Получив сигнал, диспетчер LSPM получает текущее время и обновляет время получения последнего сигнала для соответствующего процесса. После получения первого сигнала LSPM изменит состояние `starting` процесса на `running`, чтобы отразить тот факт, что процесс успешно запустился.

Функция `checkHeartbeat` перебирает в цикле все процессы, находящиеся под управлением LSPM, и проверяет, не истекло ли предельное время ожидания сигнала о работоспособности. Если процесс все еще находится в состоянии `starting`, возможно, что процедура инициализации протекает слишком медленно, поэтому функция дает процессу еще немного времени, чтобы завершить инициализацию и прислать первый сигнал. Если с момента получения последнего сигнала прошло времени больше, чем `gMaxRunningHeartbeat` секунд, значит, с серверным процессом произошло что-то непредвиденное. Для устранения проблемы LSPM пытается вручную остановить дочерний процесс, если он еще не остановлен. Когда процесс остановится, зарегистрированное событие закрытия удалит его из списка процессов. Диспетчер LSPM вызывает функцию `checkHeartbeat` каждые `gHeartbeatCheckPeriod` миллисекунд при помощи таймера, устанавливаемого вызовом `setInterval` в конце сценария.

Чтобы сообщить диспетчеру LSPM о своей работоспособности, каждый процесс должен посылать POST-запросы конечной точке `heartbeat` не реже одного раза в `gHeartbeatCheckPeriod` секунд. Чтобы послать запрос REST-службе из программы на C++, нужно сконструировать HTTP-запрос в виде строки и послать ее в соответствующий порт диспетчера LSPM с помощью класса `TCPSocket`, описанного в главе 3. Например, если LSPM, прослушивающий порт 3000, запустил процесс с идентификатором `-processUUID`, равным `49b74f902d9711e5-8de0f3f32180aa49`, то процесс сможет сообщать о своей работоспособности, посылая следующую строку в порт 3000 по протоколу TCP:

```
POST /api/processes/49b74f902d9711e5-8de0f3f32180aa49/heartbeat HTTP/1.1\r\n\r\n
```

Обратите внимание на две последовательности символов в конце запроса, обозначающих конец строки, — они обозначают конец HTTP-запроса. Ссылки на дополнительную информацию о текстовом представлении HTTP-запросов можно найти в разделе «Для дополнительного чтения». Как вариант, можно воспользоваться готовым сторонним решением REST для C++, например открытой и кроссплатформенной библиотекой C++ REST SDK от Microsoft. Листинг 13.3 демонстрирует, как послать сигнал с помощью C++ REST SDK.

Листинг 13.3. Посылка извещения о работоспособности с помощью 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());
}
```

Чтобы проверить результат отправки сигнала, добавьте задачи продолжения в задачу, которая вернула вызов запроса. C++ REST SDK предоставляет библиотеку с богатыми функциональными возможностями, которые реализуют не только функции асинхронной отправки HTTP-запросов, основанные на задачах, но и некоторые серверные функции, парсинг формата JSON, поддержку WebSocket и многое другое. За дополнительной информацией о библиотеке C++ REST SDK и ее возможностях обращайтесь к ресурсам, указанным в разделе «Для дополнительного чтения».

ПРИМЕЧАНИЕ Запросы REST не единственный способ отправки сигналов о работоспособности диспетчеру LSPM. LSPM может открыть порт TCP или даже UDP непосредственно в Node, после чего серверный процесс сможет посылать очень маленькие пакеты-сигналы без накладных расходов, которые влечет за собой применение протокола HTTP. Игра может просто записывать некоторые данные в свой файл журнала, а диспетчер LSPM — проверять их. Однако, учитывая, что игра почти наверняка будет использовать REST API для взаимодействий с одной или несколькими другими службами, простоту отладки REST-данных, а также тот факт, что LSPM все равно должен принимать входящие запросы REST, отправка сигналов через REST API позволит снизить общую сложность реализации.

Диспетчер виртуальных машин

Упрощая удаленный запуск и мониторинг произвольного числа процессов на виртуальной машине, диспетчер LSPM решает значительную часть проблем, связанных с облачным хостингом. Однако он не содержит средств для поддержки самих виртуальных машин. Поэтому нам необходим *диспетчер виртуальных машин* (Virtual Machine Manager, VMM). Диспетчер VMM отвечает за мониторинг всех диспетчеров LSPM, отправку запросов диспетчерам LSPM на запуск игровых процессов и за запуск и остановку целых виртуальных машин вместе с соответствующими им диспетчерами LSPM.

Чтобы запустить новую виртуальную машину с использованием средств, предоставленных поставщиком услуг облачного хостинга, диспетчер VMM должен определить, какое программное обеспечение запустить на машине. Делается это путем передачи ссылки на *образ виртуальной машины* (Virtual Machine Image, VMI). Образ VMI представляет содержимое диска, с которого виртуальная машина должна загрузиться. Он содержит ОС, выполняемые файлы процессов и любые сценарии инициализации для запуска во время загрузки. Каждый поставщик услуг облачного хостинга имеет свой формат VMI и, обычно, свой комплект инструментов для создания виртуальных машин. В ходе подготовки к управлению виртуальными

машинами создайте образ VMI со своей ОС, скомпилируйте исполняемые файлы игры, которые будут действовать на сервере, подготовьте диспетчер LSPM и все необходимые файлы ресурсов.

ПРИМЕЧАНИЕ Даже при том, что каждый поставщик услуг облачного хостинга поддерживает свой формат образов VMI, в скором времени многие могут перейти на стандартизованный формат Docker Container. Ссылки на более подробную информацию о стандарте Docker смотрите в разделе «Для дополнительного чтения».

Порядок запуска виртуальной машины из образа VMI определяется поставщиком услуг. Обычно для этой цели поставщики предоставляют REST API с обертками на распространенных языках программирования, таких как JavaScript и Java. Поскольку вы можете сменить поставщика услуг или пользоваться услугами разных поставщиков в разных регионах, нелишним будет скрыть за абстракцией детали взаимодействия с API провайдеров от реализации VMM.

В дополнение к простому запуску новых виртуальных машин, когда это потребуется, диспетчер VMM должен иметь возможность запускать на каждой виртуальной машине новые процессы посредством диспетчеров LSPM. Кроме того, он должен иметь возможность «попросить» поставщика услуг остановить и удалить неиспользуемую виртуальную машину. Наконец, в его функции входит осуществление мониторинга состояния всех управляемых им виртуальных машин с целью исключения любых утечек в случае ошибки. Несмотря на то что среда Node является однопоточной, асинхронные взаимодействия между диспетчером VMM и диспетчером LSPM открывают широкие возможности для появления состояния гонки. Кроме того, даже при известной надежности протокола TCP каждый REST-запрос осуществляется через собственное соединение, то есть взаимодействия могут протекать неупорядоченно. В листинге 13.4 приводятся структуры данных и процедура инициализации VMM.

Листинг 13.4. Структуры данных и инициализация VMM

