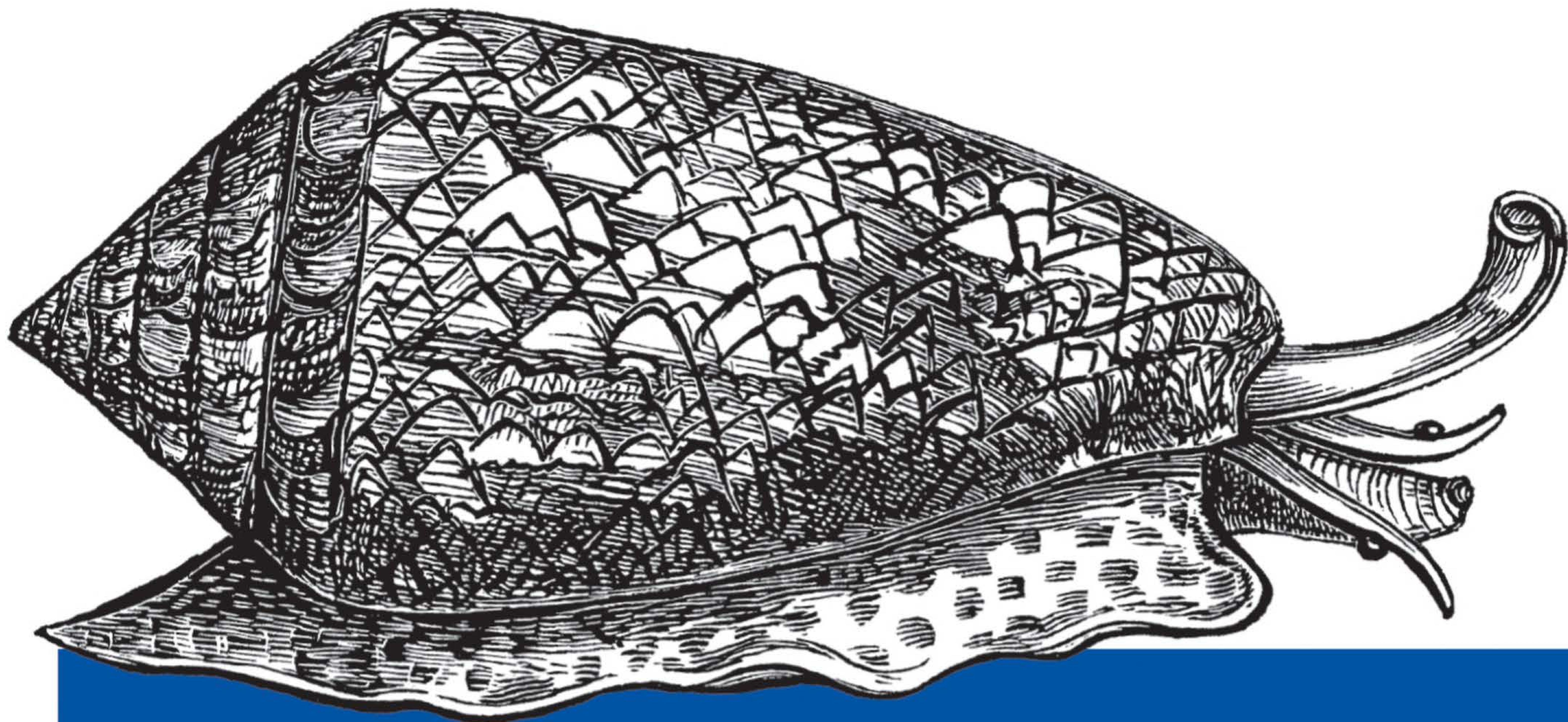
O'REILLY®



Масштабирование приложений

Выращивание сложных систем

ПИТЕР®

Ли Атчисон

Copyrighted material

Architecting for Scale

High Availability for Your Growing Applications

Lee Atchison

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Copyrighted material

Ли Атчисон

Масштабирование приложений

Выращивание сложных систем

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

2018

Copyrighted material

ББК 32.988.02
УДК 004.738.5
А93

Атчисон Ли

А93 Масштабирование приложений. Выращивание сложных систем. — СПб.: Питер, 2018. — 256 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-02952-0

Мы живем в мире растущих приложений. Практически любые программные продукты рано или поздно приходится расширять, надстраивать, адаптировать к обслуживанию растущей пользовательской аудитории и к пиковым нагрузкам. Для того чтобы подобное масштабирование протекало гладко и быстро, нужно закладывать такие возможности уже на уровне архитектуры приложения. В этой незаменимой прикладной книге автор рассказывает не только об архитектурных тонкостях, необходимых для эффективного масштабирования приложений, но и о рисках, присущих такой работе, о грамотной организации масштабирования и об использовании облачных сервисов.

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

ББК 32.988.02
УДК 004.738.5

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1491943397 англ.
Authorized Russian translation of the English edition of Architecting for Scale, ISBN 9781491943397
© 2016 Lee Atchison

ISBN 978-5-496-02952-0
© Перевод на русский язык ООО Издательство «Питер», 2018
© Издание на русском языке, оформление ООО Издательство «Питер», 2018
© Серия «Бестселлеры O'Reilly», 2018

Краткое содержание

Предисловие 14

Введение 17

Часть I. Доступность

Глава 1. Что такое доступность 28

Глава 2. Пять приоритетных направлений для улучшения доступности приложения 33

Глава 3. Измерение доступности 44

Глава 4. Улучшение неудовлетворительной доступности 48

Часть II. Управление рисками

Глава 5. Что такое управление рисками 58

Глава 6. Критичность и вероятность 65

Глава 7. Матрица рисков 71

Глава 8. Смягчение рисков 83

Глава 9. Дни большой игры 89

Глава 10. Создание систем со сниженными рисками 95

Часть III. Сервисы и микросервисы

Глава 11. Зачем нужны сервисы 106

Глава 12. Использование микросервисов 114

Глава 13. Обработка отказов сервисов 126

Copyrighted material

Часть IV. Масштабирование приложений

Глава 14. Запас на две ошибки 142

Глава 15. Владение сервисами 156

Глава 16. Классы сервисов 163

Глава 17. Использование классов сервисов 172

Глава 18. Соглашения сервисного уровня 178

Глава 19. Непрерывное совершенствование 192

Часть V. Облачные сервисы

Глава 20. Облака и перемены в них 202

Глава 21. Распределение облака 206

Глава 22. Управление инфраструктурой 216

Глава 23. Распределение облачных ресурсов 225

Глава 24. Другие средства масштабирования 235

Глава 25. AWS Lambda 243

Часть VI. Заключение

Глава 26. Общий обзор всех аспектов масштабирования 250

Copyrighted material

Оглавление

Предисловие 14

Введение 17

Для кого предназначено издание 17
Почему была написана эта книга 18
Современные проблемы масштабирования 18
Структура книги 19
Сетевые ресурсы 23
Соглашения, принятые в книге 23
Об авторе 24
Благодарности 24
Об обложке 26

Часть I. Доступность

Глава 1. Что такое доступность 28

Доступность и надежность 29
Что ухудшает доступность 31

Глава 2. Пять приоритетных направлений для улучшения доступности приложения 33

1. Учитывайте возможные отказы 35
2. Всегда помните о масштабировании 36
3. Смягчайте последствия рисков 38
4. Контролируйте доступность 40
5. Разработайте процедуру решения проблем с доступностью 42

Будьте подготовлены 43

Copyrighted material

Глава 3. Измерение доступности 44

Девятки 45

Что считать разумной доступностью 45

Не обманывайте себя 46

Доступность в цифрах 47

Глава 4. Улучшение неудовлетворительной доступности 48

Измеряйте и отслеживайте текущий уровень доступности 49

Автоматизируйте ручные процессы 50

Совершенствуйте свои системы 55

Рост и перемены в вашем приложении 56

Удерживайте доступность на высоком уровне 56

Часть II. Управление рисками

Глава 5. Что такое управление рисками 58

Управление рисками 59

Выявление рисков 60

Начните с самого страшного 62

Смягчите последствия 62

Регулярно пересматривайте матрицу 63

Принципы управления рисками 64

Глава 6. Критичность и вероятность 65

Список топ-10: низкая вероятность и низкая критичность риска 66

База данных с заказами: низкая вероятность и высокая критичность риска 67

Специфические шрифты: высокая вероятность и низкая критичность риска 68

Фотографии футболок: высокая вероятность и высокая критичность риска 70

Глава 7. Матрица рисков 71
Объем матрицы рисков 74
Создание матрицы рисков 75
Использование матрицы для планирования 79

Глава 8. Смягчение рисков 83
Планы восстановления 85
Планы аварийного восстановления 87
Улучшение ситуации 87

Глава 9. Дни большой игры 89
Среда: стейджинговая или продуктовая? 89
Недостатки запуска Дней большой игры в производственной среде 92
Дни большой игры 94

Глава 10. Создание систем со сниженными рисками 95
Избыточность 95
Примеры идемпотентных интерфейсов 96
Повышение избыточности, ведущее к росту сложности 97
Независимость 98
Безопасность 100
Простота 100
Самовосстановление 101
Оперативные процессы 103

Часть III. Сервисы и микросервисы

Глава 11. Зачем нужны сервисы 106
Монолитное приложение 106
Сервисно-ориентированное приложение 107
Преимущества выделенного владения сервисами 109
Преимущества масштабирования 112

Copyrighted material

Глава 12. Использование микросервисов 114
Что должно быть сервисом? 115
Не переходите границ разумного..................................... 123
Соблюдение баланса 124

Глава 13. Обработка отказов сервисов 126
Каскадные аварии сервисов 126
Реагирование на отказ сервиса 128
Сдавайтесь как можно раньше..................................... 136
Ошибки, создаваемые пользователями..................................... 138

Часть IV. Масштабирование приложений

Глава 14. Запас на две ошибки 142
Что такое запас на две ошибки?..................................... 143
Запас на две ошибки на практике..................................... 144
Управление приложениями 153
Космический корабль 154

Глава 15. Владение сервисами 156
Отдельная команда, владеющая сервисной архитектурой 156
Преимущества организаций и приложений, соответствующих принципам ОКВСА 159
Что значит быть владельцем сервиса 159

Глава 16. Классы сервисов 163
Сложность приложения 163
Что же такое классы сервисов..................................... 165
Присвоение сервисам меток сервисных классов 165
Что дальше?..................................... 171

Глава 17. Использование классов сервисов 172
Ожидания..................................... 172
Реагирование 173

Copyrighted material

Зависимости ... 174
Резюме ... 177

Глава 18. Соглашения сервисного уровня ... 178
Что такое соглашения сервисного уровня ... 178
Внутренние и внешние SLA ... 181
Почему внутренние SLA так важны ... 182
SLA создают доверие ... 182
SLA помогают в диагностике проблем ... 183
Измерение производительности для SLA ... 185
Сколько и какие внутренние SLA установить ... 190
Дополнительно о SLA ... 191

Глава 19. Непрерывное совершенствование ... 192
Регулярно проверяйте приложение ... 193
Микросервисы ... 193
Владение сервисами ... 193
Сервисы без сопровождения состояния ... 194
А где же данные? ... 194
Партиционирование данных ... 195
Значение непрерывного совершенствования ... 200

Часть V. Облачные сервисы

Глава 20. Облака и перемены в них ... 202
Что изменилось в облаках ... 202
Изменения продолжаются ... 205

Глава 21. Распределение облака ... 206
Архитектура AWS ... 206
Обзор архитектуры ... 208
Зоны доступности и дата-центры не одно и то же ... 212
Поддержка распределения локаций для обеспечения доступности ... 215

Глава 22. Управление инфраструктурой 216
Структура облачных сервисов 216
Особенности использования управляемых ресурсов 222
Особенности использования неуправляемых ресурсов 223
CloudWatch и мониторинг 224

Глава 23. Распределение облачных ресурсов 225
Ресурсы, основанные на выделении мощности 225
Ресурсы с расчетом задействования 231
Преимущества и недостатки технологий распределения ресурсов.. 234

Глава 24. Другие средства масштабирования 235
Облачные серверы 236
Вычислительные слайсы 237
Динамические контейнеры 239
Микровычисления 240
Дальнейшие действия 242

Глава 25. AWS Lambda 243
Использование Lambda.. 244
Преимущества и недостатки Lambda 247

Часть VI. Заключение

Глава 26. Общий обзор всех аспектов масштабирования 250
Доступность 250
Управление рисками.. 251
Сервисы 252
Масштабирование 253
Облачные технологии.. 253
Архитектура под масштабирование 254

Copyrighted material

Посвящается Бет

Copyrighted material

Предисловие

Мы живем в интересное время — его можно назвать периодом кембрийского взрыва в программном обеспечении. Стоимость создания новых систем упала на порядки, и на столько же порядков выросла их способность к взаимодействию. Такие ресурсы, как Amazon AWS, Microsoft Azure и Google GCP, позволяют физически масштабировать системы до размеров, казавшихся невероятными всего несколько лет назад.

Предлагая неведомые ранее экономические возможности и практически безграничные вычислительные мощности, эти ресурсы стимулируют мощный поток новых идей, продуктов и рынков. Но все эти возможности реальны только с учетом масштабируемости создаваемых систем. Построить что-то несложное сегодня проще, чем когда-либо, а вот систему, которую можно быстро и стабильно масштабировать, оказывается, создать куда сложнее: простого увеличения вычислительных мощностей и дискового пространства недостаточно.

У программных систем предсказуемый жизненный цикл. Он начинается с простых, хорошо реализованных решений, полностью понятных одному человеку. Затем они быстро растут, превращаясь в монолит технического долга. Потом дело доходит до случайного набора нестабильных сервисов, и наконец рождается оптимально спроектированная распределенная система, способная к росту вширь (по количеству пользователей) и вглубь (по количеству функциональностей). Нетрудно понять, к чему надо прийти снаружи (сделать систему более надежной!), гораздо сложнее увидеть маршрут к цели изнутри. К счастью, эта книга является незаменимым путеводителем в таком путешествии — автор описывает ключевые решения и практики масштабируемых систем: доступность, классы сервисов, плановые проверки, матрицы рисков.

Copyrighted material

Ли присоединился ко мне в New Relic, когда мы превращались из однопродуктового монолита в многопродуктовую фирму, наблюдая взрывной рост количества довольных клиентов, сделавший New Relic такой успешной. Он пришел из Amazon с большим опытом как в сфере розничной торговли, где Amazon добился больших успехов, так и в области веб-сервисов, где, как легко догадаться, он тоже очень успешен. Ли состоял в разных командах, руководил командами, активно участвовал в огромном количестве проектов, связанных с масштабированием, и набил кучу шишек на этом поприще. Получив горький опыт восстановления после самых сложных системных сбоев, Ли готов поделиться полученными уроками, чтобы нам не пришлось наступать на те же грабли.

Когда Ли устроился в New Relic, компания переживала переходный период. Примитивная монолитная система не могла угнаться за нашими успехами: ее доступность, надежность и производительность не соответствовали предъявляемым требованиям. Внедряя приемы, приведенные в этой книге, мы вышли из подросткового периода и построили гибкую корпоративную систему, существующую и по сей день. Одним из таких приемов стали четыре уровня обеспечения доступности: бронзовый, серебряный, золотой и платиновый. Для получения бронзового уровня команде достаточно иметь матрицу рисков и определить соглашения на уровне сервисов (Service Level Agreement, SLA). Чтобы получить серебряный уровень, команда должна следить за проблемами, определенными в матрице рисков, и использовать плановые проверки. Золотой уровень означает, что риски обработаны, а их возможные последствия смягчены. Платиновый уровень соответствует пятому уровню модели зрелости возможностей (Capability Maturity Model, CMM), на котором системы способны к самовосстановлению, а первоочередное внимание уделяется непрерывному совершенствованию. Основные усилия были направлены на сервисы первого класса (Tier 1), затем второго и т. д. Со временем мы довели все команды до серебряного уровня, большинство — до золотого, а несколько — и до платинового.

Когда я перешел на работу в InVision App, мне вновь пришлось работать в молодой компании, переходящей от ранних успехов

Copyrighted material

к взрывному росту, а значит, там я тоже применил и продолжаю применять те инструменты и практики, которые описывает Ли. И я призываю вас, если ваши системы, продукты или компании переживают период бурного роста, делать то же самое — учиться у Ли тому, как нужно создавать масштабируемые системы.

Бьорн Фримен-Бенсон,
кандидат наук,
главный технический директор
компании InVision App

Copyrighted material

Введение

По мере роста приложений с ними начинают происходить две вещи: во-первых, они становятся гораздо более сложными и, как следствие, хрупкими, во-вторых, им приходится обрабатывать гораздо большее количество запросов, управляемых более новыми и сложными механизмами. Приложение попадает в крутое пике — пользователи испытывают снижение производительности, отказы и прочие проблемы, связанные с качеством работы приложения и его доступностью.

Но ваших клиентов это мало интересует. Они всего лишь хотят использовать ваше приложение для решения своих задач и получать от него ожидаемые результаты. Если приложение работает медленно или нестабильно, а то и вовсе недоступно, они просто откажутся от его использования и пойдут искать конкурентов, которые смогут справиться с их запросами.

Эта книга научит вас базовым приемам проектирования крупных приложений и управления ими, которые позволят избежать таких ситуаций. Когда вы освоите эти приемы, ваши приложения смогут надежно обрабатывать большие объемы запросов и справляться с резкими скачками их количества, не теряя ожидаемого клиентами качества.

Для кого предназначено издание

Эта книга предназначена для программистов, системных архитекторов, технических руководителей и директоров, которые создают и эксплуатируют крупномасштабные приложения и системы. Если вы управляете программистами, инженерами по надежности систем, специалистами DevOps или руководите организацией, применяющей крупномасштабные приложения и системы, то приведенные в книге рекомендации и руководства помогут вам заставить эти приложения работать более гладко и надежно.

Copyrighted material

Если ваше приложение изначально было небольшим, а сейчас переживает невероятный рост (и сопутствующие ему болезни роста), его надежность и работоспособность могут снизиться. Если вы не можете справиться с техническими долгами и связанными с ними отказами приложения, эта книга покажет вам, как отработать эти долги и таким образом сделать приложение более масштабируемым.

Почему была написана эта книга

Проведя многие годы в Amazon за проектированием крупномасштабных приложений в области как розничной торговли, так и веб-сервисов Amazon (AWS), я перешел на работу в компанию New Relic, которая переживала период взрывного роста. В компании ощущали болезненную необходимость в системах и процессах для управления крупномасштабными приложениями, но пока не пришли к четко установленному порядку масштабирования своего приложения.

В New Relic я лицом к лицу столкнулся с проблемами компании, масштабирующей свое приложение. Тогда и осознал, что многие компании сталкиваются с теми же проблемами ежедневно.

Цель этой книги — помочь людям, работающим с быстро растущими приложениями, изучить процессы и лучшие практики, которые помогут избежать подводных камней при масштабировании.

Неважно, выросло ли ваше приложение за год в десять раз или на 10 %, выражается ли рост в количестве пользователей, транзакций, хранимых данных или сложности кода, — данная книга поможет вам создавать и поддерживать приложения, способные выдержать рост, но сохраняющие при этом высокий уровень доступности.

Современные проблемы масштабирования

Облачные сервисы растут и расширяются невероятно высокими темпами. Подход «программное обеспечение как сервис» (Software as a Service, SaaS) становится нормой в современной практике разработки в первую очередь из-за наличия спроса на облачные сервисы.