```
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;
    }
  }
}
```

Основные данные диспетчера VMM хранятся в двух ассоциативных массивах. Ассоциативный массив `gVMs` хранит информацию обо всех виртуальных машинах, находящихся в текущий момент под управлением диспетчера VMM. Ассоциативный массив `gAvailableVMs` — это подмножество виртуальных машин, доступных для запуска новых процессов. Это означает, что в настоящий момент эти машины не находятся на стадии остановки или запуска, не запускают процесс и число процессов в них не достигло максимума. Каждый объект, представляющий виртуальную машину, имеет следующие члены:

- `machineState`. Представляет текущее состояние виртуальной машины, хранит один из членов объекта `eMachineStates`. Эти состояния являются надмножеством `eMachineStates`, используемых диспетчером LSPM, и включают дополнительные состояния, имеющие отношение только к диспетчеру VMM.
- `uuid`. Уникальный идентификатор виртуальной машины, присвоенный диспетчером VMM. Запуская новую виртуальную машину, диспетчер VMM передает идентификатор диспетчеру LSPM, чтобы тот мог включать его в любые пакеты, посылаемые диспетчеру VMM.
- `url`. Хранит IP-адрес и номер порта диспетчера LSPM в виртуальной машине. IP-адрес и порт могут присваиваться поставщиком услуг облачного хостинга в момент создания виртуальной машины. Диспетчер VMM должен сохранить их, чтобы иметь возможность взаимодействовать с соответствующим диспетчером LSPM.
- `lastHeartbeat`. Подобно тому как LSPM принимает сигналы процессов о работоспособности, диспетчер VMM принимает такие же сигналы от диспетчеров LSPM. В этом поле сохраняется время получения последнего сигнала.
- `lastSequenceIndex`. Так как для каждого запроса REST создается собственное TCP-соединение, есть вероятность, что сообщения через них будут доставляться получателю не по порядку. Чтобы диспетчер VMM мог игнорировать любые устаревшие сообщения от LSPM, диспетчер LSPM отмечает пакеты последовательно увеличивающимися числами, а диспетчер VMM игнорирует любые входящие данные с порядковым числом, меньшим чем `lastSequenceIndex`.

- `cloudProviderId`. Хранит идентичность виртуальной машины, как ее определил поставщик услуг. Диспетчер VMM использует это значение, когда посылает поставщику запрос на удаление виртуальной машины.

Если возникает необходимость запустить новую виртуальную машину, функция `getFirstAvailableVM` находит первую виртуальную машину в карте `gAvailableVMs` и возвращает ее. Функция `updateVMState` отвечает за включение виртуальных машин в карту `gAvailableVMs` и исключение их оттуда, а также за изменение их состояния. Для поддержания целостности диспетчер VMM должен изменять значение `state` виртуальной машины только с помощью функции `updateVMState`. В листинге 13.5 показаны все необходимые структуры данных и обработчик конечной точки REST, который фактически запускает процесс. Предварительно он запускает новую виртуальную машину, если это необходимо.

Листинг 13.5. Запуск процесса и виртуальной машины

```
router.post('/processes/', function(req, res)
{
  var params = req.body.params;
  var vm = getFirstAvailableVM();
  async.series(
  [
    function(callback)
    {
      if(!vm ) // запустить, если необходимо
      {
        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);
      }
    },
    // виртуальная машина действует и находится в состоянии ожидания
```

```
    // поэтому никто другой не может обратиться к ней
    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); // неудача — вероятно, полная
            }
          }
          else
          {
            callback("seq# out of order: can't trust state");
          }
        }
        else
        {
          callback("error from lspm: " + error);
        }
      });
    }
  ],

  function(err)
  {
    if(err)
    {
      // если виртуальная машина существует, проверить,
      // не находится ли она в состоянии ожидания
      if(vm)
      {
        updateVMState(vm, eMachineState.recentLaunchUnknown);
      }
      res.send({msg: "Error starting server process: " + err});
    }
    else
    {
      res.send({msg: 'OK'});
    }
  });
});
```

ПРИМЕЧАНИЕ Этот обработчик конечной точки использует функцию `async.series`, утилиту из популярной библиотеки **async** на JavaScript. Она принимает массив функций и заключительную функцию завершения. Все функции в массиве вызываются поочередно и ожидают, пока выполнятся соответствующие им функции `callback`. После выполнения последовательности функций `async.series` вызывает функцию завершения. Если какая-либо из функций в массиве передаст своей функции `callback` признак ошибки, `series` немедленно прервет выполнение последовательности функций в массиве и вызовет функцию завершения с признаком ошибки. Библиотека `async` содержит множество других полезных высокоуровневых асинхронных конструкций и является одной из самых зависимых от других пакетов Node.

Кроме того, обработчик посылает REST-запросы диспетчеру LSPM, используя библиотеку **request**. Библиотека `request` — многофункциональная библиотека для создания HTTP-клиентов, своими возможностями напоминающая утилиту командной строки `curl`. Подобно `async`, она также является одной из самых востребованных библиотек в сообществе Node и достойна глубокого изучения. Ссылки на дополнительную информацию о библиотеках `async` и `request` можно найти в разделе «Для дополнительного чтения».

Передача параметров игры конечной точке `/processes/` диспетчера VMM вызывает запуск игрового процесса с этими параметрами. Обработчик делится на два основных раздела: получение виртуальной машины и запуск процесса. Сначала обработчик проверяет карту `gAvailableVMs` с целью найти виртуальную машину, доступную для запуска процесса. Если таковой не обнаруживается, создается уникальный идентификатор для новой виртуальной машины и поставщику услуг посылается запрос на ее создание. Функция `askCloudProviderForVM` зависит от специфических особенностей программного интерфейса поставщика услуг облачного хостинга и потому здесь не приводится. Она должна обратиться к API поставщика для создания и запуска виртуальной машины, указав образ с игрой и диспетчером LSPM, а затем запустить LSPM и передать ему идентификатор виртуальной машины.

Получив новую или уже работающую виртуальную машину, обработчик устанавливает в ней признак состояния `pending` (ожидание). Это гарантирует, что диспетчер VMM не попытается запустить на ней еще один процесс, пока не был запущен этот. Однопоточная природа Node препятствует традиционному состоянию гонки, но из-за того, что обработчик конечной точки использует асинхронные обратные вызовы, существует вероятность появления запроса на запуск другого процесса до того, как запустится текущий. В этом случае запрос должен быть направлен другой виртуальной машине, чтобы избежать наложения значений состояния друг на друга. При изменении состояния виртуальной машины на `pending` она удаляется из карты `gAvailableVMs`.

Переведя виртуальную машину в состояние `pending`, обработчик посылает REST-запрос диспетчеру LSPM этой виртуальной машины с требованием запустить игровой процесс. Если запуск прошел успешно, обработчик переводит виртуальную машину в новое состояние, которое возвращает диспетчер LSPM, — это должно быть состояние `partial` (частично заполнена) или `full` (заполнена), в зависимости от числа игровых процессов, выполняющихся в данный момент на этой виртуальной машине. Если от диспетчера LSPM придет отрицательный ответ или не придет никакого, диспетчер VMM не сможет определить состояние виртуальной машины. Может так случиться, что процесс не успел запуститься до того, как диспетчер вернул признак ошибки, или успел, но ответ был потерян где-то в сети. Даже при том, что TCP является надежным протоколом, клиенты и серверы HTTP

ограничивают время ожидания. Ненадежные сетевые кабели, всплески трафика или плохой сигнал Wi-Fi — все это вызывает превышение предельного времени ожидания. В случае неопределенной ошибки обработчик переводит виртуальную машину в состояние `recentLaunchUnknown` (результат последней попытки запуска неизвестен). В результате сервер выводится из состояния `pending`, чтобы система мониторинга работоспособности, о которой рассказывается ниже, смогла вернуть виртуальную машину в известное состояние или остановить ее. Кроме того, виртуальная машина исключается из карты `gAvailableVMs`, поскольку ее доступность неизвестна.

Если все сложилось благополучно, обработчик ответит на оригинальный запрос сообщением «OK», означающим, что игровой процесс на удаленной виртуальной машине запустился.

Мониторинг виртуальных машин

Диспетчер LSPM может зависнуть или завершиться аварийно в любой момент, поэтому диспетчер VMM должен осуществлять мониторинг работоспособности всех подчиненных ему диспетчеров LSPM. Чтобы гарантировать точность информации о состоянии LSPM в диспетчере VMM, диспетчер LSPM посылает изменения о своем состоянии с каждым сигналом работоспособности и маркирует их последовательно увеличивающимися значениями `sequenceIndex`, чтобы помочь диспетчеру VMM игнорировать сигналы, доставленные с задержкой. Когда поступает сигнал, сообщающий, что LSPM не имеет запущенных процессов, диспетчер VMM инициирует процедуру остановки, согласуя ее с диспетчером LSPM. Такое согласование предотвращает возникновение состояния гонки, при котором может произойти запуск процесса, пока VMM пытается остановить виртуальную машину. Из-за необходимости согласования и включения информации о состоянии в сигналы о работоспособности система получается немного сложнее, чем та, что использует LSPM для мониторинга процессов. В листинге 13.6 приводится реализация системы мониторинга в VMM.

Листинг 13.6. Система мониторинга работоспособности в VMM