Copyrighted material

SaaS-приложения особо чувствительны к проблемам масштабирования в силу своей многопользовательской природы.

Мир меняется, все больше внимания уделяется SaaS, облачным сервисам и приложениям, обрабатывающим большие объемы данных. Росту размеров и сложности облачных приложений не видно конца.

Механизмы, представляющие сегодня последнее слово техники в области управления масштабированием, завтра станут не более чем базовыми строительными блоками. Решения современных проблем масштабирования будут выглядеть просто и минималистично по сравнению с решениями будущих проблем. Индустрия программного обеспечения будет требовать все более и более сложных систем, способных выдержать будущие нагрузки.

Цель этой книги — предоставить информацию, которая будет актуальна не один год.

Структура книги

Управление масштабированием — это не только управление количеством поступающих запросов, но и управление рисками и доступностью. Все эти аспекты часто описывают одну и ту же проблему с разных сторон, все они идут рука об руку. Следовательно, чтобы в полной мере рассмотреть вопрос масштабирования, необходимо рассмотреть и вопросы доступности, управления рисками, а также современные архитектурные парадигмы, такие как микросервисы и облачные вычисления.

Таким образом, данная книга имеет следующую структуру.

Часть I. Доступность

Доступность и управление доступностью — те области, которые часто первыми ощущают последствия роста приложения.

Глава 1. Что такое доступность. Для начала определим, что такое высокая доступность и чем она отличается от надежности.

Глава 2. Пять приоритетных направлений для улучшения доступности приложения. В этой главе я перечисляю пять основных

моментов, на которых при создании приложения следует сосредоточиться для улучшения его доступности.

Глава 3. Измерение доступности. В этой главе рассмотрен стандартный алгоритм измерения доступности, а также глубже исследовано понятие высокой доступности.

Глава 4. Улучшение неудовлетворительной доступности. Если ваше приложение труднодоступно или вы хотите предупредить возникновение этой проблемы в будущем, то приведенные в данной главе организационные шаги помогут вам повысить доступность приложения.

Часть II. Управление рисками

Понимание того, в чем заключаются риски для вашей системы, жизненно необходимо как для повышения ее доступности, так и для улучшения ее масштабируемости до уровня, необходимого сегодня и в будущем.

Глава 5. Что такое управление рисками. В этой главе начинаем раскрывать тему управления рисками в крупномасштабных приложениях, рассматривая основные понятия и суть этой задачи.

Глава 6. Критичность и вероятность. В этой главе рассматриваются различия между критичностью риска и вероятностью его реализации. Оба эти фактора важны, но каждый по-своему.

Глава 7. Матрица рисков. В данной главе я представляю систему для понимания рисков в крупномасштабных приложениях и управления ими.

Глава 8. Смягчение рисков. В этой главе обсуждается, как можно намеренно пойти на риск, в то же время минимизируя негативные последствия для приложения.

Глава 9. Дни большой игры. В этой главе рассматриваются текущая проверка и оценка ваших планов управления рисками, смягчения последствий и восстановления после катастрофических отказов. В ней также приводятся методики их использования в производственной среде и их преимущества.

Copyrighted material

Глава 10. Создание систем со сниженными рисками. В этой главе я привожу рекомендации относительно того, как уменьшить риски в ваших приложениях и создавать приложения с меньшими рисками.

Часть III. Сервисы и микросервисы

Сервисы и микросервисы — архитектурная стратегия построения крупных и сложных приложений, работающих с большими объемами запросов.

Глава 11. Зачем нужны сервисы. Из этой главы мы узнаем, почему сервисы важны для создания масштабируемых приложений.

Глава 12. Использование микросервисов. В этой главе я даю введение в создание архитектур, основанных на микросервисах. Особое внимание уделяется размерам сервисов и определению их границ с целью улучшения масштабируемости и доступности.

Глава 13. Обработка отказов сервисов. В последней главе этой части мы рассмотрим, как создавать сервисы, грамотно обрабатывающие отказы.

Часть IV. Масштабирование приложений

Масштабирование касается не только количества запросов, но всей вашей организации и ее способности справляться с потребностями растущего приложения.

Глава 14. Запас на две ошибки. В этой главе описывается, как масштабировать вашу систему и обеспечить ее высокую доступность даже при возникновении сбоев.

Глава 15. Владение сервисами. В данной главе мы рассмотрим, как важно уделять внимание владению сервисами для обеспечения масштабирования вашей организации и приложения.

Глава 16. Классы сервисов. Эта глава описывает методику определения значимости ваших сервисов, в результате чего можно предъявлять к их работе определенные требования.

Глава 17. Использование классов сервисов. После того как мы дали определение сервисных классов, начнем применять их для управления

Copyrighted material

последствиями рисков, требованиями ко времени реакции и ожиданиями пользователей.

Глава 18. Соглашения сервисного уровня. В этой главе рассмотрим SLA как механизм управления взаимозависимостями владельцев сервисов.

Глава 19. Непрерывное совершенствование. В этой главе приводятся методики и рекомендации по улучшению масштабируемости приложения в целом.

Часть V. Облачные сервисы

Облачные сервисы начинают играть все более важную роль в создании крупных критических приложений с высокими требованиями к масштабируемости и управлении ими.

Глава 20. Облака и перемены в них. В этой главе рассматривается, как облачные вычисления изменили подходы к построению высокомасштабируемых приложений.

Глава 21. Распределение облака. В данной главе в общих чертах описывается, как эффективно использовать регионы и зоны доступности для улучшения доступности и масштабируемости.

Глава 22. Управление инфраструктурой. В этой главе рассматривается использование управляемых сервисов, таких как RDS, SQS, SNS и SES, для масштабирования приложений и снижения затрат на управление.

Глава 23. Распределение облачных ресурсов. В данной главе мы обсудим, как распределяются ресурсы в облаке и как различные стратегии их распределения влияют на масштабируемость вашего приложения.

Глава 24. Другие средства масштабирования. В этой главе рассматриваются такие модели масштабируемых вычислений, как AWS Lambda. Они используются для улучшения масштабируемости, доступности и управляемости приложений.

Глава 25. AWS Lambda. В последней главе этой части более подробно рассматривается AWS Lambda — технология, предоставляющая

Copyrighted material

сверхширокие возможности для масштабирования событий с небольшими требованиями к вычислительной мощности.

Часть VI. Заключение

Глава 26. Общий обзор всех аспектов масштабирования. В данной главе кратко обобщаются важнейшие темы, рассмотренные в книге. Это позволит вам вспомнить, о чем говорилось в каждой из глав.

Сетевые ресурсы

Сайт книги http://www.architectingforscale.com/ содержит полезную информацию, касающуюся данного издания, включая ссылки на дополнительные материалы (http://bit.ly/architectbookdl). Вы можете узнать обо мне больше на моем сайте http://www.leeatchison.com/. Вы также можете подписаться на мой блог (http://bit.ly/leeatscale).

Соглашения, принятые в книге

В книге приняты следующие шрифтовые соглашения.

Курсив

Обозначает новые термины и информацию, на которую нужно обратить особое внимание.

Рубленый шрифт

Применяется для адресов URL и электронной почты.

Этот элемент обозначает общее замечание.

Этот элемент обозначает совет или указание.

Copyrighted material

Об авторе

Ли Атчинсон — главный облачный архитектор New Relic. Он проработал там четыре года. За это время Ли спроектировал продукты инфраструктуры New Relic и руководил их разработкой, а также помог New Relic создать надежную сервисно-ориентированную архитектуру, которая масштабировалась на протяжении всего времени, пока организация росла от простого стартапа SaaS до высоконагруженного крупного предприятия. Наибольший опыт Ли получил при разработке высокодоступных систем.

Ли работает в индустрии 28 лет, изучал облачные масштабируемые системы все семь лет работы главным менеджером в Amazon.com. В Amazon он руководил созданием первого в компании магазина загружаемого программного обеспечения, создал AWS Elastic Beanstalk и руководил командой, переводившей интернет-магазин Amazon с монолитной на сервисно-ориентированную архитектуру.

Благодарности

Хотя при создании книги мне помогало больше людей, чем я могу здесь перечислить, хотелось бы особо упомянуть тех, кто внес наибольший вклад.

Бьорн Фримен-Бенсон, который оказал мне значительную помощь на ранних этапах создания книги и дал возможность осознать суть вещей, описываемых здесь, во время работы в New Relic.

Кевин Макгуайр, который был моим другом и доверенным лицом. Мы начали работать в New Relic вместе, и благодаря его предусмотрительности и воображению моя карьера приобрела необходимое направление, в котором я двигаюсь и по сей день.

Наташа Литт — хороший друг, источник поддержки и одобрения.

Джейд Рубик — улыбчивый человек с позитивным взглядом на жизнь, который давал мне аргументированные советы и рекомендации. Какой хороший друг!

Джим Гочи познакомил меня с волшебным миром под названием New Relic — фирмой, продуктом и моей будущей карьерой.

Copyrighted material

Лью Цирн, чья дальновидность подарила нам New Relic, а мне — карьеру и дом. Радость и безудержный энтузиазм после встречи с Лью один на один чрезвычайно заразительны и дают прилив сил. Неудивительно, что New Relic настолько успешна.

Абнер Германов, Джей Фрай, Бхарат Гоуда и Робсон Грив поверили в меня и поборолись за мое нынешнее место в New Relic. Кто сказал, что нельзя засунуть кубик в круглое отверстие, да еще так, чтобы он подошел? Это, несомненно, самая интересная, сто́ящая и приносящая наибольшее удовлетворение работа, которую я когда-либо выполнял.

Майки Батлер, Ник Бендерс, Мэтью Флеминг и другие ведущие инженеры New Relic — спасибо за вашу поддержку на протяжении многих лет.

Курт Куфельд, который был моим наставником и помог мне найти место в странном, хаотичном, утомительном, но все равно приносящем невероятное удовлетворение рабочем окружении компании Amazon.

Грег Харт, Скотт Грин, Патрик Франклин, Суреш Кумар, Колин Боделл, Энди Джесси, обеспечившие мне невообразимые перспективы в Amazon и AWS.

Брайан Андерсон — редактор, который сделал на меня ставку при написании книги и помогал мне на всех этапах этого пути.

Я хотел бы выразить глубочайшую благодарность Абнеру Германову и Бьорну Фримену-Бенсону. Они сделали возможной мою работу над этой книгой. Она не появилась бы на свет без их поддержки. За это я им буду вечно благодарен.

Я благодарен своей семье, в особенности жене Бет — моей путеводной звезде в совместной жизни. Мои дни ярче и мой путь чище благодаря тому, что она со мной.

Я сердечно благодарен всем этим людям и всем тем, кого не упомянул.

Не могу не упомянуть моих пушистых друзей: храпящего спаниеля Иззи, веселого корги Эбби и безумного кота Будду, который внес львиную долю опечаток в книгу.

Copyrighted material

Об обложке

Животное на обложке Architecting for Scale — парчовый конусный моллюск (Conus Textile), известный также как узорчатый конус из-за уникального желто-коричневого с белым узора на раковине, длина которой 8–10 см. Парчовый конус обычно обитает на мелководье Красного моря, на побережьях Австралии и Западной Африки, а также в тропических регионах Тихого океана.

Как и остальные члены семейства Conus, парчовый конус является хищником и поедает других моллюсков. Он убивает свою добычу, вводя яд из радулы — нароста, напоминающего маленькую иглу. Яд, используемый этим моллюском, чрезвычайно опасен и может повлечь за собой паралич и смерть.

Парчовый конус откладывает за раз несколько сотен яиц, которые вырастают во взрослых особей. Иногда ракушки продаются как брелоки, но популяция настолько многочисленна, что моллюску это никак не угрожает.

Большинство животных на обложках книг O'Reilly находятся на грани вымирания, все они — важная часть природы. Чтобы узнать о них больше, посетите страницу animals.oreilly.com.

Изображение для обложки взято из Wood's Illustrated Natural History.

Copyrighted material

Часть I
Доступность

Если у вас нет высокой доступности, вам не требуется масштабирование.

Copyrighted material

1 Что такое доступность

Никого не заинтересуют замечательные возможности вашей системы, если ее невозможно использовать.

Одним из важнейших аспектов архитектуры распределенных систем является доступность. Хотя есть такие компании и сервисы, для которых временная недоступность оправданна и допустима, большая часть фирм не может позволить себе ни секунды недоступности, так как ее последствия неизбежно повлияют на удовлетворенность клиентов и, как следствие, на финансовое благополучие фирмы.

Далее приведены фундаментальные вопросы, которыми должны задаваться все компании, которые хотят определить важность доступности систем для себя и своих клиентов. Эти вопросы и очевидные на них ответы лежат в основе того, почему доступность критична для крупномасштабных приложений.

- Зачем кому-то покупать ваш сервис, если он не будет работать в нужный момент?
- Что думают или чувствуют ваши клиенты, когда у них возникает потребность в нашем сервисе, а он не работает?
- Как удовлетворить наших клиентов, заработать средства для компании, выполнить обязательства и требования, когда сервис недоступен?

Клиенты будут довольны вашим продуктом и заинтересованы в нем только тогда, когда система работает. Существует прямая и весьма

Copyrighted material

существенная корреляция между доступностью системы и удовлетворенностью клиентов.

Высокая доступность настолько важна для построения масштабируемых систем, что я посвящу этой теме значительную часть книги. Как создать систему (сервис, приложение, среду) с высокой доступностью, когда к ней предъявляется и без того широкий спектр требований?

В этой главе мы определим, что такое доступность и как она соотносится с надежностью. Это будет использоваться в последующих главах, когда мы начнем обсуждать, какую роль доступность играет в проектировании высокомасштабируемых приложений.

Большая игра

Сегодня воскресенье — день важного матча. Вы пригласили 20 близких друзей на просмотр матча на новом 300-дюймовом телевизоре. Все пришли, дом полон еды и пива. Все веселятся. Игра вот-вот начнется. И...

...пропадает электричество;

...телевизор гаснет;

...для вас и ваших друзей матч окончен.

В раздражении вы снимаете трубку и набираете номер энергетической компании. Представитель компании, не проявляя ни капли сочувствия, сообщает: «Приносим извинения, но мы гарантируем только 95%-ную доступность электросети».

Почему важна доступность? Потому, что клиенты рассчитывают на то, что ваши сервисы работают постоянно. Доступность менее 100 % может обернуться катастрофой для вашего бизнеса.

Доступность и надежность

Доступность и надежность — похожие, но очень разные понятия. Очень важно понимать разницу между ними.

В нашем контексте надежность в общем случае означает качество системы. Обычно так называют способность системы стабильно

работать в соответствии со спецификациями. Программное обеспечение считается надежным, если оно проходит набор тестов и делает то, что от него ожидается.

В нашем контексте доступность в общем случае означает способность системы выполнять задачи, которые она способна выполнять. Есть ли система? Работает ли она? Отвечает ли она на запросы? Утвердительный ответ на все три вопроса означает, что система доступна.

Как видим, доступность и надежность очень похожи. Систему трудно назвать доступной, если она ненадежна. Систему трудно назвать надежной, если она недоступна.

Однако, когда речь идет о надежности программного обеспечения, обычно говорят о способности программного обеспечения исполнять возложенные на него обязанности. В общем, основным индикатором доступности является то, проходит ли программное обеспечение все наборы тестов.

Кроме того, когда мы имеем в виду доступность системы, мы говорим о том, что система развернута и работоспособна. Ответит ли она на запрос, который я отправил?

Вот что мы подразумеваем при использовании этих терминов.

- *Надежность*. Способность системы выполнять требуемые действия, не допуская ошибок.
- *Доступность*. Готовность системы к работе при необходимости выполнить требуемые действия.

Система, которая при сложении 2 и 3 получает 6, характеризуется плохой надежностью. Система, которая при сложении 2 и 3 не возвращает ответа, характеризуется плохой доступностью. Надежность может быть улучшена путем тестирования. Доступность улучшить обычно гораздо сложнее.

Можно привнести в программу ошибку, которая приведет к тому, что 2 + 3 будет равно 6. Такую ошибку легко отловить и исправить с помощью набора тестов.

Допустим, есть приложение, которое надежно выдает результат 2 + 3 = 5. А теперь представим, что оно запущено на компью-

тере с нестабильным сетевым подключением. Каков результат? Когда вы запускаете приложение, оно иногда возвращает 5, а иногда не возвращает ничего. Приложение может быть надежным, но при этом недоступным.

Эта книга в основном посвящена созданию систем с высоким уровнем доступности. Предположим, что ваша система надежна. Допустим также, что вы знаете, как создавать и запускать наборы тестов. Мы будем обсуждать надежность только тогда, когда она непосредственно влияет на архитектуру системы или ее доступность.

Что ухудшает доступность

Что заставляет приложение, которое раньше работало нормально, демонстрировать низкую доступность? Причин много.

- *Исчерпание ресурсов*. При увеличении количества пользователей или объема данных, требуемых системе, ваше приложение может стать жертвой исчерпания ресурсов, что приведет к снижению производительности и замедлению работы.
- *Незапланированные изменения, связанные с увеличением нагрузки*. Рост популярности приложения может потребовать внести изменения в код, чтобы обеспечить поддержку более высокой нагрузки. Эти изменения часто реализуются на скорую руку, в последний момент, без должного планирования и продумывания, что увеличивает вероятность появления ошибок.
- *Увеличение количества подвижных частей*. По мере того как приложение набирает популярность, часто необходимо подключать к его поддержке все больше и больше разработчиков, проектировщиков, тестировщиков и других людей. Множество работающих над приложением создают большое количество подвижных частей в результате добавления новых или изменения существующих возможностей или обычного технического обслуживания. Чем больше людей трудится над приложением, тем больше в нем подвижных частей и тем больше вероятность их некорректного взаимодействия.

Copyrighted material

- *Внешние зависимости*. Чем сильнее ваше приложение зависит от внешних ресурсов, тем больше оно подвержено проблемам, связанным с их доступностью.
- *Технический долг*. Усложнение приложений обычно ведет к росту технического долга (накоплению желаемых изменений в программном обеспечении и ожидающих исправления багов, связанному с ростом и развитием приложения). Технический долг увеличивает вероятность появления проблем.

Все быстрорастущие приложения сталкиваются с одной, несколькими или всеми перечисленными проблемами. Вследствие этого в приложениях, ранее работавших безупречно, могут появиться проблемы с доступностью. Часто они подкрадываются незаметно и возникают неожиданно.

Но большинство растущих приложений имеют одну и ту же проблему: они со временем начинают испытывать сложности с доступностью.

Проблемы с доступностью стоят денег вам и вашим клиентам. Они стоят доверия и преданности клиентов. Ваша компания долго не просуществует, если у нее будут постоянные проблемы с доступностью.

Создание масштабируемых приложений означает создание приложений с высоким уровнем доступности.

Copyrighted material

2 Пять приоритетных направлений для улучшения доступности приложения

Построить масштабируемое приложение с высоким уровнем доступности нелегко, это не произойдет само собой. Проблемы могут проявиться неожиданно, в результате чего даже абсолютно исправное приложение не будет работать у некоторых или у всех клиентов.

Проблемы доступности часто возникают там, где их меньше всего ожидаешь, а некоторые из наиболее серьезных проблем вызывают самые незначительные причины.