```
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); // послать код состояния, чтобы lspm мог закрыть
соединение
    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);
            // lspm все еще готов к завершению?
            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);
      // попытка будет повторена после пропуска сигнала
    }
    else
    {
      delete gVMs[vm.uuid]; // успех... удалить отовсюду
      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);
```

Обработчик конечной точки `heartbeat` игнорирует сигналы от виртуальных машин в состоянии `pending` или `shuttingDown` (остановка). Виртуальные машины в состоянии ожидания изменяют свое состояние, как только будет получен ответ на запрос запуска, поэтому переход в любое другое состояние в это время должен обрабатываться после завершения запуска. Виртуальные машины в состоянии `shuttingDown` находятся в процессе остановки, поэтому для них не требуется осуществлять мониторинг изменений состояния. Обработчик также игнорирует сигналы с индексами, следующими не по порядку. Если сигнал о работоспособности прошел предварительные проверки, обработчик изменяет свойства `lastSequenceIndex` и `lastHeartbeat` виртуальной машины. Затем, если состояние имеет значение `empty`, указывающее на отсутствие игровых процессов в виртуальной машине, обработчик начинает процедуру остановки виртуальной машины, посылая соответствующий запрос диспетчеру LSPM. Обработчик запроса на остановку в диспетчере LSPM проверяет собственное значение `gMachineState`, чтобы убедиться, что оно не изменилось с момента отправки сигнала с состоянием `empty`. Если оно не изменилось, диспетчер LSPM изменяет собственное состояние на `shuttingDown` и посылает диспетчеру VMM ответ, что запрос на остановку принят. Затем диспетчер VMM отмечает данную виртуальную машину значением состояния `shuttingDown` и посылает запрос поставщику услуг о необходимости ликвидации виртуальной машины.

Функция `checkHeartbeats` диспетчера VMM действует подобно одноименной функции в LSPM, но игнорирует любые тайм-ауты для виртуальных машин в состоянии `pending`. Если предельное время ожидания сигнала о работоспособности виртуальной машины истекло, это означает, что с диспетчером LSPM произошло что-то непредвиденное, поэтому диспетчер VMM, не заботясь о согласовании процедуры остановки, сразу же посылает поставщику услуг запрос на остановку и удаление виртуальной машины.

Когда диспетчер LSPM обнаруживает изменение состояния из-за завершившегося процесса, ему не нужно ждать, пока пройдет предопределенный интервал посылки сигналов о работоспособности, чтобы известить диспетчер VMM. Вместо этого в ответ на изменение состояния он может просто послать дополнительный сигнал немедленно. Это простейший способ организовать быструю обратную связь с диспетчером VMM, он не требует дополнительного кода на стороне VMM.

Эта реализация диспетчера VMM функционально безупречна: она препятствует появлению состояния гонки и достаточно эффективна. Однако если одновременно поступит несколько запросов, она обслужит их все, но для каждого создаст свою виртуальную машину. При равномерном трафике это не превратится в проблему,

но если трафик носит пиковый характер, может быть создано излишне большое число виртуальных машин. Более удачная реализация могла бы отслеживать эту ситуацию и придерживать запросы на создание виртуальных машин. Точно так же диспетчер VMM, возможно, неэффективен в агрессивном стремлении немедленно останавливать опустевшие виртуальные машины. В зависимости от частоты запросов на вступление в игру и выход из нее мы рекомендуем сохранять пустые виртуальные машины в течение некоторого времени, прежде чем удалять их. В более надежной реализации VMM можно предусмотреть настраиваемое пороговое значение для этого интервала. Реализацию описанных улучшений мы оставляем читателям в качестве самостоятельного упражнения.

СОВЕТ Если диспетчер VMM должен обрабатывать несколько тысяч запросов в секунду, возможно, вам следует настроить балансировщик нагрузки и запустить за ним несколько экземпляров Node для обслуживания запросов. В этом случае состояния виртуальных машин в массиве gVMs должны будут совместно использоваться всеми экземплярами, поэтому данный массив следует разместить не в локальной памяти процесса, а в быстродействующем хранилище данных, таком как *redis*. Ссылки на дополнительную информацию об этом хранилище можно найти в разделе «Для дополнительного чтения». С другой стороны, если запросы поступают достаточно часто, возможно, будет лучшим решением распределить игроков по географическому признаку, с диспетчерами VMM, статически привязанными к определенным регионам.

В заключение

С увеличением числа поставщиков услуг облачного хостинга каждая студия, занимающаяся производством многопользовательских игр, должна рассмотреть возможность размещения выделенных серверов в облаке. Но несмотря на то что в настоящее время это стало намного проще, чем когда-либо, услуги хостинга выделенных серверов все еще стоят дороже, чем размещение серверов у игроков, и сложность подобной реализации также выше. Кроме того, возникает зависимость от сторонних поставщиков услуг облачного хостинга, и у игроков исчезает ощущение причастности. Однако преимущества хостинга выделенных серверов часто перевешивают перечисленные недостатки. Размещение серверов в облаке дает дополнительную надежность, увеличивает их доступность и пропускную способность, помогает предотвращать мошенничество и обеспечивает ненавязчивую защиту от копирования.

Для хостинга выделенных серверов требуется создать несколько серверных утилит. Инструменты разработки серверного кода существенно отличаются от тех, что применяются в разработке клиентского игрового кода. Программные интерфейсы REST осуществляют обмен между службами текстовыми данными в читаемом виде, пригодном для анализа и отладки. JSON — ясный и компактный формат, предназначенный для обмена данными. Node JS — оптимизированный, управляемый событиями движок на JavaScript для быстрой разработки.

Инфраструктура выделенных серверов состоит из нескольких компонентов. Серверный экземпляр игры совместно используется несколькими игроками. Серверный игровой процесс, который может включать один или несколько серверных