Недоступность аватара

Когда-то я работал над приложением, где не была проработана ситуация возможной недоступности зависимостей, из-за чего однажды произошел серьезный сбой. Приложение предоставляло клиентам некую услугу. Вверху каждой страницы находился настраиваемый

Copyrighted material

аватар, соответствующий вошедшему в систему пользователю. Изображение генерировала сторонняя система.

Однажды она отказала. Приложение, предполагавшее, что сторонняя система будет работать всегда, не знало, что делать, и поэтому падало. Все приложение падало из-за того, что система, генерировавшая изображения пользователей, — функция, по сути, несущественная — отказала.

Как можно было избежать этой проблемы? Если бы мы предвидели, что сторонняя система может отказать, то отработали бы сценарий недоступности внешнего ресурса при проектировании и обнаружили, что приложение упадет. Тогда добавили бы логику для обнаружения недоступности внешнего ресурса и удаления изображения при отказе внешнего сервиса. Можно также перехватывать ошибку при ее возникновении, предотвращая тем самым ее распространение и влияние на работоспособные части приложения.

Простая проверка и наличие логики восстановления предотвратили бы отказ приложения. Но вместо этого приложение было недоступным в течение длительного времени. А все из-за отсутствия аватара.

Никто не может предсказать источник проблем, и никакой объем тестирования не выявит все недостатки. Многие из этих проблем коренятся не в коде, а в самой системе. Чтобы их обнаружить, необходимо подняться на уровень выше и системно изучить приложение и порядок его работы. Приведу пять пунктов, на которых надо сосредотачивать внимание при проектировании систем, остающихся высокодоступными при масштабировании вверх.

1. Учитывайте возможные отказы.
2. Всегда помните о масштабировании.
3. Смягчайте последствия рисков.
4. Контролируйте доступность.
5. Разработайте процедуру решения проблем с доступностью.

Рассмотрим каждый из них по отдельности.

Copyrighted material

1. Учитывайте возможные отказы

Технический директор Amazon Вернер Вогельс говорит: «Все постоянно ломается». Всегда подразумевайте, что ваши приложения и сервисы будут отказывать. Это когда-нибудь случится. Будьте готовы.

Ваше приложение откажет. Вопрос в том, каким образом это произойдет. При создании системы учитывайте нюансы доступности во всех аспектах проектирования и построения, например.

- *Проектирование*. Какие шаблоны проектирования и типовые конструкции вы рассматриваете или применяете для улучшения доступности своего приложения?

  Использование при проектировании шаблонов и типовых конструкций, таких как перехват ошибок в глубине приложения, повторения запроса, прерывателей, позволяет перехватывать ошибки тогда, когда они еще существенно не повлияли на функциональность приложения. Это позволяет ограничить область воздействия проблемы и дает возможность сделать так, чтобы приложение предоставляло полезный функционал даже тогда, когда его часть отказала.

- *Зависимости*. Что вы делаете, когда компонент, от которого вы зависите, отказывает? Как повторяете запросы? Что делаете, когда случается невосстановимый (катастрофический) отказ, а не просто временный сбой?

  Прерыватели наиболее полезны для обработки проблем с зависимостями, поскольку снижают влияние неисправных зависимостей на вашу систему.

  Не используя таких шаблонов, можно вызвать снижение производительности приложения из-за неполадок с зависимостями (например, из-за неприемлемо долгого интервала ожидания ответа от недоступного сервиса). Применяя прерыватель, можно отказаться от дальнейших попыток использовать зависимость до тех пор, пока она не восстановит работоспособность.

- *Клиенты*. Что вы делаете, когда компонент, являющийся клиентом вашей системы, начинает вести себя некорректно? Выдерживает

Copyrighted material

ли система избыточную нагрузку? Можете ли вы ограничивать избыточный трафик? Корректно ли обрабатываются «мусорные» входные данные? А как насчет избыточных данных?

Иногда атаки типа «отказ в обслуживании» инициируются доверенными источниками. Например, клиент вашего приложения может столкнуться с неожиданным ростом активности, которая вызовет увеличение количества запросов к приложению. Или, к примеру, ошибка в приложении вашего клиента может привести к тому, что оно вызывает методы вашей программы с неприемлемо высокой интенсивностью. Что вы делаете, когда такое случается? Приводит ли неожиданное увеличение трафика к отказу приложения? Можете ли вы обнаружить эту проблему и ограничить частоту запросов, ограничивая или устраняя тем самым влияние на ваше приложение?

2. Всегда помните о масштабировании

Если ваше приложение работает сегодня, это не значит, что оно будет работать завтра. Трафик веб-приложений имеет тенденцию расти со временем. Сайт, генерирующий определенное количество трафика сегодня, может начать генерировать существенно больше трафика раньше, чем ожидалось. Создавайте системы не с учетом сегодняшних объемов трафика, а с учетом завтрашних.

В частности, это может означать следующее.

- Закладывайте в архитектуру возможность увеличения размера и емкости своих баз данных.
- Поразмышляйте о том, каковы логические границы масштабирования данных вашего приложения. Что случится, если база данных достигнет предела своих возможностей? Определите эти границы и отодвиньте их прежде, чем достигнете.
- Создавайте приложения так, чтобы легко было добавить дополнительные серверы приложений. Это часто требует внимательно

относиться к тому, где и как хранится состояние и как маршрутизируется трафик[1].

- Перенаправьте запросы статичного содержимого внешним провайдерам. Это позволит вашей системе обрабатывать только динамическое содержимое, для чего она, собственно, и разрабатывалась. Использование сетей доставки контента (Content Delivery Network, CDN) не только уменьшает количество обрабатываемого системой трафика, но и позволяет задействовать эффект масштаба CDN, чтобы быстрее доставлять статичную информацию клиентам.
- Подумайте, какие части динамического контента могут быть сгенерированы статически. Часто информация, которая выглядит динамической, на самом деле большей частью статична. Масштабируемость приложения можно улучшить, сделав ее статической. Такие «статическо-динамические» данные часто прячутся в неожиданных местах, что рассматривается в следующем примере.

КОНТЕНТ: СТАТИЧЕСКИЙ ИЛИ ДИНАМИЧЕСКИЙ?

Часто контент, кажущийся динамическим, на самом деле большей частью статичен. Представьте типичный баннер на простом сайте. Такой контент часто статичен, но иногда в него могут включаться динамические элементы.

Например, верхняя часть страницы может содержать сообщение «Вход», если вы не вошли в систему, и «Здравствуйте, Ли», если вход уже выполнен (и если вас зовут Ли, разумеется).

Значит ли это, что вся страница должна генерироваться динамически? Отнюдь не обязательно. Вся страница, за исключением части, отвечающей за вход в систему и приветствие пользователя, статична и может без проблем предоставляться серверами CDN, не требуя от вас никаких вычислений.

Если большая часть баннера статична, можно в браузере пользователя добавлять изменяемое содержимое динамически («Вход», «Здравствуйте, Ли» и др.). Группируя динамические данные и обрабатывая их отдельно от статических, можно

[1] Эта тема довольно обширна и может занять целую главу или даже книгу.

увеличить производительность веб-страницы и уменьшить объем работы по генерации динамического контента, выполняемой вашим приложением. Это улучшает масштабируемость и в конечном счете доступность.

3. Смягчайте последствия рисков

Поддержание высокой доступности системы требует избавления ее от рисков. Причину отказа часто можно идентифицировать как риск еще до того, как она реализуется. Идентификация рисков — ключевой метод повышения доступности. Все системы несут в себе следующие риски:

- риск отказа сервера;
- риск повреждения базы данных;
- риск возврата некорректного ответа;
- риск неисправности сетевого подключения;
- риск отказа вновь развернутого программного модуля.

Поддержание доступности системы требует ликвидации рисков. Но по мере того, как системы становятся все более и более сложными, задача все больше усложняется. Поддержка доступности крупной системы заключается скорее в идентификации риска, принятии допустимого объема риска и мер по смягчению его последствий.

Это называется *управлением рисками*. Подробно оно будет рассмотрено в части II. Управление рисками — основа построения высокодоступных систем.

Частью управления рисками является *смягчение последствий рисков*. Смягчать последствия рисков — значит знать, что делать при возникновении проблемы, чтобы максимально уменьшить ее воздействие на приложение. Суть смягчения последствий состоит в том, что ваше приложение должно работать настолько хорошо и настолько полно, насколько возможно, даже если сервисы и ресурсы испытывают проблемы. Смягчение последствий рисков под-

Copyrighted material

разумевает определение того, что может пойти не так, и составление всеобъемлющего плана быстрого выхода из нештатной ситуации прежде, чем она возникнет.

Пример 2.1. Смягчение последствий рисков — интернет-магазин в отсутствие поиска

Допустим, у нас есть интернет-магазин, где продаются футболки. С приложением такого типа знаком каждый: футболки можно просматривать прямо на начальной странице, можно просматривать разные категории футболок с помощью системы навигации, а можно воспользоваться поиском, чтобы найти какую-нибудь особенную футболку.

Для внедрения системы поиска магазины такого рода, как правило, прибегают к услугам отдельного поискового движка, который может быть отдельным сервисом или же сторонней поисковой системой. Но поскольку поиск в этом случае является сторонним компонентом, ваше приложение подвергается риску в случае возможного сбоя поискового сервиса. Выявив эту проблему, необходимо занести ее в план управления рисками как «Сбой поискового движка», представляющий угрозу для приложения.

Без плана смягчения последствий рисков сбой в работе поисковой системы вызовет появление страницы с ошибкой или, возможно, генерацию некорректных либо невалидных результатов поиска. В любом случае у клиента останутся самые негативные впечатления о нашем сервисе.

В плане смягчения последствий рисков может быть написано, скажем, следующее: «Мы знаем, что наш самый популярный товар — футболки в красную полоску. Шестьдесят процентов людей, которые пользовались на сайте поиском, интересовались именно этими футболками (и хочется надеяться, в конце концов купили их). Поэтому, если поисковый сервис перестанет функционировать, мы отобразим страницу с извинениями, а также предложим список самых популярных товаров, включая футболки в красную полоску. Таким образом, покупатели, оказавшиеся на странице с ошибкой, смогут по крайней мере просмотреть футболки, которые, как показывает статистика покупок, являются наиболее популярными.

Copyrighted material

Кроме того, мы отобразим для этих покупателей скидочный код на 10 % со следующей покупки, в результате те из них, кто не смог найти требуемого, получат стимул вернуться на наш сайт позже, когда работа поиска будет восстановлена, а не отправиться за футболками куда-нибудь еще».

Таким образом, в примере 2.1 мы видим процесс смягчения рисков: выявление риска, определение мер смягчения его последствий и внедрение этих мер в план управления рисками.

Действуя подобным образом, вы, скорее всего, обнаружите ранее неизвестные проблемы в своем приложении, которые лучше будет исправить незамедлительно, не ожидая их возникновения. Кроме того, это поможет создать правила и процедуры обработки известных видов сбоев, в результате чего последствия ошибки уменьшатся за счет снижения ее серьезности или длительности существования.

Доступность и управление рисками идут рука об руку. Суть построения высокодоступной системы во многом состоит в управлении рисками.

4. Контролируйте доступность

Вы не знаете, есть ли в вашем приложении сбои, если никак не можете их обнаружить. Обеспечьте приложение средствами для внутреннего и внешнего мониторинга его функционирования.

Реализация полноценного мониторинга зависит от ваших нужд и специфики приложения, но, как правило, нужно обеспечить следующие возможности:

- *серверный мониторинг* — контроль рабочего состояния серверов и эффективности их работы;
- *мониторинг изменений конфигурации* — наблюдение за конфигурацией вашей системы, выявляющее изменения в инфраструктуре, повлиявшие на работу приложения;
- *мониторинг производительности приложения* — внутренний контроль за должной эффективностью работы ваших приложения и сервисов;

Copyrighted material

- *синтетическое тестирование* — проверка в реальном времени, как функционирует приложение с точки зрения пользователей. Это нужно для того, чтобы отлавливать ошибки в приложении, с которыми могли бы столкнуться пользователи, прежде чем это произойдет на самом деле;
- *оповещения* — при возникновении сбоя соответствующий персонал получает сигнал об этом, в результате чего проблему можно быстро и эффективно решить, минимизировав ее влияние на пользователей.

Хороших систем мониторинга, как платных, так и бесплатных, существует много. Я рекомендую New Relic: эта система включает в себя все перечисленные возможности мониторинга и оповещения. Являясь SaaS (Software as a Service — программное обеспечение как сервис), New Relic поддерживает мониторинг практически любого масштаба, который может потребоваться для вашего приложения[1].

Запустив мониторинг своих приложения и сервисов, начинайте отслеживать закономерности в производительности. Выявив какие-либо закономерности, обратите внимание на отклонения и рассматривайте их как потенциальные проблемы доступности. Вы можете, например, настроить систему оповещения так, чтобы она выдавала предупреждение в случае возникновения этих отклонений, прежде чем все приложение упадет. Кроме того, можно отслеживать рост вашей системы и следить за тем, чтобы план масштабирования оставался актуальным.

Установите внутренние нормативные показатели для коммуникации «сервис — сервис» и фиксируйте их в течение длительного времени. Таким образом, заметив проблемы с производительностью или доступностью, вы легко сможете понять, в каких сервисе или системе что-то идет не так, и выявить ошибку. Кроме того, вы можете определить

---

[1] Я хочу подчеркнуть, что рекомендую New Relic не потому, что там работаю. Я познакомился с этим инструментом и начал его использовать еще до того, как перешел работать в компанию. Успешное использование инструментов New Relic для решения проблем с производительностью и доступностью послужило причиной моего перехода на работу в эту компанию, а не наоборот.

проблемные места — области, где производительность не соответствует желаемым показателям, и внести решение этих проблем в свой план разработки.

5. Разработайте процедуру решения проблем с доступностью

В мониторинге системы нет никакого толку, если вы не готовы решить проблему, как только ее заметите: вы начинаете действовать сразу после получения оповещения. Кроме того, необходимо разработать процессы и процедуры, которых может придерживаться ваша команда, чтобы совместно установить источник проблемы, а затем быстро исправить самые распространенные ошибки.

Например, если какой-либо сервис перестает отвечать на запросы, у вас есть несколько вариантов действий по восстановлению его работы, например запуск тестов для выявления места ошибки, перезапуск демона, если известно, что он может служить причиной остановки сервиса, или перезагрузка сервера, если все остальное не помогло. Наличие стандартных процессов для обработки распространенных сценариев сбоя значительно сократит время, в течение которого ваша система является недоступной. Кроме того, с их помощью можно получить ценную диагностическую информацию о возникшей ошибке, чтобы команда разработки смогла найти причину ее возникновения и предупредить появление таких ошибок в будущем.

Если предупреждение было отправлено сервисом, первыми его должны получить владельцы сервиса — в конце концов, это они ответственны за решение любых проблем с ним. Но и другим командам, тесно связанным с поврежденным сервисом и заинтересованным в его работе, не помешает информация о возникших проблемах. Например, членам команды, использующей конкретный сервис, нужно знать об отказах его работы, чтобы они могли принять меры к поддержанию своей системы в активном состоянии, пока сервис, от которого они зависят, не функционирует.

Все эти процессы и процедуры должны быть включены в руководство по обслуживанию системы, доступное всем членам команды,

работающим в качестве оперативной поддержки. В него следует включить контактные данные лиц, ответственных за работу систем и сервисов, от которых зависит ваше приложение, а также тех, к кому следует обратиться, если решить проблему простыми средствами не удалось.

Все эти процессы, процедуры и руководства должны быть подготовлены заблаговременно, чтобы в случае возникновения сбоя дежурный персонал точно знал, что предпринять для быстрого восстановления работоспособности системы в различных обстоятельствах. Особенно полезны эти процессы и процедуры потому, что сбои имеют обыкновение происходить в самое неподходящее время, например посреди ночи или в выходные — в это время дежурные, вполне возможно, чувствуют себя не слишком бодрыми. Эти рекомендации помогут вашей команде принять разумные и безопасные решения по восстановлению работоспособности системы.

Будьте подготовлены

Никто не может предсказать, когда и где доступность даст сбой. Но вы смело можете исходить из того, что это точно случится, особенно если ваша система растет, наблюдается увеличение числа пользовательских запросов и возрастание сложности приложений. Быть готовыми к возникновению сбоев доступности — лучший способ снизить вероятность и серьезность этих проблем. Пять техник, описанных в этой главе, образуют четкую стратегию по поддержанию высокого уровня доступности вашего приложения.

Copyrighted material

3 Измерение доступности

Измерять доступность очень важно для поддержания высокого уровня доступности вашей системы. Только делая это, вы можете судить о работе своего приложения в настоящий момент и о том, как его доступность меняется с течением времени.

Самый распространенный механизм измерения доступности веб-приложения — это вычисление процента времени, в течение которого оно доступно для пользователей. Мы можем выразить это следующей формулой для какого-либо периода:

Степень доступности сайта =
= (Сумма секунд в промежутке времени –
– Сумма секунд недоступности системы) /
/ Сумма секунд в промежутке времени.

Рассмотрим пример. Допустим, в течение апреля ваш сайт не работал два раза: в первый раз период недоступности составил 37 мин, а во второй — 15 мин. Какова степень доступности вашего сайта?

Пример 3.1. Степень доступности

Сумма секунд недоступности системы = (37 + 15) × 60 = 3120 с.

Сумма секунд в месяце = 30 дней × 86 400 с/сут. = 2 592 000 с.

Copyrighted material

Степень доступности сайта = (Сумма секунд
в промежутке времени – Сумма секунд недоступности системы) /
/ Сумма секунд в промежутке времени.

Степень доступности сайта =
= (2 592 000 – 3120) / 2 592 000 = 0,998795.

Степень доступности вашего сайта равна 99,8795 %.

Как вы можете видеть из этого примера, даже небольшой период неработоспособности может негативно повлиять на степень доступности сайта.

Девятки

Доступность часто указывают девятками. Это сокращенный способ обозначения степеней доступности. В табл. 3.1 объясняется его суть.

Таблица 3.1. Девятки

Девятки	Процент, %	Период неработоспособности за месяц*
2 девятки	99,00	432 мин
3 девятки	99,9	43 мин
4 девятки	99,99	4 мин
5 девяток	99,999	26 с
6 девяток	99,9999	2,6 с

* За основу взят 30-дневный месяц, продолжительность которого 43 200 мин.

В примере 3.1 мы видим, что время неработоспособности нашего сайта близко к значению 3 девятки (99,8795 и 99,9 %), а для сайта, который претендует на оценку доступности 5 девяток, допускается только *26 с* неработоспособности *в месяц*!

Что считать разумной доступностью

Какова же разумная степень доступности, позволяющая считать, что уровень доступности вашей системы достаточно высок?

Невозможно коротко ответить на этот вопрос. Этот показатель сильно зависит от специфики вашего сайта, ожиданий пользователей, нужд бизнеса и бизнес-целей. Вы сами должны определить, какой показатель подойдет для вашего бизнеса.

Для большинства веб-приложений *приемлемой степенью доступности* считается 3 девятки. Согласно табл. 3.1, в этом случае допустимы 43 мин неработоспособности в месяц. Чтобы веб-приложение считалось высокодоступным, ставится планка в 5 девяток, что допускает лишь 26 с неработоспособности в месяц.

Не обманывайте себя

Не обманывайте себя, считая, что ваш сайт обладает высокой степенью доступности, когда это не так. Плановое и регулярное обслуживание, во время которого ваше приложение недоступно, также учитывается при оценке степени доступности.