экземпляров игры, представляет игру в ОС. На одном игровом сервере-машине может выполняться один или несколько серверных игровых процессов. Обычно игровые серверы-машины в действительности являются виртуальными машинами, работающими в одиночку или одновременно с другими виртуальными машинами на одной и той же физической машине.

Для управления всеми этими компонентами применяются локальный диспетчер серверных процессов (LSPM) и диспетчер виртуальных машин (VMM). В каждой виртуальной машине действует свой диспетчер LSPM, и он отвечает за запуск и мониторинг процессов на этой машине, а также за своевременную отправку информации о своем состоянии диспетчеру VMM. Диспетчер VMM, в свою очередь, является главной точкой входа для запросов на запуск новых процессов. Когда служба координации решает, что нужно запустить новый игровой сервер, она посылает REST-запрос конечной точке диспетчера VMM. Затем обработчик этой конечной точки либо находит недоиспользованную виртуальную машину, либо запрашивает у поставщика услуг облачного хостинга запуск новой виртуальной машины. Получив подходящую виртуальную машину, он посылает диспетчеру LSPM в этой виртуальной машине запрос на запуск нового серверного игрового процесса.

Все эти компоненты тесно взаимодействуют друг с другом с целью обеспечить надежное окружение выделенного сервера, масштабируемое в зависимости от числа игроков, без предварительных финансовых вложений.

Вопросы для повторения

1. Какие преимущества и недостатки имеет хостинг выделенных серверов? Почему хостинг выделенных серверов в прошлом был намного более сложной задачей?
2. Перечислите «за» и «против» поддержки нескольких экземпляров игры в одном серверном игровом процессе.
3. Что такое «виртуальная машина»? Почему в облачном хостинге обычно предоставляются виртуальные машины?
4. Какие основные функции выполняет локальный диспетчер серверных процессов?
5. Перечислите несколько способов организации обратной связи между серверным игровым процессом и локальным диспетчером серверных процессов.
6. Что такое «диспетчер виртуальных машин» и какие задачи он решает?
7. Объясните, почему иногда диспетчер VMM может запустить больше виртуальных машин, чем требуется в действительности. Реализуйте код, устраняющий эту проблему.
8. Объясните, почему стремление диспетчера VMM немедленно остановить ненужные виртуальные машины может ухудшить эффективность. Реализуйте код, устраняющий эту проблему.

Для дополнительного чтения

Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. (1999, июнь). «Hypertext Transfer Protocol — HTTP/1.1». Доступно по адресу: *http://www.rfc-base.org/rfc-2616.html*.[1] Проверено 28 января 2016.

Домашняя страница библиотеки C++ REST SDK. Доступна по адресу: *https://casablanca.codeplex.com*. Проверено 28 января 2016.

Домашняя страница библиотеки caolan/async. Доступна по адресу: *https://github.com/caolan/async*. Проверено 28 января 2016.

Домашняя страница проекта Docker. Доступна по адресу: *https://www.docker.com*. Проверено 28 января 2016.

Домашняя страница фреймворка Express для создания веб-приложений в Node.js. Доступна по адресу: *http://expressjs.com*. Проверено 28 января 2016.

Введение в JSON. Доступно по адресу: *http://json.org*. Проверено 28 января 2016.

Домашняя страница проекта Node.js. Доступна по адресу: *https://nodejs.org*. Проверено 28 января 2016.

Документация к проекту Redis. Доступно по адресу: *http://redis.io/documentation*. Проверено 28 января 2016.

Библиотека request/request. Доступно по адресу: *https://github.com/request/request*. Проверено 28 января 2016.

Вики-страница с описанием Rest. Доступна по адресу: *https://ru.wikipedia.org/wiki/REST*. Проверено 28 января 2016.

[1] Получить перевод на русский язык можно по адресу: *http://www.php.su/phphttp/docs/rfc2616/. — Примеч. пер.*

Приложение. Современный C++

Язык C++ является индустриальным стандартом в программировании видеоигр. Несмотря на то что для реализации игровой логики многие компании используют языки более высокого уровня, низкоуровневый код, например, реализующий логику сетевых взаимодействий, практически всегда пишется на C++. В примерах программного кода, приводимых в этой книге, используются относительно новые для языка C++ концепции, которые как раз рассматриваются в этом приложении.

C++11

Утвержденный в 2011 году стандарт C++11 привнес множество изменений в язык C++. Из наиболее важных новшеств, добавленных в C++11, можно назвать фундаментальные языковые конструкции (такие, как лямбда-выражения) и новые библиотеки (например, библиотека поддержки многопоточного выполнения). Несмотря на большое число нововведений в C++11, в этой книге используется только их малое подмножество. Для ознакомления со всеми новшествами языка вам лучше обратиться к дополнительной литературе. В первом разделе мы рассмотрим некоторые общие концепции C++11, которые не соответствуют тематике остальных разделов настоящего приложения.

Стандарт C++11 все еще считается относительно новым, и не все компиляторы полностью совместимы с ним. Однако все концепции C++11, использованные в этой книге, поддерживаются тремя наиболее популярными компиляторами: Microsoft Visual Studio, Clang и GCC.

Нельзя не упомянуть еще более новую версию стандарта C++ — C++14. Однако C++14 является скорее корректирующей версией, поэтому в ней не так много изменений, как в C++11. Следующую версию стандарта планируется выпустить в 2017 году.

auto

Ключевое слово `auto` существовало в предыдущих версиях C++, но в C++11 оно получило новое значение. В частности, использование этого ключевого слова вме-

сто типа сообщает компилятору, что тот должен «угадать» тип переменной на этапе компиляции. А поскольку тип определяется еще на этапе компиляции, это означает, что использование типа `auto` не влечет никаких накладных расходов времени исполнения, но позволяет писать более краткий код.

Например, в прежних стандартах C++ объявление итераторов было настоящей головной болью (если вы не знакомы с понятием итераторов, то сможете прочитать о них далее в этом приложении):