Я часто слышу: «Наше приложение никогда не отказывает, потому что мы регулярно производим обслуживание системы. Мы планируем еженедельные двухчасовые промежутки на обслуживание и выполняем работы в это время, поддерживая таким образом доступность на высоком уровне».

Так ли высока доступность приложения этой компании? Давайте разберемся.

Пример 3.2. Пример доступности приложения с окнами для обслуживания

Степень доступности сайта = (Сумма секунд в промежутке времени – Сумма секунд недоступности системы) / / Сумма секунд в промежутке времени.

Часов в неделю = 7 дней · 24 ч = 168 ч.

Часов неработоспособности в неделю = 2 ч.

Доступность сайта (без отказов) = (168 – 2) / 168 = 0,988.

Доступность сайта (без отказов) = 98,8 %.

Copyrighted material

Не имея ни единого сбоя, эта организация может достичь лишь 98,8 % доступности. Это не дотягивает даже до 2 девяток (98,8 против 99 %).

Плановое обслуживание ненамного лучше неожиданного сбоя. Если пользователи ожидают, что приложение будет работать, а это не так, они получают негативный опыт, и совершенно неважно, планировали вы это отключение или нет.

Доступность в цифрах

Измерение доступности очень важно для поддержания высокого уровня этого показателя в вашей системе как в настоящий момент, так и в будущем. В этой главе был рассмотрены общий механизм измерения доступности и дан несколько рекомендаций по поводу того, что считать разумной степенью доступности.

Copyrighted material

4 Улучшение неудовлетворительной доступности

Ваше приложение успешно работает в сети. Все системы в порядке, а команда действует эффективно. Похоже, все идет хорошо. Трафик стабильно растет, а вместе с ним и продажи. Все очень довольны.

Затем что-то происходит, и неожиданно ваша система дает небольшой сбой. Это не катастрофа: прежде ваша доступность была на превосходном уровне, и небольшой перебой не слишком сильно на нее влияет. Трафик продолжает расти. Никто не принимает его во внимание, все думают: ну, такие вещи случаются со всеми.

Потом это происходит снова. Упс... Ну ладно, пока катастрофы все еще нет. Не время паниковать, это все еще досадная проблема, которая случается со всеми.

Еще один сбой...

Ваш СЕО-отдел напрягается. Клиенты начинают жаловаться и спрашивать, что происходит. В отделе продаж нарастает беспокойство.

И еще один...

Вскоре ваша система, когда-то стабильная и работоспособная, становится все менее и менее надежной, сбои притягивают все больше внимания.

Copyrighted material

И сейчас у вас уже проблемы.

Что же произошло? Восстановление доступности системы — задача не из легких. Что предпринять, если доступность системы нарушилась? Что делать, если ее уровень упал или готов упасть до неприемлемого значения и вам нужно срочно исправить ситуацию, пока клиенты не разбежались?

Водоворот проблем может вас затянуть, и только знание того, что нужно делать, если уровень доступности приложения начал падать, поможет вам избежать этого. В следующих разделах описываются шаги, которые вы можете предпринять в случае обнаружения проблем с доступностью.

Измеряйте и отслеживайте текущий уровень доступности

Чтобы понять, что происходит с доступностью вашего приложения, для начала нужно измерить ее текущий уровень. Измерение периодов времени, когда приложение работает и когда оно вышло из строя, поможет вычислить *степень доступности*, которая, в свою очередь, покажет состояние дел за определенный период. Вы можете использовать эту величину и для того, чтобы определить, снижается доступность или растет.

Нужно непрерывно отслеживать степень доступности и фиксировать результаты. Кроме этого, отмечайте ключевые изменения в вашем приложении, такие как выполнение системных улучшений и перестроений. Таким образом вы сможете заметить определенную корреляцию между событиями в системе и проблемами с доступностью. Это поможет вам вычислить риски для доступности.

Обратитесь к главе 3, чтобы освежить в памяти инструкции по измерению доступности.

Затем нужно составить прогноз поведения вашего приложения с точки зрения доступности. Существует инструмент, который поможет

вам управлять доступностью приложения, — метод *сервисных классов*. По сути, класс сервиса — это просто метка, связанная с определенным сервисом и обозначающая, насколько критичен данный сервис для функционирования вашего бизнеса. Это позволяет вам и вашим командам отделить критически важные сервисы от тех, что важны, но не жизненно необходимы. Мы обсудим теорию сервисных классов более подробно в главе 16.

И наконец, составьте и обеспечьте постоянную поддержку *матрицы рисков*. Имея этот инструмент, вы сможете добиться прозрачности своих технических долгов и связанных с ними рисков в своем приложении. Матрицы риска подробно рассматриваются в главе 7, а риски в целом обсуждаются в главах 5 и 6.

Теперь, когда у вас есть способ отслеживания доступности приложения, а также метод выявления рисков и управления ими, разумно будет регулярно пересматривать планы по управлению рисками. Кроме того, необходимо разработать и внедрить планы по смягчению последствий рисков, чтобы снизить возможный вред для приложения. В результате получится набор конкретных задач для вас и вашей команды разработки, в результате чего будут проработаны самые рискованные области в приложении. Этот процесс детально обсуждается в главе 8.

Автоматизируйте ручные процессы

Для поддержки доступности на высоком уровне вы должны исключить любые неопределенности. Выполнение некоторых операций вручную может привести к непредсказуемым результатам, что и происходит на практике довольно часто.

Ни в коем случае нельзя выполнять вручную никакие операции в системе, находящейся в эксплуатации.

Когда вы вносите в систему какие-то изменения, они могут улучшить ее или, напротив, поставить под угрозу. Выполнение только регулярных задач дает вам следующие преимущества.

- *Возможность протестировать задачу перед внедрением*. Тестирование результатов работы конкретного изменения чрезвычай-

Copyrighted material

но важно для предотвращения сбоев, которые ведут к выходу системы из строя.

- *Возможность внесения в задание мелких правок и доведение его до точного соответствия задуманному*. Иными словами, вы можете несколько раз уточнить и выверить изменения, которые собираетесь внести в систему, перед тем как сделаете это.
- *Возможность проверки выполненной задачи третьей стороной*. Это повышает вероятность того, что реализация задачи приведет к непредсказуемым результатам.
- *Возможность регистрации задания в системе контроля версий*. Системы контроля версий позволяют вам определить, когда было сделано какое-либо изменение, кем и по какой причине.
- *Возможность выполнить одну и ту же задачу для нескольких ресурсов одновременно*. Внести улучшения в работу определенного сервера — это здорово, но возможность согласованно применить это улучшение ко всем нужным серверам принесет намного больше пользы.
- *Возможность организации согласованной работы всех имеющихся ресурсов*. Если вы систематически вносите изменения в различные ресурсы (например, серверы) по отдельности, их работа мало-помалу рассинхронизируется и они начнут действовать по-разному. После этого выявить, на каком сервере возникла проблема, станет очень трудно, так как у вас не будет общей линии ожидаемого поведения, которую вы могли бы использовать для сравнения.
- *Возможность внедрить повторяющиеся задачи*. Повторяющиеся задачи поддаются учету и контролю, следовательно, позднее вы можете проанализировать их влияние, позитивное или негативное, на систему в целом.

Существует много систем, где вообще никто не имеет доступа к производственной среде. Вообще никто. Взаимодействие с этой средой осуществляется через автоматические процессы и процедуры. Владельцы этих систем надежно оградили доступ к ним именно по причинам, изложенным ранее.

В заключение я хотел бы сказать, что, если вы не можете повторно выполнить задачу, ее полезность сомнительна. Во многих случаях возможность повторения изменений значительно повлияет на стабильность ваших системы и приложения. К таким задачам я отношу изменения в конфигурации, регулировку и настройку производительности, перезапуск серверов, процессов и задач, изменение правил маршрутизации, а также обновление и развертывание пакетов программного обеспечения.

Автоматическое развертывание

Если вы автоматизировали развертывание, у вас есть гарантия, что во всей системе применены одинаковые изменения и вы можете изменять что угодно с предсказуемым результатом. Кроме того, откаты к заведомо корректным состояниям становятся более надежными при наличии автоматических систем развертывания.

Управление конфигурацией

Вместо того чтобы «подправлять переменные конфигурации» в ядре сервера, настраивайте четкие процессы автоматического применения изменений. К примеру, можно написать скрипт, который внесет изменения, а затем опробовать его в системе управления изменениями в системе. Таким образом вы сможете равномерно применить идентичные изменения во всей системе. Кроме того, когда придется добавлять в систему новые серверы или замещать старые, наличие готовой к применению конфигурации повысит вероятность того, что вы сможете выполнить это действие безопасно и с минимальным влиянием на систему. Такие инструменты, как Puppet или Chef, могут упростить организацию этого процесса.

Это относится ко всем компонентам инфраструктуры системы, а не только к серверам. Работать по такому принципу можно со свитчами, роутерами, сетевыми компонентами, а также системами мониторинга приложений и систем.

Чтобы извлечь из управления конфигурацией максимальную пользу, нужно пользоваться им *всегда* и для *всех* изменений в системе.

Copyrighted material

Ни в коем случае нельзя обходить систему управления конфигурацией, чтобы внести какие-либо изменения. Никогда и ни при каких обстоятельствах.

Не волнуйтесь, я уже все пофиксил

Вы не поверите, когда я скажу, сколько раз получал сообщение об оперативном обновлении, где говорилось что-то вроде: «С одним из наших серверов ночью произошел сбой. Количество открытых файлов, которые может обработать сервер, достигло максимума, поэтому я подправил системную переменную, увеличил допустимое количество открытых файлов, так что сервер сейчас снова работает».

Ну да, все будет работать, пока кто-нибудь случайно не перепишет это изменение, так как оно нигде не задокументировано. Или, например, какой-нибудь другой сервер, на котором запущено приложение, не выйдет из строя точно таким же образом, потому что к нему это изменение не было применено.

Согласованность действий, способность к воспроизведению и неусыпное внимание к деталям критически важны для работающего процесса управления конфигурацией. А стандартизированный процесс управления конфигурацией с высокой точностью повторяемости, который мы рассмотрели здесь, критически важен для поддержания высокого уровня доступности вашей системы.

Высокая периодичность изменений и эксперименты с ними

Еще одно преимущество наличия строго автоматизированного процесса внесения изменений и обновлений в систему, имеющего высокую точность повторяемости, — возможность экспериментировать с изменениями. Допустим, вы хотите внести в конфигурацию серверов какие-то изменения, которые, по вашему мнению, улучшат производительность системы, — то же увеличение максимального количества открытых файлов, упомянутое в примере «Не волнуйтесь, я уже все пофиксил». При наличии автоматизированного процесса управления изменениями вы можете проделать следующее:

Copyrighted material

- задокументировать предлагаемое изменение;
- согласовать изменения с людьми, которые компетентны в этой области и могут внести какие-то правки или дать полезные рекомендации;
- проверить изменения на серверах тестовой или стейджинговой среды;
- быстро и просто развернуть изменения;
- немедленно проверить результаты изменений. В том случае если они окажутся неудовлетворительными, можно будет быстро откатиться к предыдущему заведомо хорошему состоянию.

Суть внедрения этого процесса — наличие автоматизированного процесса изменений с возможностью отката, а также возможность регулярно и без труда вносить в систему мелкие изменения[1]. Первое обеспечивает согласованность изменений, а второе позволяет экспериментировать, отменяя изменения с неудачным результатом практически без негативных последствий для клиентов.

Автоматизированное тестирование корректности изменений

Имея автоматические процессы внедрения изменений и развертывания[2], вы можете реализовать также автоматическое выполнение тестов корректности для всех изменений. Можете использовать браузерное приложение для тестирования или что-нибудь вроде New Relic Synthetics для эмуляции пользовательского взаимодействия со своим приложением.

---

[1] Согласно Вернеру Воглсу, CTO в Amazon, в 2014 г. Amazon произвел 50 млн развертываний на индивидуальных хостингах (примерно каждую секунду в году).

[2] Это может быть (но не обязательно) современный процесс непрерывной интеграции и непрерывного развертывания (Continuous Integration and Continuous Deploy, CI/CD).

Собираясь внести какое-нибудь изменение в производственной среде, вы можете с помощью автоматической системы развертывания сперва внедрить изменения в тестовой или стейджинговой среде. После этого там можно запустить автоматические тесты и убедиться, что изменения не нарушают правильную работу приложения.

Если (или когда) эти тесты будут пройдены успешно, вы автоматически и согласованно разворачиваете изменения в производственной среде. В зависимости от того, как устроены ваши тесты, нужно обеспечить возможность регулярного запуска этих тестов и там. Таким образом вы можете убедиться в том, что изменения не вызвали в производственной среде никаких сбоев.

Добившись полной автоматизации этого процесса, вы сможете получить уверенность в том, что изменения не окажут негативного воздействия на ваши продуктовые системы.

Совершенствуйте свои системы

Теперь, когда у вас есть система мониторинга доступности, способ отслеживания рисков и смягчения их последствий для системы, а также методы простого, безопасного и согласованного внесения изменений в систему, можно сконцентрировать усилия на улучшении доступности вашего приложения самой по себе.

Регулярно пересматривайте свои матрицу рисков (она упоминалась ранее, а также будет обсуждаться в главе 7) и планы восстановления. Возьмите за правило пересматривать их при разборе полетов после каждого сбоя. Реализуйте запланированные изменения по смягчению последствий рисков, описанные в матрице, причем внедрите их автоматизированным путем с обеспечением безопасности и выполнением тестов на корректность, которые мы обсудили ранее. Убедитесь, что принятые меры и в самом деле улучшили степень доступности. Продолжайте этот процесс до тех пор, пока уровень доступности не достигнет желаемого и необходимого.

Из главы 13 вы узнаете, как восстановить работу системы после выхода из строя сервисов.

Copyrighted material

Доведите метрики доступности до сведения каждого, кто связан с управлением системой. Подобная прозрачность сослужит вам добрую службу при внедрении такого рода проектов для улучшения доступности системы.

Рост и перемены в вашем приложении

По мере того как система растет, вам приходится обрабатывать все больший и больший объем трафика и запросов. Рост потребностей может повлечь за собой снижение доступности. В части IV подробно описано масштабирование приложений, и многие из тем, раскрываемых здесь, могут помочь в решении проблем с доступностью в растущем приложении. В частности, обработка ошибок и сбоев, вызванных ростом, обсуждается в главе 14. Организация соглашений сервисного уровня (Service-Level Agreement, SLA) рассматривается в главе 18. Метод классов сервисов, который вы можете использовать для выявления сервисов, наиболее сильно влияющих на доступность, освещается в главах 16 и 17.

Рекомендую также внедрить такой инструмент тестирования, как День большой игры, в результате чего вы узнаете, как ваше приложение ведет себя при различных видах сбоев. Это обсуждается в главе 9.

Удерживайте доступность на высоком уровне

Скорее всего, приложение будет постоянно изменяться. Следовательно, ваши планы по управлению рисками, смягчению последствий рисков, восстановлению после сбоев и последовательность действий в чрезвычайных ситуациях нуждаются в регулярном пересмотре.

Зная, что нужно предпринять, когда доступность приложения ухудшается, вы никогда не окажетесь погребены под гнетом бесконечных проблем. Идеи, описанные в этой главе, помогут вам правильно организовать работу команды над приложением, своевременно реагируя на проблемы и поддерживая доступность на высоком уровне.

Copyrighted material

Часть II
Управление рисками

Вам не удастся управлять рисками в своей системе, если вы о них не знаете.

...Но существуют также неизвестные неизвестные — то, о чем мы не знаем, что мы этого не знаем. Если посмотреть на историю нашей страны и других свободных стран, эта последняя категория относится к наиболее сложным.

Дональд Рамсфелд

Copyrighted material

5 Что такое управление рисками

Риски есть во всех сложных системах, это неотъемлемая часть любой из них. Ни в одной сложной системе, в том числе в веб-приложении, нельзя избавиться от всех рисков. Однако анализ рисков и определение того, насколько данный риск приемлем, очень важны для поддержания благополучия системы.

В этой главе мы поговорим о том, что такое риск и как его определить. Все это относится к процессу, который называется *управлением рисками*. Именно он поможет уменьшить количество рисков для ваших приложений.

А сейчас рассмотрим пример 5.1, вернувшись в День большой игры из главы 1.

Пример 5.1. Управление рисками в важной игре

Коротко вспомним суть примера с важным матчем, который мы рассматривали ранее. Сегодня воскресенье, день важной игры. Вы пригласили друзей, чтобы вместе посмотреть ее по своему новому телевизору. Игра вот-вот начнется. И вдруг... гаснет свет, экран телевизора выключается. Для вас и ваших друзей игра окончена. А вежливый сотрудник энергоснабжающей компании, отвечая на ваш возмущенный вопрос, говорит, что он очень сожалеет, но компания гарантирует только 95%-ный уровень доступности электрической сети.

Copyrighted material

В этом примере компания энергоснабжения принимает риски, один из которых состоит в том, что подача электроэнергии может прекратиться во время важного матча.

Они даже подсчитали вероятность того (95 %), что все будет работать.

Компания знает, какие происшествия могут вызвать прекращение энергоснабжения — скажем, авария на линии электропередачи. Для нивелирования этих рисков они могут предпринять какие-то меры, например:

- проложат кабельные линии (для защиты от непогоды);
- укрепят опоры воздушных линий электропередачи (для снижения вероятности того, что их опрокинет ветер);
- обеспечат резервные электрические системы, которые будут работать вместо вышедших из строя.

Но все эти стратегии стоят денег. Разумно ли вкладывать в укрепление линий электропередачи? Оправданны ли траты на прокладку кабеля? Сопоставима ли стоимость последствий риска с затратами на его предотвращение? Все эти вопросы относятся к управлению рисками, и мы подробно рассмотрим их в этой главе.

Управление рисками

Управление рисками включает в себя определение того, где именно в вашей системе существуют риски, решение о том, какие риски непременно нужно удалить, а какие можно оставить, а затем принятие мер по смягчению последствий оставшихся рисков, призванное снизить вероятность их наступления и серьезность последствий.

Когда риск *запускается* (или *случается*), у вас или в вашей системе наблюдаются потери. Могут быть потеряны, скажем, данные вашей компании или от вас уйдет клиент. Ваше приложение может стать недоступным для клиентов. Как потери можно рассматривать также некорректные или ошибочные результаты. В любом случае пострадает доверие ваших клиентов к вам и вашей способности заботиться об их данных и их бизнесе, что в конечном итоге будет стоить вам денег.

Copyrighted material

И все же стоит рассмотреть денежные потери с противоположной точки зрения: а какова стоимость нивелирования риска и предотвращения его последствий?

Таким образом, управление рисками сводится к балансированию между стоимостью нивелирования риска и стоимостью наступления его последствий.

Выявление рисков

Первый шаг в управлении рисками — создание списка известных рисков с указанием их критичности и вероятности наступления. Назовем его *матрицей рисков*. Пример матрицы показан на рис. 5.1.

Первым шагом в создании такой матрицы может быть мозговой штурм. Вы можете почерпнуть идеи о том, что следует в нее включить, из следующих источников:

- коллективного разума разработчиков;
- наиболее проблемных, по вашим сведениям, областей приложения;
- известных вам направлений угроз и уязвимостей;
- известных вам недоработанных областей или областей с недостающими возможностями;
- известных вам областей с плохой производительностью;
- известных вам закономерностей в трафике и резких отклонений от них;
- специфических проблем, о которых сообщают владельцы бизнеса, сотрудники технической поддержки или пользователи;
- известных вам технических долгов в системе.

Вероятно, вы думаете сейчас, что известные вам вещи в этом списке — это хорошо, но там должно быть и что-то неожиданное. Это правильно. Ваша цель — осветить как можно больше уязвимостей, и если ничто из найденного вас не удивляет, скорее всего, вы недостаточно глубоко копали.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Risk ID	System	Owner	Description	Date Identifie	Likelihoo	Sever	Mitigation Plan	Statu	ETA	Monitoring	Triggered Plan(what to do if risk occurs?)	Comments
2	1	FrontEnd	FE Dev Team	Requires User Identity service to function. Front end fails if service is down.	10/13/15	Low	High	Perhaps we can cache the date for a period of time?	Open	5/26/16	Yes	If it happens often, we may have to look into improving independence on UI service.	
3													
4													
5													
6													
7													
8													
9													
10													
11													
12													
13													
14													
15													
16													
17													
18													
19													
20													

Рис. 5.1. Пример матрицы рисков

Создание матрицы рисков включает в себя расстановку приоритетов по степени вероятности наступления риска и тяжести его последствий (критичности).

Мы обсудим этот список подробно в главе 7.

Начните с самого страшного

Составив первую версию списка, пройдитесь по нему и отметьте самые опасные риски. Как выбрать самые опасные? Ищите те, которые случаются чаще других, или те, которые, может быть, еще не случались, но последствия наступления которых для системы могут быть очень серьезными. Самые опасные риски — те, у которых высока вероятность наступления и самые тяжелые последствия для системы. В главе 6 обсуждаются разница между критичностью и вероятностью, а также использование этой информации для поддержки управления рисками, в том числе обнаружения самых опасных рисков в системе.

На рис. 5.1 приведен пример риска, который может быть одним из самых опасных в системе: Frontend fails if user identity service is down (Пользовательский интерфейс не работает, если сервис вышел из строя).

Обнаружив несколько самых опасных рисков, внесите их в свои планы работ и убедитесь, что они попадут к разработчикам в самое ближайшее время.

Смягчите последствия

Для всех рисков независимо от их тяжести проведите мозговой штурм для выяснения имеющихся возможностей снижения частоты или вероятности их наступления либо снижения серьезности их последствий. Эти возможности можно назвать *средствами смягчения рисков*.

Эти средства могут быть очень полезными. Особенно интересны для вас будут те из них, которые могут снизить серьезность риска, уменьшив его вероятность, или критичность, или и то и другое, но при этом просты в реализации и недороги.

Copyrighted material

Давайте обсудим риск «Пользовательский интерфейс не работает, если сервис вышел из строя». Возможное средство смягчения этого риска — кэширование информации о пользователе, в результате чего эта информация будет доступна, даже если сервис пользовательской идентификации недоступен.

Можете сконцентрироваться на самых опасных рисках, находя способы уменьшить их критичность, но не забывайте и о тех, которыми вряд ли займетесь в ближайшее время. Обнаружение средств, которые могут снизить критичность или вероятность наступления этих рисков, почти так же полезно, как и непосредственное их исправление.

Регулярно пересматривайте матрицу

Если не будете регулярно пересматривать свою матрицу, она очень скоро устареет. Необходимо проверять актуальность матрицы всей командой не реже чем раз в квартал, а для очень активных систем с высокой критичностью — раз в месяц. Кроме того, пересматривайте матрицу после каждого инцидента, проверяя, был ли этот инцидент отражен как известный риск?

Пересматривая матрицу, первым делом подумайте о новых рисках, которые были обнаружены недавно. Внесите их в матрицу. Кроме того, удалите старые записи, которые больше не являются рисками.

После этого проверьте, нет ли изменений в критичности или вероятности наступления рисков. Зачастую реализованные средства смягчения последствий рисков снизят либо их критичность, либо вероятность наступления. Или, напротив, открылись новые сведения об этом риске, которые позволяют увеличить какой-либо из этих параметров. Чаще всего так происходит, если с момента вашего последнего пересмотра этот риск реализовался; надо думать, если риск, обозначенный как маловероятный, все же случился, величину вероятности следует пересмотреть. Проверьте также, есть ли риски, от которых вы можете избавиться (то есть исправить недостатки в приложении, которые служат их причиной), внеся работу над ними в текущий план.

И наконец, поищите новые или обновленные средства смягчения последствий рисков, которые вы можете ввести в дело.

Copyrighted material

Принципы управления рисками

Как вы управляете рисками в своей системе? Есть базовые правила, на которые следует ориентироваться.

- *Выявление рисков*. Начните с составления списка известных рисков в системе. Такой список называется матрицей рисков. Отсортируйте его по приоритетам.
- *Ликвидация самых опасных рисков*. Найдите самые опасные риски в списке и запланируйте работу по их ликвидации.
- *Смягчение последствий рисков*. Составьте план по снижению вероятности наступления и смягчению последствий опасных рисков, которые вы не можете ликвидировать.
- *Регулярный пересмотр плана*. Регулярно пересматривайте матрицу рисков.

Copyrighted material

6 Критичность и вероятность

Очень важно понять соотношение между критичностью и вероятностью. Управление рисками, в частности, требует понимания того, когда вам нужно беспокоиться о критичности риска, а не о вероятности его наступления и наоборот. Понимание разницы между этими понятиями необходимо для анализа серьезности рисков в системе.

Мы рассматриваем риски как комбинацию двух компонентов.

- *Критичность*. Стоимость последствий риска в случае его наступления (например, чего нам будет стоить отключение электроэнергии у клиентов?).
- *Вероятность*. Вероятность того, что риск наступит (например, какова вероятность урагана?).

Управление рисками состоит из управления этими двумя величинами. Вы можете снизить вероятность наступления риска или серьезность последствий этого. Не обязательно обеспечивать и то и другое для каждого конкретного риска, но стоит учитывать обе величины, чтобы видеть перспективу управления рисками в целом.

Значительность риска определяется сочетанием вероятности наступления риска и серьезности последствий этого. Чтобы успешно управлять рисками, нужно учитывать обе эти величины, а также соотношение между ними. Чтобы снизить серьезность какого-либо конкретного риска, надо снизить хотя бы одну из этих величин.

Copyrighted material

Для того чтобы понять разницу, лучше всего рассмотреть несколько примеров разных рисков и проанализировать, как различаются между собой их вероятность и критичность. В дальнейшем в этой главе мы рассмотрим следующий пример.

Вернемся в интернет-магазин футболок — типичный представитель розничной интернет-торговли. В нем имеется каталог доступных футболок, индивидуальные страницы для каждой футболки, где можно рассмотреть ее в деталях на фотографиях, а также система обработки заказов, которой пользуются клиенты для заказа и оплаты футболок, которые они желают получить.

Давайте рассмотрим, какие риски угрожают этому магазину.

Список топ-10: низкая вероятность и низкая критичность риска

Итак, предположим, что в нашем интернет-магазине есть функциональность, выводящая в правом верхнем углу страницы список десяти самых популярных футболок. Посетители сайта видят этот список, нажимают на понравившуюся футболку и тут же могут ее заказать.

Спросим себя: что будет, если список топ-10 по какой-либо причине (чаще всего из-за отказа сервиса) не может быть загружен? Если нельзя отобразить этот список, допустим, мы показываем статический список каких-то футболок, не обязательно самых популярных на настоящий момент. Этот сервис, скорее всего, не будет отказывать часто, поскольку сгенерировать список десяти самых популярных футболок несложно.

Как оценить риск этой функциональности для нашего магазина?

Давайте проанализируем его.

- *Вероятность* риска низка, так как сервис, обеспечивающий отображение списка, достаточно надежен (допустим, что список сгенерировать несложно).
- Но если список не отобразится, насколько серьезны последствия? Как я уже говорил, если нельзя отобразить топ-10, мы просто показываем статичный список. Не идеально, но все же влияние

Copyrighted material

на наших покупателей невелико, да и бизнес особого ущерба не понесет. Поэтому можно считать, что *критичность* этого риска также низка.

Риски наподобие этого смело можно игнорировать и не возвращаться к ним в дальнейшем, так как вероятность таких событий очень низка, да и негативные последствия невелики.

База данных с заказами: низкая вероятность и высокая критичность риска

Продолжая рассматривать пример с футболками, предположим, что сведения о наших заказах хранятся в типичной базе данных. Как только покупатель разместил заказ, в базе данных создается соответствующая запись. По мере того как вы обрабатываете заказы, получаете платежи и отправляете футболки покупателям, данные в базе обновляются. Позднее они используются для генерации финансовых отчетов, которые вы можете применять для анализа бизнеса, планирования или представления в налоговую инспекцию.

Хорошо понимая важность базы данных, вы размещаете ее на высококачественном оборудовании с репликационными компонентами (например, с избыточным массивом независимых дисков — RAID). Кроме того, регулярно выполняете резервное копирование данных. Однако база данных все еще остается компонентом, сбой которого выводит из строя всю систему: в ней содержится значительное количество критических для бизнеса данных, а ваш сайт не может функционировать (заказы невозможно разместить), если база данных недоступна. Потеря базы данных — потеря потерь.

Как описать риск для вашего магазина, связанный с отказом базы данных, где хранятся сведения о заказах?

Давайте рассмотрим его подробнее.

- *Вероятность* наступления риска довольно низка, так как вы используете высококачественное оборудование с репликационными компонентами. База данных надежна.

- А вот *критичность* последствий наступления риска будет очень высока. Если база данных вышла из строя, то не работает вся система обработки заказов, кроме того, вы рискуете потерять важные для бизнеса данные.

Таким образом, можно сказать, что это Н/В-риск — риск с низкой вероятностью наступления, но высокой критичностью. Такого рода риски легко пропустить, так как они и правда реализуются довольно редко (низкая вероятность). Однако если их проигнорировать, это может дорого обойтись: цена сбоя очень высока.

Поскольку критичность высока, стоит поискать возможности для смягчения последствий рисков. Например, можно всегда иметь под рукой готовую к использованию резервную копию базы, на которую немедленно переключиться в случае выхода из строя основной. В результате вы быстро сможете наладить работу без потери времени или значительной части данных. Или, скажем, можно обратиться к технологиям баз данных, которые распределяют данные между несколькими серверами. Так приложение будет функционировать, даже если какой-либо из серверов не работает.

Использование одного из этих методов может позволить перевести данный риск из категории Н/В в категорию Н/С (низкая вероятность, средняя критичность) или даже Н/Н (низкая вероятность, низкая критичность).

Такого рода средства смягчения последствий рисков, позволяющие значительно снизить серьезность проблемы, дополнительно обсуждаются в главе 8.

Специфические шрифты: высокая вероятность и низкая критичность риска

Продолжая разговор об интернет-магазине футболок, предположим, что вы решили придать сайту стильности за счет использования специфических шрифтов для текстовых строк и описаний.

На сервисе, который предоставляет различные шрифты, размещенные на его собственном хостинге, вы обнаружили замечательный подходящий вашей задаче шрифт. Для того чтобы отобразить шрифт, браузер вашего пользователя скачивает его непосредственно с этого стороннего сервис-провайдера. Если же этот специфический шрифт недоступен, то используется стандартный системный шрифт и страница выглядит так, как она выглядела до внесения каких-либо изменений.

К сожалению, вы обнаруживаете, что провайдер сервиса шрифтов довольно часто не работает — чаще, чем вам хотелось бы. А когда он не работает, клиенты не видят стильного, придирчиво выбранного вами шрифта. И данная проблема встречается достаточно часто.

Каков же риск для вашего магазина при использовании дополнительных шрифтов? Рассмотрим его подробнее.

- *Вероятность* того, что шрифт не отобразится, высока, потому что сервис-провайдер работает нестабильно и часто выходит из строя.
- Но даже если это случилось, сайт продолжает работать — он просто выглядит не так стильно, как вам хотелось бы. *Критичность* проблемы невелика. Стильность сайта страдает, но все же он полностью функционирует без особых проблем.

Таким образом, это В/Н-риск — высокая *вероятность* наступления риска, но низкая критичность его последствий.

Средства смягчения последствий этого риска включают в себя снижение вероятности его наступления. Вы можете достичь этого, поработав со сторонним провайдером и увеличив доступность данного сервиса. Или можно приготовить список резервных провайдеров, которые предоставляют эти же или похожие шрифты, и переключаться на них, если основной сервис выходит из строя. Так можно снизить *вероятность* возникновения данной проблемы.

А вот для снижения критичности проблемы мало что можно сделать, учитывая, что она и так невысока.

Copyrighted material

Фотографии футболок: высокая вероятность и высокая критичность риска

Продолжим говорить об интернет-магазине и обратим внимание на фотографии футболок, появляющиеся на сайте. Это крайне важная часть магазина, потому что никто не купит футболку, не зная, как она выглядит. Если изображения футболок не появляются, пользователи покинут сайт и вы потеряете покупателей.

Однако сервер, на котором вы размещаете изображения, не очень-то стабилен: он то работает, то не работает и, кроме того, сталкивается с проблемами при чтении изображений с диска. Сервер староват, его давно пора бы заменить. Он часто выходит из строя и требует регулярного перезапуска. Его комплектующие также часто сбоят и требуют замены. Тем не менее вы храните изображения именно на нем.

Каков риск того, что ваш сайт нельзя будет использовать из-за недоступности изображений? Рассмотрим подробнее.

- *Вероятность* того, что изображения не отобразятся, высока, так как сервер ненадежен и часто выходит из строя.
- *Критичность* риска высока, так как, если изображения будут недоступны, клиенты немедленно покинут сайт, ничего не купив.

Таким образом, это В/В-риск, что означает высокую вероятность (оборудование часто выходит из строя) и высокую критичность последствий (все клиенты покидают сайт, ничего не купив).

Такие типы рисков являются самыми опасными. Весьма вероятно, что проблема не только возникнет, но и причинит серьезный ущерб вашему бизнесу. Именно таким рискам вы должны уделить максимум внимания.

Этот пример может показаться банальным, но, скорее всего, рисков В/В-типа в ваших приложениях обнаружится немало. Впрочем, приглядевшись к своему приложению более внимательно, вы, возможно, измените мнение о таких рисках. Вот почему управление рисками так важно.

Copyrighted material

7 Матрица рисков

Первый шаг в управлении рисками — понять, что риски в вашей системе уже есть. Суть матрицы рисков составляют выявление, классификация и приоритизация известных рисков.

Матрица рисков, впервые упомянутая в главе 5, — основной аспект управления рисками в вашей системе. Она представляет собой таблицу, которая содержит обзор актуального состояния всех известных рисков системы.

На рис. 7.1 представлен пример матрицы рисков.

Каждая строка матрицы представляет собой один измеряемый риск, присутствующий в вашей системе. В графах таблицы указаны различные детали этого конкретного риска.

Для каждого риска следует привести следующую информацию.

- *Идентификационный номер риска.* Уникальный идентификационный номер, назначенный данному риску. Он может быть любым, но проще и надежнее всего работать с уникальными идентификаторами, представляющими собой целое число[1].

---

[1] Не стоит использовать в качестве идентификатора номер строки в таблице. Строки матрицы могут быть отсортированы, какие-то из них будут добавляться или удаляться, в результате чего номера рисков окажутся перемешаны. Идентификатор не должен меняться в течение всего жизненного цикла риска.

Copyrighted material

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Risk ID	System	Owner	Description	Date Identifie	Likeliho	Sever	Mitigation Plan	Statu	ETA	Monitor ing	Triggered Plan(what to do if risk occurs?)	Comments
2	1	FrontEnd	FE Dev Team	Requires User Identity service to function. Front end fails if service is down.	10/13/15	Low	High	Perhaps we can cache the date for a period of time?	Open	5/26/16	Yes	If it happens often, we may have to look into improving independence on UI service.	
3													
4													
5													
6													
7													
8													
9													
10													
11													
12													
13													
14													
15													
16													
17													
18													
19													
20													

Рис. 7.1. Пример матрицы рисков

Copyrighted material

- *Система.* Обозначение системы, подсистемы или модуля, в которых присутствует риск. Информация эта зависит от специфики приложения, но в качестве примера можно привести «Пользовательский интерфейс», «Первичная база данных», «Сервис А» и т. п.
- *Владелец.* Имя лица или название команды, которые занимаются областью, где имеется риск, и несут ответственность за планы по его нивелированию или смягчению его последствий.
- *Описание риска.* Описания рисков должны быть довольно краткими, чтобы легко было просканировать их список и быстро понять, о чем речь, но в то же время достаточно полными, чтобы точно описывать суть риска, выделяя самое главное.
- *Дата обнаружения.* Дата, когда риск был выявлен и добавлен в матрицу.
- *Вероятность.* Означает вероятность (низкая, средняя или высокая) реализации риска. Эта величина детально обсуждалась в главе 6. С ее помощью можно отсортировать матрицу и определить, какие риски заслуживают наибольшего внимания или даже требуют немедленных действий[1].
- *Критичность.* Серьезность последствий (низкая, средняя или высокая) реализации риска. Эта величина детально обсуждалась в главе 6. С ее помощью можно отсортировать матрицу и определить, какие риски заслуживают наибольшего внимания или даже требуют немедленных действий.
- *План смягчения последствий риска.* Эта графа должна содержать описание любых средств смягчения последствий риска, которые могут быть использованы или уже используются для снижения критичности или вероятности данного риска.

---

[1] Для упрощения сортировки таблицы по величинам вероятности или критичности можно присвоить им числовые значения, например от 1 (низкая) до 3 (высокая). Удобно использовать запись «1 — низкая», «2 — средняя» и «3 — высокая», а затем задействовать возможности программы, в которой вы составляете матрицу, для ограничения возможных значений величин в графе этими тремя.

- *Статус*. В графе выставляется статус данного риска, например: «активный», «смягчен», «в процессе исправления» или «ликвидирован».
- *Оценка времени*. Здесь указывается предварительная оценка времени, необходимого для окончательного избавления от этого риска (если возможно).
- *Мониторинг*. В этой графе указывается, проводится ли у вас мониторинг реализации этого риска, и если да, то описываются шаги, которые вы предприняли для этого. Если же вы не отслеживаете данный риск, то должны указать, почему, и обозначить дату, когда планируете начать такой мониторинг.
- *План на случай реализации риска*. Если проблема все-таки возникнет, знаете ли вы, что будете предпринимать? В любом случае опишите здесь высокоуровневый план на уровне менеджмента, а не план реагирования на проблему[1].
- *Комментарий*. Укажите здесь любую информацию об этом риске, которая не поместилась или не может быть приведена в его описании. Кроме того, можно указать любые параметры, которые были бы полезны именно в вашей организации, например:
  - *номер отслеживания*. Если у вас есть система регистрации багов или запланированной работы, куда вы заносите элементы, соответствующие рискам в данной матрице, то в этом поле можете указать номер бага или задачи для разработчиков;
  - *история*. Случался ли этот риск ранее? Когда? Как часто?

Объем матрицы рисков

На этой стадии вы, должно быть, задались вопросом: нужно ли создавать общую матрицу для всей компании, или лучше отдельные матрицы для каждой команды или каждого сервиса?

[1] Планы реагирования на проблему должны быть размещены там, где к ним легко могут получить доступ сотрудники службы технической поддержки, — в журнале регистрации инцидентов или в других инструментах.