```
// Объявление вектора целых чисел
std::vector<int> myVect;
// Объявление итератора, ссылающегося на начало вектора
std::vector<int>::iterator iter = myVect.begin();
```

Однако в C++11 сложное объявление типа итератора можно заменить ключевым словом `auto`:

```
// Объявление вектора целых чисел
std::vector<int> myVect;
// Объявление итератора (с помощью auto), ссылающегося на начало вектора
auto iter = myVect.begin();
```

Так как тип значения, возвращаемого `myVect.begin()`, известен на этапе компиляции, компилятор может определить тип `iter`. Ключевое слово `auto` можно также использовать для определения простых типов, таких как целые или вещественные числа, но ценность такого приема довольно сомнительна. Имейте также в виду, что по умолчанию `auto` не подразумевает ни ссылочного типа, ни спецификатора `const`, если эти свойства желательны, используйте явные объявления: `auto&`, `const auto` или `const auto&`.

nullptr

В прежних версиях стандарта, предшествовавших C++11, для очистки указателя необходимо было присвоить ему число `0` или макрос `NULL` (который определяется директивой `#define` как простой псевдоним для числа `0`). Однако основной недостаток такого подхода заключался в том, что `0` интерпретируется в первую очередь как целое число. Это могло вызвать проблемы в случае перегрузки функций. Предположим, например, что определены следующие две функции:

```
void myFunc(int* ptr)
{
  // Выполнить некоторые действия
  //...
}
void myFunc(int a)
{
  // Выполнить некоторые действия
  //...
}
```

Проблема возникнет, если вызвать `myFunc` с аргументом `NULL`. Можно было бы ожидать, что в результате будет вызвана первая версия функции, но это не так. Причина в том, что `NULL` — это `0`, а `0` интерпретируется как целое число. С другой стороны, если передать в функцию аргумент `nullptr`, то будет вызвана первая функция, потому что `nullptr` интерпретируется как указатель.

Хотя это всего лишь пример, он показывает, что ключевое слово `nullptr` строго интерпретируется как указатель, тогда как `NULL` или `0` — нет. Использование ключевого слова имеет еще одно преимущество: оно легко идентифицируется при просмотре исходных текстов — его трудно спутать с чем-либо другим — в отличие от числа `0`, которое может встречаться во множестве мест, где вообще нет указателей.

Ссылки

Ссылка (reference) — это тип переменной, которая ссылается на другую переменную. Следовательно, при изменении переменной-ссылки должна изменяться оригинальная переменная. Чаще всего ссылки используются при определении функций, которые изменяют свои параметры. Например, следующая функция меняет значения параметров `a` и `b` местами:

```
void swap(int& a, int& b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

То есть если функции `swap` передать две целочисленные переменные, по завершении ее работы эти две переменные поменяются значениями. Это происходит благодаря тому, что параметры `a` и `b` представляют собой ссылки на оригинальные переменные. Если бы функцию `swap` потребовалось написать на C, пришлось бы вместо ссылок использовать указатели. На уровне компиляции ссылки фактически реализованы как указатели, но семантика использования ссылок намного проще, потому что их разыменование происходит неявно. Кроме того, ссылки безопаснее использовать в параметрах функций, потому что они гарантированно не могут быть пустыми (хотя теоретически ничто не помешает написать уродливый код, в котором ссылка будет пустой).

Константные ссылки

Возможность изменения параметров — лишь одна из множества причин использования ссылок. Для сложных типов (таких, как классы и структуры) передача по ссылке почти всегда оказывается эффективнее передачи по значению. Это обусловлено необходимостью создания копии переменной при передаче по значению, а для сложных типов, таких как вектор или строка, создание копии требует динамического выделения памяти, что значительно увеличивает накладные расходы.

С другой стороны, если вектор или строка просто передаются в функцию по ссылке, это означает, что функция сможет изменить значение оригинальной переменной. А как быть, когда такая возможность должна быть запрещена, например когда переменная представляет данные класса? Решение этой проблемы заключается в использовании *константных ссылок* (const reference). Константная ссылка подобна обычной, но ее можно использовать только для чтения — любые изменения запрещены. В результате получается соединение лучшего с хорошим: отпадает необходимость создания копии, и функция не сможет изменить данные. Следующая функция `print` является одним из примеров передачи параметров по константной ссылке:

```
void print(const std::string& toPrint)
{
  std::cout << toPrint << std::endl;
}
```

Считается хорошим тоном передавать значения сложных типов в функции по константной ссылке, если только функция специально не предназначена для изменения оригинальной переменной, — в этом случае следует использовать обычную ссылку. Однако применение ссылок для передачи простых типов (целые или вещественные числа) вместо их копий, наоборот, ухудшает эффективность. Простые типы предпочтительнее передавать по значению, если только функция специально не предназначена для изменения оригинальной переменной, — в этом случае следует использовать обычную ссылку.

Константные функции-члены

К функциям-членам и их параметрам применяются те же правила, что и к обычным функциям. То есть сложные типы обычно предпочтительнее передавать по константной ссылке, а простые — по значению. Ситуация осложняется, если речь заходит о значениях, возвращаемых так называемыми *функциями чтения* (getter functions) — функциями, возвращающими значения полей объектов. Обычно такие функции должны возвращать константные ссылки на данные-члены — это не позволяет вызывающему коду нарушить правила инкапсуляции и изменить данные.

Используя константные ссылки с классами, помните, что любые функции-члены, не изменяющие данные в членах, определяются как константные. *Константная функция-член* (const member function) гарантирует, что не изменит внутренние данные класса (и строго соблюдает это обязательство). Это важно, потому что к константной ссылке на объект можно применить только константную функцию-член этого объекта. Если попытаться вызвать неконстантную функцию для константной ссылки, это приведет к ошибке компиляции.

Чтобы объявить функцию-член константной, в ее определение следует добавить ключевое слово `const` после круглой скобки, закрывающей список параметров. Следующий класс `Student` демонстрирует правильное использование ссылок

и константных функций-членов. Такое использование `const` часто называют *const-корректностью* (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;}
};
```

Шаблоны

Шаблон (template) — это способ объявить функцию или класс так, чтобы они могли применяться к любым типам. Например, следующая шаблонная функция `max` будет применима к любым типам, поддерживающим оператор «больше чем»:

```
template <typename T>
T max(const T& a, const T& b)
{
  return ((a > b) ? a : b);
}
```

Когда компилятор встречает вызов `max`, он создает версию шаблона для данного типа. Соответственно, если в программе имеются два вызова `max` — один для целых чисел и один для вещественных, — компилятор создает две версии `max`. То есть размер исполняемого модуля и скорость работы будут идентичны версии с двумя функциями `max`, объявленными вручную.

Аналогичный подход применяется к классам и/или структурам, и он широко используется в библиотеке STL (описывается далее в этом приложении). Однако, так же как в случае со ссылками, существуют и дополнительные способы использования шаблонов.

Специализация шаблона

Допустим, что у нас есть шаблонная функция `copyToBuffer`, принимающая два параметра: указатель на буфер для записи и переменную (шаблонную), значение

которой нужно записать. Ниже приводится один из вариантов реализации такой функции:

```
template <typename T>
void copyToBuffer(char* buffer, const T& value)
{
  std::memcpy(buffer, &value, sizeof(T));
}
```

Однако эта функция обладает серьезным недостатком. Она прекрасно справляется с простыми типами, но сложные типы, такие как строки, обрабатываются неправильно. Дело в том, что функция выполняет поверхностное копирование — в отличие от глубокого, затрагивающего все внутренние данные. Чтобы исправить ситуацию, создайте специализированную версию `copyToBuffer`, выполняющую глубокое копирование строк:

```
template <>
void copyToBuffer<std::string>(char* buffer, const std::string& value)
{
  std::memcpy(buffer, value.c_str(), value.length());
}
```

Теперь, если в программе вызвать `copyToBuffer` со строкой в качестве второго аргумента, компилятор подставит вызов специализированной версии. Такая специализация применима и к шаблонам, принимающим множество шаблонных параметров, — в этом случае можно специализировать любое число таких параметров.

Статические утверждения и свойства типов

Утверждения (assertions) времени выполнения очень полезны для проверки значений. В играх вместо исключений зачастую предпочтительнее использовать утверждения, потому что они влекут меньшие накладные расходы и легко могут быть удалены перед выпуском оптимизированной версии.

Статические утверждения (static assertion) — это разновидность утверждений, которые выполняют проверки на этапе компиляции. А так как проверки выполняются на этапе компиляции, логические значения выражений, реализующих проверку, также должны быть известны на этапе компиляции. Ниже приводится пример простой функции, которая не компилируется из-за наличия статического утверждения:

```
void test()
{
  static_assert(false, "Doesn't compile!");
}
```

Конечно, статическое утверждение с условием `false` может вызывать только прерывание компиляции. На практике статические утверждения часто используются с заголовочным файлом `type_traits`, определяемым стандартом C++11, это позволяет запретить использование шаблонных функций с определенными типами.

Вернемся к примеру функции `copyToBuffer`. Для нас было бы удобнее, если бы обобщенная версия функции работала только с простыми типами. Этого можно добиться с помощью статического утверждения, например:

```
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));
}
```

Свойство `is_fundamental` будет иметь истинное значение, только когда `T` является простым типом. То есть любой вызов обобщенной версии `copyToBuffer` не будет компилироваться, если `T` не относится к простым типам. Самое интересное начинается, когда в игру вступает специализация: если для данного типа предусмотрен специализированный шаблон, обобщенная версия будет проигнорирована и, соответственно, статическое утверждение будет пропущено. То есть если в программе существует строковая версия `copyToBuffer`, как в примере выше, вызов функции со строкой во втором параметре будет компилироваться без ошибок.

«Умные» указатели

Указатель (pointer) — это тип переменной, хранящей адрес, и одна из основных конструкций, используемых программистами на C/C++. Однако при неправильном использовании указателей существует вероятность появления труднообнаружимых ошибок. Одна из них — утечка памяти, когда память динамически распределяется из кучи, но никогда не освобождается. Например, следующий класс страдает утечкой памяти:

```
class Texture
{
private:
  struct ImageData
  {
    //...
  };
  ImageData* mData;
public:
  Texture(const char* filename)
  {
  mData = new ImageData;
    // Загрузить ImageData из файла
    //...
  }
};
```

Обратите внимание: здесь конструктор класса динамически выделяет память для структуры, но она не освобождается в деструкторе. Чтобы устранить эту утечку,

добавьте деструктор, который освободит память, на которую ссылается `mData`. Исправленная версия `Texture` приводится ниже:

```
class Texture
{
private:
  struct ImageData
  {
    //...
  };
  ImageData* mData;
public:
  Texture(const char* fileName)
  {
    mData = new ImageData;
    // Загрузить ImageData из файла
    //...
  }

  ~Texture()
  {
    delete mData; // Устраняет утечку памяти
  }
};
```

Вторая, более коварная ошибка может возникнуть, если несколько объектов содержат указатели на одну и ту же переменную, размещенную в динамической памяти. Например, взгляните на следующий класс `Button` (он использует класс `Texture`, объявленный выше):

```
class Button
{
private:
  Texture* mTexture;
public:
  Button(Texture* texture)
    : mTexture(texture)
  {}

  ~Button()
  {
    delete mTexture;
  }
};
```

Задача состоит в том, чтобы отобразить текстуру на поверхности кнопки, при этом текстура должна заранее динамически размещаться в памяти. А теперь представьте, что получится, если создать два экземпляра кнопки `Button`, каждый из которых будет указывать на один и тот же экземпляр `Texture`? Пока обе кнопки активны, ничего не случится. Но как только первый экземпляр `Button` будет удален, указатель на экземпляр `Texture` станет недействительным. Но второй экземпляр `Button` все еще будет иметь указатель на память, где прежде находился только что удаленный экземпляр `Texture`, что в лучшем случае вызовет искажения в изображении кнопки,

а в худшем — аварийное завершение программы. Эта проблема не имеет простого решения при использовании обычных указателей.

«Умные» указатели решают обе описанные проблемы, и с появлением стандарта C++11 они стали частью стандартной библиотеки (в заголовочном файле `memory`).

Разделяемые указатели

Разделяемый указатель (shared pointer) — это один из «умных» указателей, позволяющих иметь несколько указателей на одну и ту же переменную, размещенную в динамической памяти. «За кулисами» кода разделяемый указатель подсчитывает количество указателей на целевую переменную, — этот процесс называют *подсчетом ссылок* (reference counting). Память, занятая целевой переменной, освобождается, только когда счетчик ссылок достигнет нуля. Благодаря этому разделяемый указатель может гарантировать, что переменная, на которую еще есть ссылки, не будет удалена раньше времени.

Создавать разделяемые указатели предпочтительнее с помощью шаблонной функции `make_shared`. Ниже приводится простой пример использования разделяемых указателей:

```
{
  // Сконструировать разделяемый указатель на значение типа int
  // Инициализировать целевую переменную числом 50
  // Счетчик ссылок равен 1
  std::shared_ptr<int> p1 = std::make_shared<int>(50);
  {
    // Создать новый разделяемый указатель
    // на ту же целевую переменную.
    // Счетчик ссылок теперь равен 2
    std::shared_ptr<int> p2 = p1;

    // Разыменовать shared_ptr как обычный указатель
    *p2 = 100;
    std::cout << *p2 << std::endl;
  } // p2 уничтожается, счетчик ссылок становится равным 1
}   // p1 уничтожается, счетчик ссылок становится равным 0,
    // поэтому целевая переменная удаляется
```

Как мы видим, обе функции, `shared_ptr` и `make_shared`, являются шаблонными по типу динамически размещаемой переменной. Функция `make_shared` автоматически выделяет необходимую память, — обратите внимание на отсутствие прямых вызовов `new` или `delete` в этом примере. Имеется возможность напрямую передать адрес в памяти конструктору `shared_ptr`, но такой прием не рекомендуется использовать, если это не является абсолютно необходимым, так как он, в отличие от `make_shared`, менее эффективен и чреват ошибками.

Если понадобится передать разделяемый указатель в функцию в виде аргумента, передавайте его по значению как переменную простого типа. Это противоречит обычным правилам передачи по ссылке, но иначе не получится гарантировать корректное состояние счетчика ссылок в разделяемом указателе.

Опираясь на новые знания, можно переписать класс `Button` из предыдущего раздела и задействовать в нем указатель `shared_ptr` на экземпляр `Texture`, как показано ниже. В данном случае экземпляр текстуры `Texture` гарантированно будет храниться в памяти, пока существует хотя бы один активный разделяемый указатель на него.