Это хороший вопрос. Одна матрица для всей компании хороша, если эта компания небольшая, но и в этом случае она может вскоре стать слишком громоздкой. В то же время отдельная матрица на каждый сервис обеспечивает хороший обзор рисков этого сервиса, но затрудняет оценку рисков в масштабе всей компании. Например, становится трудно оценить, какой сервис в компании является самым рискованным.

Я рекомендую составлять отдельную матрицу рисков для каждой команды. Поскольку решения о том, над какими новыми функциональностями или проблемами стоит начать работу, и о выборе приоритета зачастую принимаются на уровне команды, имеет смысл управлять матрицей рисков и приоритизировать ее также на уровне команды. Подробные рекомендации об управлении на уровне команды приводятся в главе 15.

И в заключение: размер матрицы рисков должен соответствовать размеру организации. Следовательно, одна матрица рисков может использоваться для одной команды, группы или организации, которая самостоятельно принимает решения об объеме работы и приоритетах. Вышестоящее руководство может давать высокоуровневые указания, но основная работа по приоритизации выполняется именно на уровне данной группы людей.

Создание матрицы рисков

Удобнее всего начать с одного из шаблонов. Мы создали для вас несколько вариантов в одной из самых популярных программ для работы с шаблонами (http://bit.ly/architectbookdl). Вы, конечно, можете изменять шаблон как пожелаете, но в первой матрице рисков лучше всего придерживаться его, насколько это возможно. Получив некоторый опыт в использовании матрицы и управлении рисками, вы можете изменять шаблон в соответствии со своими нуждами.

В этом шаблоне приведен пример риска как образец использования матрицы (см. рис. 7.1). Разумеется, вы должны удалить его после начала работы.

Copyrighted material

Составление списка: мозговой штурм

Приготовив шаблон, переходите к следующему шагу: проведите мозговой штурм для составления списка рисков, которые должны быть включены в матрицу. Старайтесь вносить в список все риски, которые придут вам на ум, а не только те, которые вас особенно беспокоят. Не анализируйте их в процессе — просто фиксируйте все, о чем подумаете.

Вот какие источники идей будут полезны при мозговом штурме.

- *Команда разработки*. Проведите совещание со своей командой разработки. Как правило, у членов команды хватает поводов для беспокойства об их сервисах. Выслушайте все, что они скажут, и занесите в список.
- *Сотрудники технической поддержки*. Оцените, какой объем работы приходится выполнять команде поддержки. Есть ли области, которые требуют заметно больше внимания с их стороны? Что говорят сотрудники? Есть ли у вас форум технической поддержки? Как правило, области, постоянно требующие внимания сотрудников службы техподдержки, являются богатыми источниками рисков в системе.
- *Векторы атаки*. Подумайте обо всех векторах атаки и уязвимостях в безопасности. Все они независимо от величины и значительности представляют собой риски для ваших сервисов.
- *Бэклог функциональностей*. Просмотрите свой бэклог. Имеются ли в нем еще не реализованные возможности, которые очень важны для здорового состояния системы? Особое внимание уделите элементам бэклога, связанным с мониторингом и обслуживанием.
- *Производительность*. Подумайте о производительности системы. Есть ли известные вам области с низкой производительностью?
- *Владельцы бизнеса*. Поговорите с ними о том, что их беспокоит.
- *Остальная часть команды*. Поговорите с остальной частью команды, включая внутренних пользователей, команды, чья работа связана с вашей, со службой внедрения и т. д. Что беспокоит их?

Copyrighted material

- *Системы и процессы*. Есть ли у вас документированные системы и процессы? Есть ли в вашем приложении области, функционирование которых никак не отражено в документации, или, может быть, она хранится только в головах отдельных людей?
- *Технический долг*. Известны ли вам конкретные технические долги в системе? Примерами технического долга могут быть области кода, которые трудно понять, переусложненные или имеющие больше подвижных частей, чем следует. Области, в которых имеется технический долг, почти гарантированно содержат риски.

Возможно, вы думаете, что в списке окажутся ожидаемые и банальные вещи, но на самом деле обязательно найдется что-то, что вас удивит. Так и должно быть. Нужно выявить как можно больше уязвимостей, и если вы не обнаружили ничего нового для себя, то, скорее всего, не углубились в проблему как следует.

Присвоение критичности и вероятности

А сейчас пройдитесь по списку и заполните параметры критичности и вероятности для каждого риска. Проставляйте значения «Низкая», «Средняя» или «Высокая» или аналогичные им в каждом из этих двух полей.

Еще раз убедитесь, что правильно понимаете концепции вероятности и критичности. Сверяйтесь, если нужно, с главой 6. На этой стадии параметры довольно легко перепутать.

Может быть, стоит сперва пройтись по списку и обозначить вероятность всех рисков, а уже потом взяться за критичность. Помните: весьма вероятно, что у риска могут быть тяжелые последствия, тем не менее он практически никогда не реализуется (или, наоборот, скорее всего реализуется, но это не слишком опасно). В результате у вас будут все комбинации — Н/Н, Н/В, В/Н и В/В. Это нормально, так и должно быть.

Главное на этом этапе — не забывать, что нельзя путать критичность и вероятность, иначе вся работа окажется бесполезной.

Copyrighted material

Еще одна сессия мозгового штурма с командой — лучший способ выполнить эту задачу. Проведите отдельную сессию для классификации рисков, не объединяя ее с той, на которой составляется список. Классификация рисков одновременно с их выявлением — не лучшая идея.

Другая информация о рисках

После этого можно заполнить и другие базовые параметры в таблице: указать систему, владельца, дату выявления, статус.

Не забывайте присваивать каждому риску идентификационный номер (проще всего просто нумеровать их целыми числами, начиная с единицы).

Отслеживаете ли вы этот риск? Пометьте в поле «Мониторинг», есть ли у вас возможность получить предупреждение в случае реализации данного риска.

План смягчения риска

Начните с самых критических рисков и совместно составьте планы смягчения последствий для каждого из них. Затем перейдите к элементам с наибольшей вероятностью.

План смягчения — это план того, какие изменения вы собираетесь внести *сейчас или в ближайшем будущем*, чтобы снизить критичность риска или вероятность его реализации. План смягчения не предусматривает *избавления* от риска, он лишь снижает его критичность или вероятность.

После выполнения шагов, намеченных для смягчения последствий, критичность или вероятность риска снизится, а план станет неактуальным. Тогда в случае необходимости можно подумать о новом плане.

Составлять план смягчения последствий для каждого риска в матрице не обязательно. Тут могут быть риски, которые легче устранить, чем составлять план смягчения последствий. Кроме того, для элементов с низкой вероятностью и низкой критичностью этот план вообще не требуется.

Copyrighted material

План на случай реализации риска

План на случай реализации риска — это действия, которые вы намерены предпринять, *если риск и в самом деле воплотится в жизнь*. Да, это может быть просто указание «Исправить баг», но лучше все-таки иметь более подробный план. К примеру, если риск осуществился, существуют ли какие-то действия, которые можно предпринять немедленно для уменьшения негативного эффекта? Если да, они должны быть включены в план на случай реализации риска.

Начиная с элементов наибольшей критичности, продумайте планы на случаи реализации риска для каждой строки.

Не забывайте, что этот план не должен заменять планы реагирования на проблему и другие документы, находящиеся в журнале регистрации инцидентов. Во время инцидентов не следует искать матрицу рисков, чтобы свериться с ней. Напротив, матрица, включающая в себя план на случай реализации риска, должна служить инструментом, с помощью которого команда определяет немедленные действия на случай происшествия.

Использование матрицы для планирования

После того как матрица рисков успешно составлена, с нею следует сверяться на всех сессиях планирования, начиная с сессий долгосрочного планирования с участием менеджеров продукта и заканчивая планированием уровня SCRUM с участием разработчиков.

Во время каждой сессии планирования стоит проверить все наиболее критические риски[1]. Рассмотрите их со следующих точек зрения.

[1] Самые критические риски — те, которым присвоена высокая критичность, высокая вероятность или то и другое сразу.

Copyrighted material

- Стал ли этот риск более опасным с того времени, когда мы последний раз исследовали его?
- Следует ли запланировать на текущий период работу по избавлению от него нашей системы?
- Следует ли запланировать на текущий период работу по смягчению рисков в нашей системе и таким образом снизить вероятность или критичность?

Нужно на каждой сессии планирования пересматривать матрицу рисков, а также включать работу над отдельными элементами матрицы (по смягчению рисков или избавлению от них) в процесс расстановки приоритетов.

Если в команде используются для планирования такие инструменты, как Jira или Pivotal Tracker, стоит добавить в эти системы отдельные рабочие элементы для отслеживания каждого риска. Если вы так поступите, затем нужно добавить идентификатор этого элемента в строку соответствующего элемента в матрицу рисков и, наоборот, идентификатор элемента в матрице — в элемент системы планирования.

Поддержка матрицы рисков в актуальном состоянии

Наибольшая трудность при работе с матрицей рисков состоит в том, что она очень скоро может стать неактуальной. Все мы так или иначе склонны составить такой документ, торжественно повесить его на стену, а затем постепенно о нем забыть. Но если вы не будете выделять время на регулярный пересмотр матрицы, она очень быстро сделается неактуальной, а следовательно, абсолютно бесполезной.

Чтобы матрица всегда содержала точные и соответствующие текущему состоянию системы данные, нужно запланировать регулярные ее пересмотры со всеми заинтересованными лицами, включая ваших партнеров и членов команды разработки. Делать это можно раз в месяц или хотя бы раз в квартал. Конкретная периодичность цикла зависит от ваших бизнес-процессов. Если вскоре вам предстоит сессия планирования, предварительно пересмотреть матрицу рисков будет идеальным решением.

Copyrighted material

Участники пересмотра матрицы рисков

Хочу отметить, что будет полезно время от времени менять лиц, участвующих в пересмотре матрицы. Если вы будете предлагать различным людям просмотреть и прокомментировать матрицу рисков, то всякий раз будете получать свежий взгляд, а обсуждение не пойдет по многократно протоптанной дорожке.

Во время пересмотра необходимо:

- *поискать новые риски*. Не появились ли в вашей системе новые риски? Не были ли выявлены не обнаруженные ранее? Не забудьте внести их в свою матрицу;
- *удалить старые риски*. Есть ли в матрице риски, которые уже неактуальны, например, потому, что их возникновение стало невозможно или причина была устранена? Если да, удалите эти элементы из матрицы;
- *обновить критичность и вероятность*. Проверьте, не изменились ли критичность и вероятность каких-либо рисков. Очень часто с помощью внедренных средств смягчения последствий рисков удается успешно снизить вероятность или критичность. Возможно, появилась новая информация, повлиявшая на оценку этих параметров. Если да, внесите соответствующие изменения;
- *просмотреть самые опасные риски*. Просмотрите все риски, обладающие высокой критичностью, высокой вероятностью либо тем и другим одновременно. Обсудите каждый из них в отдельности и удостоверьтесь, что вся имеющаяся о них в матрице информация до сих пор актуальна. Существуют ли новые или обновленные средства смягчения рисков, которые могут быть внедрены? Отслеживаете ли вы этот риск, и если нет, то почему? Подумайте, что еще можно предпринять для улучшения ситуации;
- *просмотреть менее опасные риски*. Продолжая двигаться в сторону снижения критичности и вероятности, просмотрите остальные риски, если позволяет время. Не обязательно просматривать все

риски каждый раз, удостоверьтесь лишь, что пересматриваете самые опасные риски с необходимой частотой. Кроме того, можно запланировать отдельную сессию для детальной проверки менее критических рисков, чтобы уделить им необходимое внимание и удостовериться в отсутствии скрытых или пропущенных причин для присвоения им более высокой позиции в вашем списке.

Информируйте менеджмент о матрице рисков

Необходимо убедиться в том, что менеджеры продуктов и другие представители высшего руководства в курсе информации, содержащейся в матрице рисков. Она может быть эффективным средством коммуникации для обсуждения различных проблем с людьми, которые не вовлечены в рабочие будни вашей команды. Кроме того, тем, кому должно быть известно об имеющихся проблемах, легче держать их в голове с помощью матрицы.

Однажды я наблюдал, как перед выездной сессией руководства была реализована интересная идея. Одному из сотрудников поручили собрать все матрицы рисков целой компании и составить из них один огромный список для этого заседания. После оттуда были удалены все элементы, кроме тех, которым были присвоены либо высокая критичность, либо высокая вероятность, после чего этот список В/В был использован на заседании для общего обсуждения рисков, свойственных продуктам компании, а также обмена опытом, составления требований к объектам, отправляющимся в матрицу рисков, и их описания.

Copyrighted material

8 Смягчение рисков

Графа средств смягчения рисков в матрице используется для того, чтобы показать, какими могут быть (или являются) средства для снижения вероятности, критичности или обоих параметров данного риска. Суть этого процесса в том, чтобы превратить риск В/В[1] в С/В или В/С[2]. Это связано не с *избавлением* от риска, а лишь со снижением вероятности его реализации либо смягчением ее последствий.

Как описано в разделе «План смягчения риска» главы 7, существует базовый процесс смягчения рисков, которого вы можете придерживаться. *План смягчения* детально описывает шаги, которые вы предпримете (немедленно или в ближайшем будущем) для снижения вероятности наступления рисков или критичности его последствий.

Смягчение рисков подразумевает понимание того, какие действия необходимо предпринять в случае возникновения проблемы, в результате чего вы можете максимально снизить негативные последствия. С помощью смягчения рисков вы можете заставить приложение работать настолько полно и эффективно, насколько это достижимо в условиях, когда какие-либо сервисы или ресурсы вышли из строя.

---

[1] Для того чтобы прочесть подробное описание параметров вероятности и критичности и узнать, что такое В/В и др., обратитесь к главе 6.

[2] То же касается и более низких уровней. Например, С/В снижается до С/С или С/Н — до Н/Н.

Copyrighted material

Давайте рассмотрим пример плана смягчения. Предположим, у вас есть база данных, которая используется для работы приложения (наподобие той, что описана в главе 5). Предположим также, что она запущена на высококачественном оборудовании с репликационными компонентами, такими как избыточный массив независимых дисков (RAID), а ресурсов сервера хватает с запасом. Есть основания для уверенности в том, что база данных обладает высокой стабильностью и доступностью. Риску отказа базы данных в матрице присвоена низкая вероятность.

И тем не менее база данных остается элементом, отказ которого приведет к выходу из строя всей системы. Если что-то случится с сервером базы (маловероятно, но возможно), вся ваша система перестанет работать. Поэтому мы должны присвоить этому риску высокую критичность.

Таким образом, это риск Н/В, как было описано в разделе «База данных с заказами: низкая вероятность и высокая критичность риска» главы 6.

Что можно предпринять для смягчения этого риска? Ну, скажем, создать несколько репликаций активной базы данных и постоянно держать их готовыми к вводу в действие (рис. 8.1). Если сервер основной базы данных выйдет из строя, наличие готовой к работе базы с актуальными данными многократно снизит количество времени простоя приложения в ожидании исправления проблемы. Таким образом снижается критичность риска, возможно, даже до уровня Н/С.

Вот как выглядит план смягчения.

Какова разница между смягчением рисков и управлением рисками? В целом это похожие концепты, но есть некоторые различия.

- *Смягчение рисков.* Суть смягчения рисков — в снижении негативного влияния риска путем либо снижения вероятности его реализации, либо критичности ее последствий.
- *Управление рисками.* Суть управления рисками — соблюдение баланса между избавлением от рисков и их смягчением, для чего необходимо осознавать целесообразность, своевременность и финансовую выгоду обоих вариантов.

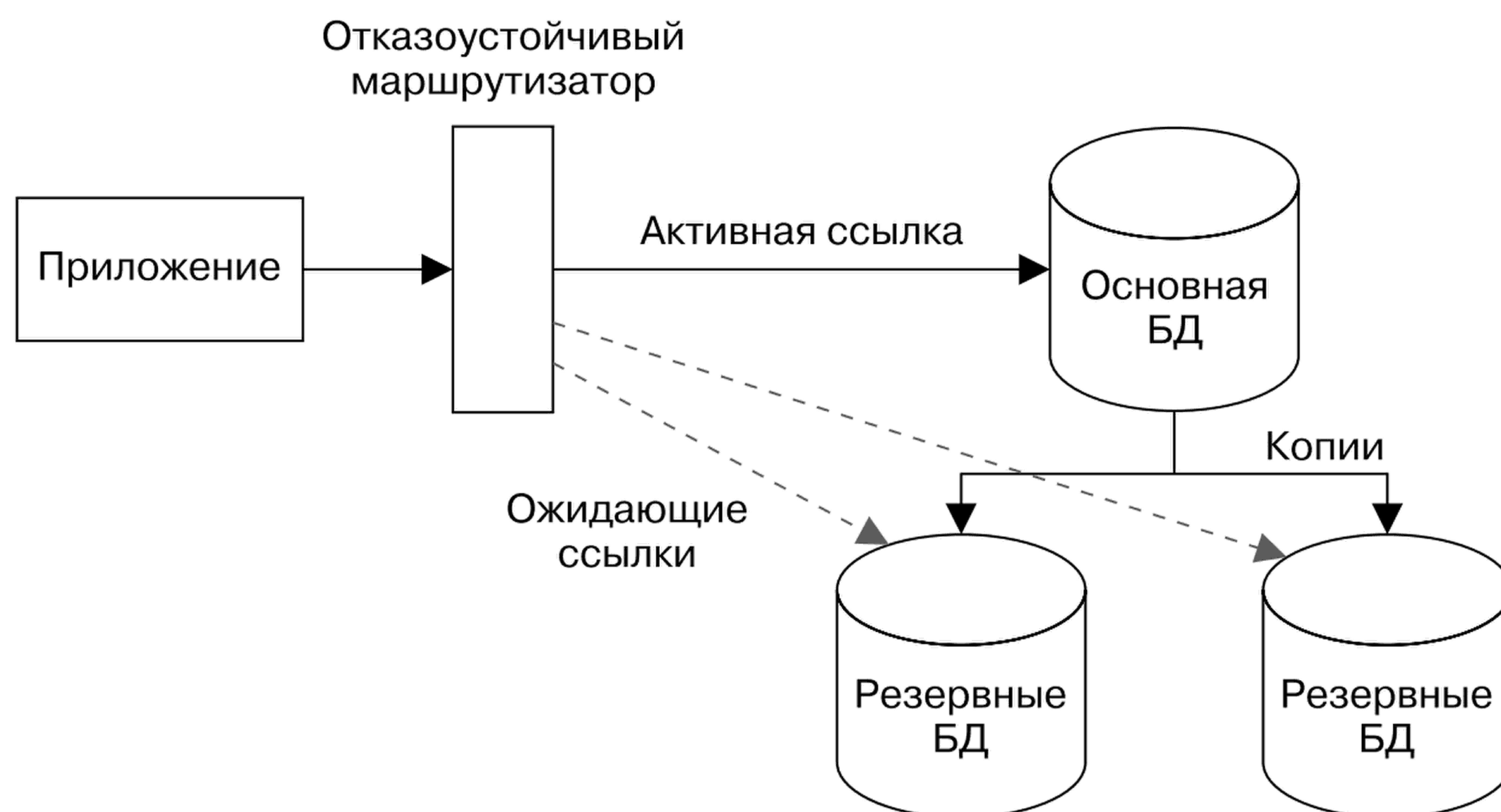


Рис. 8.1. Пример готовой к работе резервной базы данных по плану смягчения

Планы восстановления

Если известный вам риск реализовался, то так или иначе вам придется разбираться с его последствиями. Помочь в этом может *план восстановления* — составленный заранее список действий для устранения этих последствий и приведения системы в рабочее состояние после возникновения проблем, вызванных реализацией риска. Как правило, планы восстановления не затрагивают вероятность риска, а касаются лишь его критичности.

План восстановления — типичный пример средства смягчения риска, которое снижает критичность его последствий в случае реализации. План восстановления описывает, что необходимо предпринять в этом случае. Например, в него могут быть включены:

- действия по как можно более скорому выходу из проблемной ситуации;
- действия по внедрению рабочей среды, в которой негативные последствия проблемы не будут ощущаться так сильно;
- отправка информационных сообщений клиентам о сути возникшей проблемы и действиях, которые они могут предпринять, чтобы снизить негативные последствия для себя;

- перечень лиц, которых необходимо информировать о проблеме, и описание процесса оповещения. Таким образом, все сотрудники компании будут знать о проблеме и смогут предпринять необходимые действия для ее разрешения.

Хороший план восстановления разрабатывается заранее, в процессе работы над планом смягчения рисков для каждого риска в отдельности, поэтому, когда у вас возникают проблемы (риск реализовался), все знают, что нужно делать для их решения.

План восстановления должен включать:

- детальное описание признаков необходимости начала действий по плану восстановления;
- перечень лиц, которые должны быть привлечены к реализации плана восстановления;
- пошаговые инструкции реализации плана восстановления с указанием роли каждого сотрудника;
- организацию уведомления о проблеме всех, кто должен быть оповещен;
- действия, предпринимаемые после того, как проблема решена.

Местонахождение плана восстановления должны знать все члены команды — каждому должно быть хорошо известно, куда обращаться в критической ситуации. Это может быть журнал технической поддержки или страница поддержки на внутреннем интранет-портале. После того как план восстановления был задействован, необходимо проанализировать как возникшую проблему, так и план, чтобы понять, можно ли как-то улучшить или изменить последний.

Само наличие корректного плана восстановления для каждого риска является примером хорошего *плана смягчения риска*, который снижает его критичность.

ПЛАН ВОССТАНОВЛЕНИЯ

Процесс репликации, представленный на рис. 8.1, — зачаток плана восстановления для крайне опасного риска, связанного с выходом из строя базы данных. Для того чтобы он был

полностью готов, необходимо разработать процесс реализации аварийного переключения, критерии запуска переключения, процесс проверки успешности аварийного переключения, а также последующую обработку системы после него.

Планы аварийного восстановления

План аварийного восстановления — это один из видов плана восстановления, в котором указано, что должна предпринять компания в случае возникновения серьезной проблемы конкретного вида. Эти проблемы, как правило, будут иметь высокий уровень критичности, но низкий уровень вероятности.

Примером такой серьезной проблемы, требующей запуска плана аварийного восстановления, будет потеря одного или нескольких дата-центров вашего приложения, что может произойти в результате технических сбоев, форс-мажорных обстоятельств или серьезной бреши в безопасности.

Составлять планы аварийного восстановления можно точно так же, как и другие планы восстановления. Единственное значительное различие между этими двумя видами планов заключается в серьезности риска, о котором в них идет речь. Могут различаться также уровень детализации и степень участия сотрудников в реализации плана.

Как правило, планы аварийного восстановления чаще находятся на виду у сотрудников компании, а также управления и высшего руководства. Могут быть нормированы временные рамки устранения различных типов аварий. Тем не менее все это не особенно отличает их от прочих планов восстановления.

Улучшение ситуации

Смягчение рисков — важный процесс для улучшения доступности и масштабирования ваших приложений путем снижения негативного эффекта, который риск оказывает на систему. Хотя избавиться

Copyrighted material

от риска зачастую невозможно или нецелесообразно, практически всегда можно снизить критичность его последствий, и, как правило, этого достаточно для достижения желаемого уровня работоспособности приложения.

Планы смягчения рисков, используемые совместно с матрицей рисков, являются эффективным инструментом достижения здорового состояния вашего приложения.

Copyrighted material

9 Дни большой игры

Разработка планов восстановления, в том числе аварийных, и последующее размещение их на месте хранения до момента, пока в них не возникнет необходимость, — дело обычное, однако опасное. Если вы тоже так делаете, то я гарантирую: к моменту, когда ваши планы (аварийного) восстановления понадобятся, они будут неактуальны или неполны. Попытка применения неактуальных планов может привести к возникновению большого количества новых проблем.

Вот почему необходимо запланировать регулярные проверки планов (аварийного) восстановления. Процедура тестирования этих планов и других средств смягчения рисков должна глубоко укорениться в компании.

Одна из моделей проверки планов восстановления — запуск Дней большой игры. В День большой игры вы целенаправленно имитируете определенный аварийный режим в системе и наблюдаете, как операторы и инженеры справляются с ним, в том числе как они реализуют соответствующие планы (аварийного) восстановления. Последующий анализ прохождения Дня большой игры подскажет проблемы и необходимые изменения в планах. Таким образом, ваши планы всегда будут актуальными и готовыми к использованию в случае возникновения реальной проблемы.

Среда: стейджинговая или продуктовая?

Возникает вопрос: где нужно тестировать планы восстановления? Можно сделать это на стейджинге, а можно — на живом приложении

Copyrighted material

в производственной среде. Это нелегкий вопрос, и однозначного ответа на него дать нельзя. Давайте рассмотрим оба варианта подробнее.

- *Стейджинговая или тестовая среда.* Тестирование планов восстановления на стейджинге или в тестовой среде — самый безопасный вариант. Его использование позволяет вам провести самые опасные тесты, которые грубо нарушили бы нормальное функционирование производственной среды. Кроме того, вы можете выполнять эти тесты, не опасаясь ошибок, которые могли бы привести к сбоям в живом приложении. Решив использовать стейджинговую или тестовую среду для проверки планов восстановления, помните следующее.
  - Не забудьте убедиться, что тестовая или стейджинговая среда никак не связана с производственной. Эта среда не должна зависеть ни от каких ресурсов производственной среды, а они, в свою очередь, никак не должны быть связаны с ресурсами, участвующими в тестировании (рис 9.1).

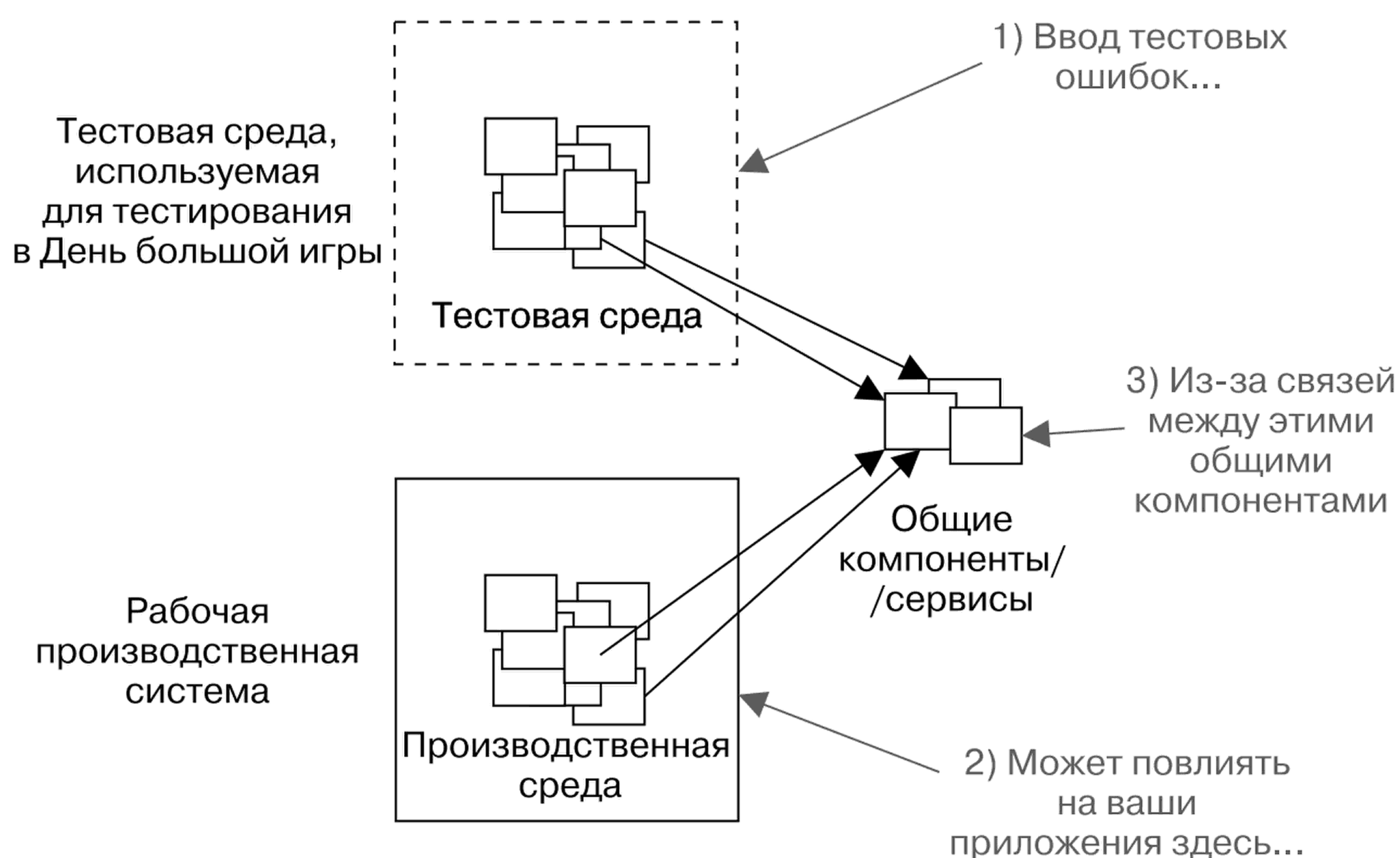


Рис. 9.1. Опасность связей между средами во время тестирования

  - Удостоверьтесь, что стейджинговая или тестовая среда как можно более точно копирует производственную. Да, исполь-

зование стейджинговой или тестовой среды для проверки планов восстановления может быть эффективно, вы можете применять эти типы сред для проверки большого количества разных аварийных сценариев. Но нельзя сказать, что аналогичные проверки в производственной среде привели бы к точно таким же результатам. Так происходит, потому что масштаб продуктовых систем практически всегда крупнее: они размещаются на большем количестве более мощных серверов, содержат больший объем данных и обрабатывают больший трафик в реальном времени. Эти отличия делают определенные типы тестов бессмысленными, если они выполняются в любой среде, кроме производственной. В идеале тестовая среда должна быть масштабирована до размера производственной и заполнена теми же данными, но, как правило, это нецелесообразно с финансовой точки зрения и приводит к сложностям в логистике. Если вы убеждены, что тестирование требует того же масштаба системы, что и в производственной среде, вам следует подумать о проведении проверок именно там.

- *Производственная среда*. Проверка рисков и планов восстановления в производственной среде на первый взгляд кажется дурацкой идеей. Намеренно вызывать сбой в действующем приложении, только чтобы убедиться, что сбоев в продуктовой системе не будет? Ответ прост: если вы тестируете восстановление производственной среды с бодрой и работоспособной командой (иными словами, не посреди ночи), а также в то время дня, когда негативное влияние на ваших клиентов минимально, с большой осторожностью выполняя все шаги тестов, то вам удастся безопасно проверить планы восстановления в производственной среде, в реальных ситуациях. В результате вы получите полезнейшую информацию об эффективности своих действий в реальных аварийных условиях.

  Если вы решили использовать производственную среду для тестирования планов восстановления, помните о следующем:

  - осознавайте последствия намеренного запуска сбоев для клиентов;

- подумайте о бизнес-аспектах такого тестирования. Вы сможете извлечь опыт из своих тестов и снизить долгосрочные риски, но оправдывает ли это дополнительный производственный риск для ваших клиентов?
- выполняйте тесты во время пика работоспособности своих сотрудников (в течение рабочего дня, когда все они, как правило, находятся в офисе), но при этом выбирайте дни, когда влияние на клиентов будет минимальным (например, в предполагаемые промежутки сниженного трафика, а не в важные периоды, скажем, в дни пиков продаж в конце месяца или квартала);
- убедитесь, что у вас есть возможность немедленно реализовать необходимые исправления или откатить неудачные.

Недостатки запуска Дней большой игры в производственной среде

Дни большой игры в производственной среде нужно планировать и отслеживать очень тщательно. Если День запланирован правильно, его проведение может выявить массу проблем в производственной среде. Вот примеры Дней большой игры, которые вы можете запустить.

- ❑ **Выход из строя сервера.** Что произойдет, если один из серверов вашей системы отключится? Попробуйте намеренно выключить один из них. Если в системе достаточное количество резервов, никакого негативного влияния продуктовая система не ощутит, но таким образом вы сможете проверить обнаружение этой проблемы в системе и свой план восстановления по замещению вышедшего из строя сервера.
- ❑ **Разделение сети.** Каковы последствия разделения или отключения сети? Вы можете эмулировать тщательно спланированное разделение сети без значительного риска для работающего приложения. Зато в результате основательно протестируете выявление этой проблемы, оповещение о ней и последующие действия команды по реагированию на событие в системе.

Copyrighted material

- **Выход из строя дата-центра.** Что произойдет, если весь дата-центр выйдет из строя? При тщательной подготовке приложение может успешно обработать и такое событие. Каковы будут ваши действия по ликвидации этой аварии?
- **Случайные отказы.** Что будет, если вы намеренно запустите в системе меньшие случайные ошибки? Восстановится ли система после них в ожидаемой степени?

Последний пункт списка можно считать самым угрожающим во многих отношениях. Почему? В случае выхода из строя дата-центра или сервера вы все-таки можете представить себе, что произойдет. Скорее всего, у вас есть планы реагирования на такую ситуацию (а если нет, позаботьтесь о них). А вот при случайной проблеме, пусть даже меньшей по масштабу, вы почувствуете, что теряете контроль над ситуацией. Отчасти это так и есть. Именно такие непредсказуемые события представляют наибольшую угрозу для разработки высокодоступных систем с тщательно обработанными рисками.

Chaos Monkey

Netflix задала новую планку работы со случайными проблемами. В этой компании есть система под названием Chaos Monkey, встроенная в приложение. Эта система регулярно случайным образом формирует непредсказуемые сбои в работающем приложении — *в производственной среде с реальными клиентами*. Инженерам и операторам, управляющим приложением, неизвестно, что именно и как именно делает Chaos Monkey. Напротив, предполагается, что инженеры готовы запустить правильные процессы восстановления системы и смягчения последствий, так что проблемы, генерируемые Chaos Monkey, могут быть решены или обработаны без влияния на конечных пользователей.

Chaos Monkey действует только в рабочее время, когда все инженеры находятся в офисе и готовы реагировать на любые проблемы, если те не будут решены автоматически. Chaos Monkey, по сути, стимулирует и, более того, *требует* наличия высокодоступных, самодостаточных сервисов и приложений, которые могут восстановиться и вернуться к работе без участия человека. Такое тестирование проводится в течение рабочего дня, когда сотрудники доступны, а приложение

> не слишком загружено; если бы проблема произошла вечером, когда клиентов больше, это было бы более рискованно, а сотрудников пришлось бы заставлять выйти на работу. Этот совершенно новый подход отлично работает в Netflix.
>
> Chaos Monkey — отличный пример успешного применения Дней большой игры, и в Netflix проделали огромную работу над инфраструктурой этих дней. Но, конечно, потребовалось довольно много усилий, ресурсов и упорства, прежде чем Chaos Monkey можно было эффективно и безопасно запустить в производственной среде Netflix. Конечно, не стоит прибегать к Chaos Monkey в качестве первой пробы тестирования в производственной среде, но это может быть своего рода целью для дальнейшей работы, если только вашей компании хватит целеустремленности, чтобы ее достичь.

Дни большой игры

Дни большой игры — важная часть тестирования, которая поможет проверить, будет ли ваша производственная среда постоянно полноценно функционировать. С их помощью вы сможете безопасно протестировать свои процессы и планы технической поддержки, в результате чего в случае реальной надобности они гарантированно будут работать.

Если все правильно рассчитать, Дни большой игры весьма значительно повысят доступность системы при масштабировании, а также снизят риск серьезных проблем или сбоев в производственной среде.

Copyrighted material

10 Создание систем со сниженными рисками

В главе 8 мы рассмотрели, как можно смягчить риски, существующие в ваших системе и приложениях. Однако вы можете начать действовать заранее, обеспечивая снижение рисков уже на стадии построения приложения. В этой главе рассматривается несколько подобных техник. Перечень будет далеко не полным, но вы сможете подумать о снижении рисков во время создания и расширения приложений.

Избыточность

Обеспечение избыточности — очевидное средство увеличения доступности и надежности приложения. Как следствие, снизится уровень рисков. В то же время избыточность усложняет приложение, что поднимает уровень рисков. Поэтому очень важно контролировать сложность при повышении избыточности, чтобы уровень риска все-таки снижался.

Вот несколько примеров безопасных улучшений путем увеличения избыточности.

- При проектировании приложения планируйте его безопасную одновременную работу на нескольких независимых компонентах

Copyrighted material

оборудования (например, работу на нескольких параллельных серверах или наличие избыточного количества дата-центров).

- При проектировании приложения заложите возможность независимого запуска разных задач. Это может быть полезно для восстановления после выхода из строя различных ресурсов без значительного усложнения приложения.
- При проектировании приложения учтите возможность асинхронного запуска задач. Таким образом запросы на исполнение задач и их запуск могут быть поставлены в очередь, что не повлияет на основной процесс работы приложения.
- Локализуйте состояния в определенных областях. Это позволит снизить необходимость управления состояниями в других частях приложения.
- Где только возможно, используйте *идемпотентные интерфейсы*. Этот термин означает, что интерфейсы могут быть при необходимости вызваны несколько раз подряд для гарантии того, что запрошенное действие в конечном итоге выполнено, но выполнено фактически лишь один раз. Использование идемпотентных интерфейсов упрощает восстановление после ошибок с помощью простых механизмов повтора.

Примеры идемпотентных интерфейсов

Рассмотрим в качестве примера автомобиль, в котором есть интерфейс, контролирующий скорость движения. Идемпотентный интерфейс выполнял бы запрос «Установить текущую скорость автомобиля равной 50 км/ч». А неидемпотентный — «Повысить текущую скорость автомобиля на 8 км/ч».

Таким образом, сколько бы раз мы ни обращались к идемпотентному интерфейсу, только первый вызов привел бы к каким-то действиям. Повторные вызовы никак не повлияли бы на скорость машины — неважно, один или десять раз вы велели бы машине ехать со скоростью 50 км/ч. А вот неидемпотентные интерфейсы меняют скорость машины каждый раз, когда вызываются. Каждый раз, когда машина

Copyrighted material

получает сигнал увеличить скорость на 8 км/ч, она начинает ехать быстрее и быстрее.

Таким образом, при идемпотентном интерфейсе водитель этой автоматической машины должен лишь сказать ей, с какой скоростью нужно ехать. Если по какой-то причине ему кажется, что автомобиль не получил этот запрос, он может без малейшего риска просто посылать его снова до тех пор, пока не увидит, что скорость автомобиля составляет ровно 50 км/ч.