```
class Button
{
private:
  std::shared_ptr<Texture> mTexture;
public:
  Button(std::shared_ptr<Texture> texture)
    : mTexture(texture)
  {}

  // Деструктор не нужен, потому что используется умный указатель!
};
```

Разделяемые указатели имеют еще одну особенность, о которой следует упомянуть. Если классу требуется получить разделяемый указатель на самого себя, программист не должен вручную конструировать новый разделяемый указатель на основе указателя `this`, потому что в этом случае не будут учтены существующие счетчики ссылок. Вместо этого следует воспользоваться шаблонным классом `enable_shared_from_this`. Например, если экземплярам `Texture` потребуется получать разделяемые указатели на самих себя, можно унаследовать этот класс от класса `enable_shared_from_this`, как показано ниже:

```
class Texture: public std::enable_shared_from_this<Texture>
{
  // Реализация
  //...
};
```

После этого внутри любой функции-члена класса `Texture` можно вызывать функцию-член `shared_from_this`, возвращающую разделяемый указатель с корректно установленным счетчиком ссылок.

Существуют также шаблонные функции для приведения разделяемых указателей к разным классам в иерархии: `static_pointer_cast` и `dynamic_pointer_cast`.

Уникальные указатели

Уникальный указатель (unique pointer) напоминает разделяемый указатель, но гарантирует, что только один указатель сможет ссылаться на целевую переменную. Если попытаться присвоить один уникальный указатель другому, это приведет к ошибке. То есть уникальным указателям не требуется подсчитывать ссылки — они автоматически удаляют целевую переменную в момент уничтожения уникального указателя.

Для работы с уникальными указателями используйте `unique_ptr` и `make_unique` — помимо подсчета ссылок, код, использующий `unique_ptr`, очень похож на код, использующий `shared_ptr`.

Слабые указатели

Указатели `shared_ptr` фактически поддерживают два счетчика ссылок: счетчик *сильных* ссылок и счетчик *слабых* ссылок. Когда счетчик сильных ссылок достигает нуля, целевой объект уничтожается. Однако счетчик слабых ссылок никак не учитывается при принятии решения об удалении или неудалении целевого объекта. Благодаря этому можно получить слабый указатель со слабой ссылкой на объект, управляемый разделяемым указателем. Основная цель использования слабого указателя — дать программному коду, который в действительности не должен владеть объектом, возможность безопасно проверить существование определенного объекта. Для этой цели в C++11 предусмотрен класс `weak_ptr`.

Допустим, что указатель `sp` уже объявлен как `shared_ptr<int>`. В этом случае можно создать `weak_ptr` из `shared_ptr`:

```
std::weak_ptr<int> wp = sp;
```

Проверить допустимость слабого указателя можно с помощью функции `expired`. Если он действительный, то с помощью вызова `lock` приобретается `shared_ptr` и тем самым увеличивается счетчик сильных ссылок. Например:

```
if (!wp.expired())
{
  // следующий вызов увеличит счетчик сильных ссылок
  std::shared_ptr<int> sp2 = wp.lock();
  // Теперь можно использовать sp2 как shared_ptr
  //...
}
```

Слабые указатели также используются, чтобы избежать циклических ссылок. Например, если объект А имеет указатель `shared_ptr` на объект Б, а объект Б имеет указатель `shared_ptr` на объект А, то программа не сможет удалить ни один из этих объектов. Однако если один из них будет хранить слабый указатель `weak_ptr`, циклическая ссылка не будет образована.

Предостережения

Реализация «умных» указателей в C++11 имеет пару особенностей, о которых следует помнить. Прежде всего, очень трудно обеспечить корректное их использование с динамическими массивами. Если «умный» указатель нужен для работы с массивом, проще использовать контейнер массива из библиотеки STL. Нужно также отметить, что, в отличие от обычных указателей, «умные» указатели потребляют больше памяти и влекут за собой накладные расходы процессорного времени. Поэтому в программах, где требуется максимально высокая скорость

выполнения, лучше отказаться от «умных» указателей. Но в большинстве случаев «умные» указатели проще и безопаснее (и потому предпочтительнее).

Контейнеры STL

Стандартная библиотека шаблонов C++ (Standard Template Library, STL) содержит огромное число структур-контейнеров. В этом разделе рассказывается о некоторых наиболее широко используемых контейнерах и типовых случаях их использования. Объявление каждого контейнера находится в заголовочном файле с соответствующим именем, поэтому часто приходится подключать несколько заголовочных файлов, чтобы воспользоваться несколькими контейнерами.

array

Контейнер `array` (был добавлен в C++11) по сути является оберткой вокруг массива фиксированного размера. Так как его размер не изменяется, в нем нет функции-члена `push_back` или похожей на нее. Элементы массива доступны при помощи стандартного оператора индексирования `[]`. Главным преимуществом массивов является возможность произвольного доступа к их элементам с алгоритмической сложностью $O(1)$.

Массивы в стиле языка C служат той же цели, но одним из преимуществ контейнера `array` является поддержка семантики итераторов. Кроме того, если использовать функцию-член вместо оператора индекса, появляется возможность проверять выход за границы массива.

vector

Контейнер `vector` — это массив, размер которого может изменяться динамически. Элементы добавляются в конец вектора и удаляются из конца вектора с помощью `push_back` и `pop_back` с алгоритмической сложностью $O(1)$. Для вставки/удаления элементов в произвольные позиции можно использовать `insert` и `remove`. Однако эти операции требуют копирования некоторых или всех данных в массиве, что делает их весьма дорогостоящими. По тем же причинам дорогостоящей считается и операция изменения размеров вектора, несмотря на алгоритмическую сложность $O(n)$. Это также означает, что даже операция `push_back` с ее алгоритмической сложностью $O(1)$ требует времени на копирование, если применяется к заполненному вектору. Так же как в случае с контейнером `array`, при использовании функции-члена осуществляется проверка границ массива.

Если заранее известно, сколько элементов потребуется поместить в вектор, воспользуйтесь функцией-членом `reserve`, чтобы выделить память для необходимого количества элементов. Это поможет избежать затрат на увеличение и копирование вектора по мере добавления элементов и сэкономить массу времени.

В стандарте C++11 появилась новая функция-член для добавления элементов в вектор: `emplace_back`. Ее отличие от `push_back` особенно заметно, если имеется

вектор с элементами сложного типа. Допустим, что существует вектор с элементами пользовательского класса `Student`. Предположим, что конструктор класса `Student` принимает имя студента и номер курса. При помощи функции `push_back` добавить новый элемент в вектор можно было бы следующим образом:

```
students.push_back(Student("John", 100));
```

Эта инструкция сначала создаст временный экземпляр класса `Student`, а затем — копию этого временного экземпляра, чтобы добавить ее в вектор. Функция `emplace_back`, в отличие от `push_back`, может сконструировать объект на месте, не создавая промежуточную временную копию. Ниже показано, как можно было бы использовать `emplace_back`:

```
students.emplace_back("John", 100);
```

Обратите внимание, что в вызове `emplace_back` отсутствует явное упоминание типа `Student`. Это называется *идеальной передачей* (perfect forwarding), потому что параметры передаются конструктору `Student`, который создает объект в векторе.

Нет ничего, что препятствовало бы использованию `emplace_back` вместо `push_back`. Все другие контейнеры STL (кроме массива) поддерживают создание экземпляров на месте, поэтому мы рекомендуем использовать эти функции для добавления элементов в контейнеры.

list

Контейнер `list` — это двунаправленный список. Элементы можно добавлять/удалять в начало и в конец списка с гарантированной алгоритмической сложностью $O(1)$. Кроме того, при наличии итератора, ссылающегося на произвольное место в списке, можно использовать функции `insert` и `remove`, имеющие сложность $O(1)$. Помните, что списки не поддерживают произвольный доступ к элементам. Одно из преимуществ связанного списка в том, что он никогда не «переполнится» — элементы добавляются по одному, поэтому не приходится беспокоиться об изменении размера списка. Однако следует отметить один из недостатков связанных списков: элементы располагаются в памяти не друг за другом, поэтому списки не так хорошо кэшируются, как массивы. Оказывается, производительность кэша фактически является чуть ли не самым узким местом в современных компьютерах. Поэтому с элементами относительно небольшого размера (до 64 байт) вектор практически всегда будет превосходить списки по производительности.

forward_list

Контейнер `forward_list` (добавлен в C++11) — это однонаправленный список. Списки `forward_list` поддерживают операции добавления и удаления, имеющие алгоритмическую сложность $O(1)$ только в начале списка. Основное преимущество списков этого типа — меньший объем памяти, занимаемый каждым элементом.

map

`map` — упорядоченный контейнер пар {ключ, значение}, в котором упорядочение выполняется по ключу. Каждый ключ в `map` должен быть уникальным и поддерживать *строгую квазиупорядоченность* (strict weak ordering), то есть если ключ А меньше ключа Б, то ключ Б не может быть меньше ключа А или равным ему. Если в роли ключа предполагается использовать собственный тип, обычно для этого требуется переопределить оператор «меньше чем». Контейнер `map` реализован как двоичное дерево поиска, то есть средняя алгоритмическая сложность поиска по ключу составляет $O(\log(n))$. Поскольку данные хранятся в контейнере в упорядоченном виде, гарантируется, что итерации будут выполняться в нем в порядке возрастания ключей.

set

Контейнер `set` напоминает `map`, но в отличие от последнего хранит не пары. Ключ в нем одновременно играет роль значения. Все остальные характеристики `set` и `map` идентичны.

unordered_map

Контейнер `unordered_map` (добавлен в C++11) — это хеш-таблица пар {ключ, значение}. Каждый ключ в контейнере должен быть уникальным. Так как это хеш-таблица, поиск в ней выполняется с алгоритмической сложностью $O(1)$. Однако она хранит данные в неупорядоченном виде, поэтому итерации через `unordered_map` будут выполняться в случайном порядке. В библиотеке существует также похожий контейнер `unordered_set`. Оба контейнера, `unordered_map` и `unordered_set`, имеют функции хеширования для встроенных типов. Если понадобится хешировать собственный тип, вам придется реализовать свою специализированную шаблонную функцию `std::hash`.

Итераторы

Итератор — это тип объектов, основное назначение которых заключается в поддержке итераций по содержимому контейнера. Итераторы поддерживают все контейнеры STL, и в этом разделе рассматриваются наиболее типичные случаи их использования.

Следующий фрагмент конструирует вектор. Он добавляет в вектор первые пять чисел Фибоначчи и затем использует итератор для вывода значений всех элементов вектора:

```
std::vector<int> myVec;
myVec.emplace_back(1);
myVec.emplace_back(1);
myVec.emplace_back(2);
```

```
myVec.emplace_back(3);
myVec.emplace_back(5);

// Выполнить итерации по содержимому вектора и вывести каждый элемент
for(auto iter = myVec.begin();
  iter != myVec.end();
  ++iter)
{
  std::cout << *iter << std::endl;
}
```

Чтобы получить итератор, ссылающийся на первый элемент в контейнере STL, используется функция-член `begin`. Подобная ей функция-член `end` возвращает итератор, ссылающийся на последний элемент. Обратите внимание, что для объявления типа итератора в коде используется ключевое слово `auto`. Это избавляет от необходимости вручную вводить полное имя типа (в данном случае `std::vector<int>::iterator`).

Отметим также, что переход к следующему элементу выполняется применением префиксного оператора `++` к итератору, — по соображениям производительности вместо постфиксного оператора следует использовать префиксный. Наконец, итераторы разыменовываются как обычные указатели — именно так осуществляется доступ к целевым элементам. Некоторые сложности могут возникнуть, если элемент сам является указателем, потому что в этом случае придется использовать две звездочки: одну для разыменования итератора и другую — для разыменования самого указателя.

Все контейнеры STL поддерживают два вида итераторов: обычные итераторы, представленные выше, и константные итераторы `const_iterator`. Константный итератор, в отличие от обычного, не позволяет изменять данные в контейнере. То есть если в коде имеется константная ссылка на контейнер STL, с ним можно будет использовать только константный итератор `const_iterator`.

Цикл for по коллекциям

В ситуации, когда требуется организовать цикл по всему контейнеру, самое простое решение — использовать новинку, появившуюся в C++11, — *цикл for по коллекциям* (range-based for loop). Цикл, который приводился чуть выше, можно переписать следующим образом:

```
// Выполнить итерации с использованием цикла for по коллекциям
for (auto i : myVec)
{
  std::cout << i << std::endl;
}
```

Цикл for по коллекциям очень похож на цикл `foreach` в других языках, таких как Java или C#. Этот код извлекает каждый элемент из контейнера и сохраняет его во временной переменной `i`. Цикл завершается сразу, как только будет достигнут последний элемент. Цикл for по коллекциям может извлекать элементы по значению

или по ссылке. То есть если потребуется изменить элементы в контейнере, воспользуйтесь ссылками, а для сложных типов всегда следует использовать ссылки или константные ссылки.

Цикл for по коллекциям можно применять к любым контейнерам, поддерживающим семантику итераторов STL (то есть имеющим члены `iterator`, `begin` и `end`, допускающие увеличение, разыменование и т. д.). Можно создать и свой контейнер, поддерживающий цикл for по коллекциям.

Другие случаи использования итераторов

В заголовочном файле `algorithm` определено множество разнообразных функций, использующих итераторы тем или иным способом. Однако еще одним распространенным случаем использования итераторов является функция-член `find`, которая поддерживается контейнерами `map`, `set` и `unordered_map`. Функция-член `find` выполняет поиск указанного ключа в контейнере и возвращает итератор, ссылающийся на соответствующий элемент. Если ключ не найден, `find` вернет итератор, равный итератору `end`.

Для дополнительного чтения

Meyers, Scott. (2014, декабрь). «Effective Modern C++». O'Reilly Media.[1]

Stroustrup, Bjarne. (2013, май). «The C++ Programming Language, 4th ed». Addison-Wesley.

[1] *Скотт Мейерс*. Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14. М.: Вильямс, 2015. — *Примеч. пер.*

Дж. Глейзер, С. Мадхав

Многопользовательские игры. Разработка сетевых приложений

Перевел с английского А. Киселев

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Римицан*
Литературный редактор	*А. Пасечник*
Художник	*С. Заматевская*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — Книги печатные профессиональные, технические и научные.

Подписано в печать 19.08.16. Формат 70x100/16. Бумага офсетная. Усл. п. л. 29,670. Тираж 700. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**