С неидемпотентным интерфейсом водитель, желающий ехать со скоростью 50 км/ч, вынужден отправлять серию команд, заставляющих машину ускоряться. Если одна или несколько из этих команд не сработали, водителю нужно, прибегнув к помощи какого-то механизма, измерить скорость машины, а затем решить, нужно ли отправлять запрос на увеличение скорости повторно. Просто повторить его нельзя — сначала необходимо проверить, нужно ли делать это. Очевидно, что это более сложная и, следовательно, сильнее подверженная риску ошибок операция.

Использование идемпотентных интерфейсов позволяет водителю выполнять простые операции с меньшим риском ошибок, чем при использовании неидемпотентных.

Повышение избыточности, ведущее к росту сложности

Рассмотрим примеры, в которых избыточность ведет к росту сложности приложения. Многие из таких изменений на первый взгляд кажутся полезными, но на самом деле создают дополнительную сложность, которая приносит больше вреда, чем пользы (во всяком случае, для большинства приложений).

Скажем, можно обсудить создание параллельной имплементации системы. В этом случае, если одна из систем выходит из строя, другая может быть использована для реализации всех необходимых функциональностей. Это может быть полезно для некоторых приложений, где критически важна высокая доступность (например,

используемых на космическом корабле), однако чаще всего такие меры не оправдывают себя и приводят к переусложненности. А высокая сложность означает высокие риски.

Другой пример — явное разделение действий. Микросервисы — прекрасная модель улучшения качества приложения и, как следствие, снижения риска. В главе 12 приведена подробная информация об использовании сервисов и микросервисов. Однако и это средство должно быть применено с умом: если перестараться при декомпозиции микросервисов, проектируя систему, это может привести к значительному увеличению ее сложности и, как следствие, к возрастанию рисков.

Независимость

Несколько компонентов, использующих общие возможности или какие-то другие компоненты, могут представляться независимыми, но на самом деле все они зависят от какого-то одного элемента (рис. 10.1).

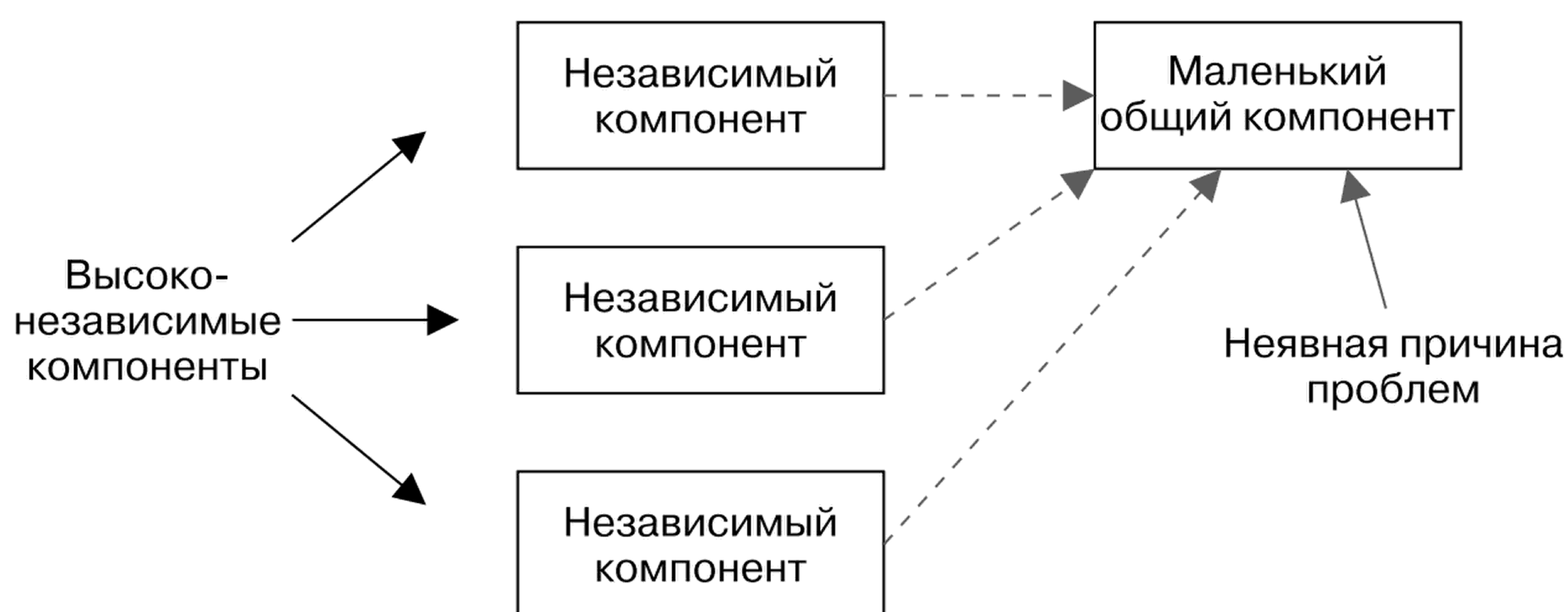


Рис. 10.1. Работа с общими компонентами нарушает независимость

Если эти общие компоненты незначительны или неизвестны, то выход из строя одного из них может привести к аварии во всей системе.

Рассмотрим приложение, работающее на пяти независимых серверах. Пять серверов нужны, чтобы повысить доступность и уменьшить риск выхода из строя единственного сервера, в результате

чего приложение перестанет работать. Это проиллюстрировано на рис. 10.2.

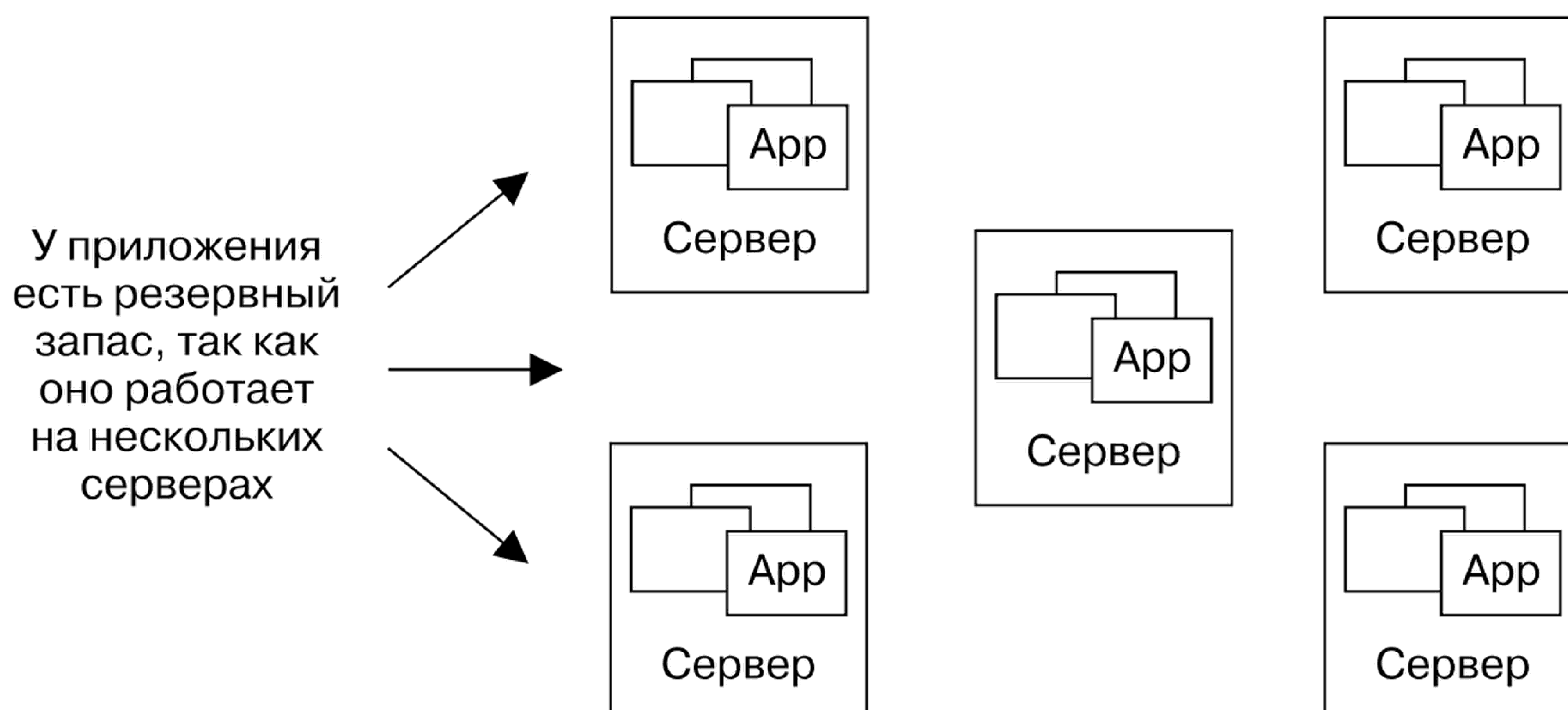


Рис. 10.2. Независимые серверы...

А что, если эти пять серверов — не физические, а виртуальные, запущенные на единственном физическом (рис. 10.3)? Или, к примеру, эти пять серверов запущены в одном серверном шкафу? Что произойдет, если электроснабжение шкафа прекратится? Что произойдет, если физический сервер выйдет из строя?

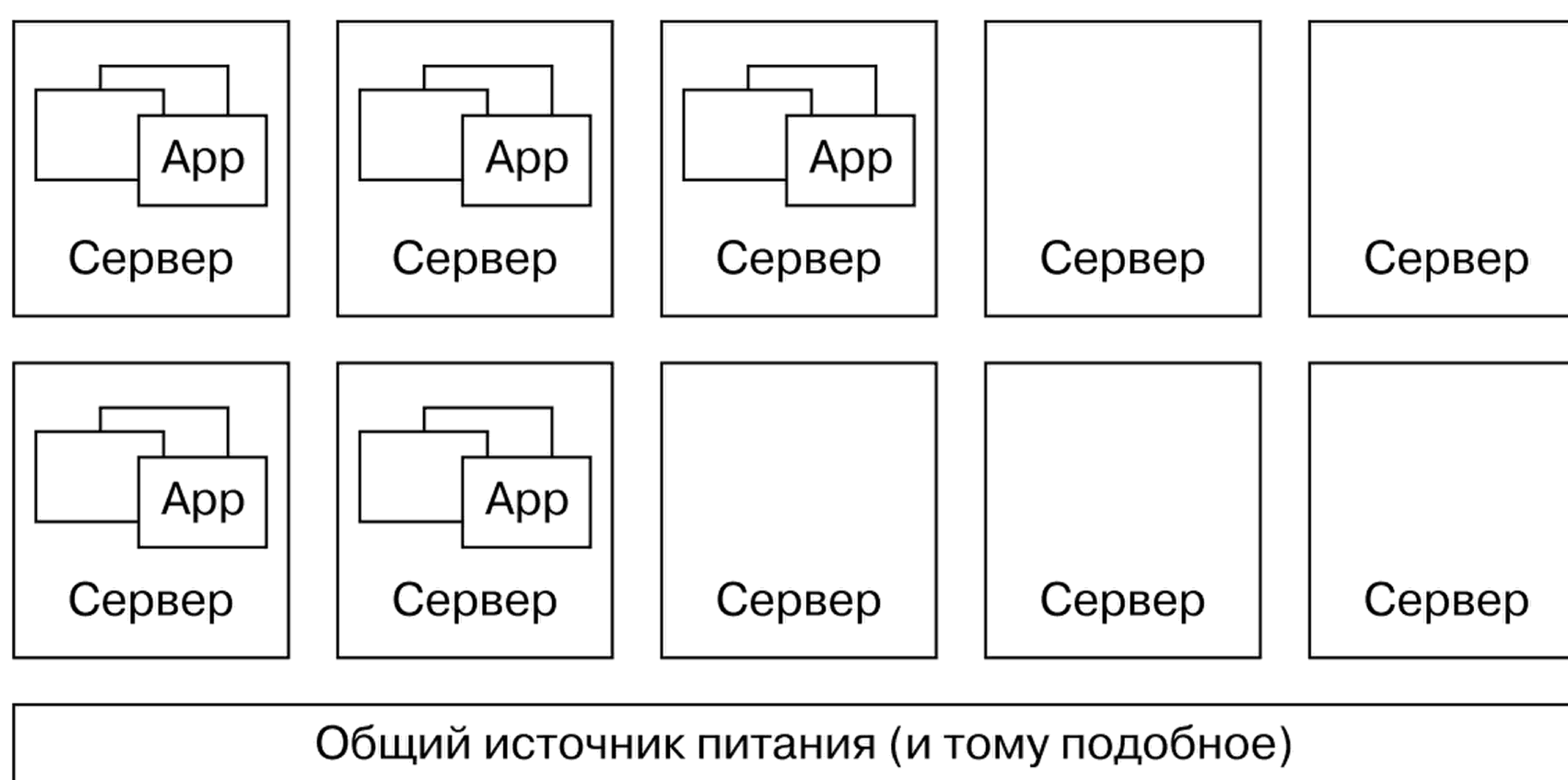


Рис. 10.3. ...Не так независимы, как кажется

Ваши независимые серверы могут быть не так независимы, как кажется!

Безопасность

Вечной проблемой программного обеспечения являются хакеры. Безопасность и ее отслеживание были важнейшей частью создания систем даже до того, как стало известно о крупномасштабных веб-приложениях.

Однако веб-приложения становятся крупнее, усложняются, хранят большее количество данных и обрабатывают большие потоки трафика. В сочетании с высокой важностью данных, хранящихся в этих приложениях, это привело к расширению рядов злоумышленников, мечтающих получить доступ к чужой информации. Целью взлома может быть доступ к засекреченной личной информации или нарушение работоспособности крупных приложений. Некоторые хакеры делают это ради денежной выгоды, а некоторые — просто для собственного удовольствия. Независимо от причин и результатов своих действий эти ребята представляют собой огромную проблему.

Безопасность веб-приложений здесь не рассматривается, но проектирование хорошо защищенных приложений чрезвычайно важно как для высокой доступности, так и для смягчения рисков в крупномасштабных приложениях. Вы непременно должны включить аспекты безопасности в анализ рисков и планы по их смягчению, а также рассматривать их во время процесса разработки. Подробное рассмотрение этого вопроса выходит за рамки данной книги.

Простота

Сложность — враг стабильности. Чем сложнее становится система, тем менее она стабильна. Чем менее она стабильна, тем рискованнее она становится и тем ниже оказывается уровень ее доступности.

Хотя ваши приложения растут и становятся все более сложными, старайтесь поддерживать простоту при проектировании архитектуры и разработке приложения. Таким образом вы сможете добиться удобства сопровождения приложения, его безопасности и низких уровней риска в нем.

Copyrighted material

При создании современного программного обеспечения наблюдается тенденция к излишней сложности, если архитектура приложения основана на микросервисах. Такая архитектура значительно снижает сложность отдельных компонентов, позволяя создавать простые для понимания сервисы с использованием незамысловатых техник. Однако при снижении сложности отдельных микросервисов повышается количество модулей, необходимое для реализации крупномасштабного приложения. А наличие большого количества совместно работающих независимых модулей повышает взаимозависимость между ними, что ведет к возрастанию общей сложности приложения.

Таким образом, при создании приложения, основанного на микросервисах, очень важно соблюдать баланс между упрощением отдельных сервисов и усложнением системы в целом.

Самовосстановление

Создание саморегулирующихся и самовосстанавливающихся процессов в ваших приложениях может снизить риск перебоев доступности.

Как обсуждалось в главе 3, если вы стремитесь к уровню доступности 5 девяток, то можете позволить себе не более 26 с недоступности ежемесячно. Даже если вы готовы удовольствоваться 3 девятками, то ваш максимум — 43 мин недоступности в месяц. Если выход из строя какого-либо сервиса заставляет кого-то вскакивать среди ночи, чтобы выявить, диагностировать и исправить проблему, эти 43 мин будут исчерпаны мгновенно. Один-единственный сбой может стоить вам потери уровня 3 девятки. А уж чтобы соответствовать 4 или 5 девяткам, вам придется обеспечить исправление проблем вообще без участия человека.

Здесь-то и вступают в игру самовосстанавливающиеся системы. Сам этот термин вызывает ассоциации с чем-то запредельно сложным и высокотехнологичным, но это не всегда так. Самовосстанавливающаяся система, например, может всего лишь включать в себя балансировку нагрузки на нескольких серверах так, что запрос переадресуется новому серверу, если основной не смог его обработать. Это вполне полноценный пример.

Copyrighted material

Можно использовать несколько уровней самовосстанавливающихся систем, от простых до сложных. Вот несколько примеров.

- Балансировщик нагрузки, переадресующий трафик новому серверу, если предыдущий не смог его обработать.
- Готовая к работе резервная база данных, полностью дублирующая основную. Если основная продуктовая база по какой-либо причине вышла из строя, резервная немедленно перехватывает главенствующую роль и начинает обрабатывать запросы.
- Сервис, который повторяет запрос, если тот завершился с ошибкой, предполагая, что изначальный запрос был неуспешен по какой-то случайной причине, а новая попытка будет благополучно обработана.
- Система обработки запросов, отслеживающая отложенные задачи: если какой-либо запрос оказывается неуспешным, позднее он может быть запланирован для нового обработчика, что повышает вероятность успешного завершения этого запроса и снижает вероятность потери задач.
- Фоновые процессы (например, Chaos Monkey в Netflix), постоянно генерирующие в системе ошибки для проверки того, что система может после них успешно восстановиться.
- Сервис, который вызывает несколько других сервисов, индивидуально разработанных и управляемых, для проведения одной и той же операции. Если результаты работы сервисов одинаковы, они используются. Если один или несколько независимых сервисов возвращают различающиеся результаты, то этот результат выбрасывается, а данный сервис отключается для проверки.

Это лишь несколько примеров. Отмечу также, что в последних пунктах списка значительно увеличена сложность системы и об этом нельзя забывать. Используйте самовосстанавливающиеся системы только в том случае, если они дают значительное снижение рисков в системе при минимальном увеличении сложности. Избегайте переусложнения систем и архитектуры, если они обеспечивают больший уровень восстановления, чем вам на самом деле требуется, но за счет повышения рисков и вероятности сбоев в системе.

Оперативные процессы

Работа систем программного обеспечения зависит от людей, а люди не могут не делать ошибок. Однако использование надежных оперативных процессов, минимизирующих влияние людей на систему и ограничивающих их доступ к области, где их участие необязательно, может значительно снизить вероятность возникновения ошибок.

Используйте документирование ритмично повторяющихся процессов, чтобы снизить один из самых значительных аспектов человеческого фактора — забывчивость: люди склонны забывать шаги, выполнять их не в том порядке, делать ошибки в выполнении шагов. Документирование способно справиться с этой важнейшей проблемой, но есть и другие: люди делают ошибки, допускают опечатки, думают: «Я знаю, что делаю», в то время как это не так. Их действия всегда неодинаковы, и они нигде не фиксируются. И наконец, на действия людей неизбежно влияет их эмоциональное состояние.

Поэтому, чем больше степень автоматизации процессов, которые обычно выполняют в вашей продуктовой системе люди, тем меньше ошибок туда будет внесено и тем выше вероятность того, что все задачи будут благополучно выполняться.

Перезагрузка сервера

Допустим, вам приходится регулярно перезагружать сервер или несколько серверов с какой-то конкретной целью (в данном случае неважно, с какой именно). Вы можете просто авторизоваться на этом сервере, получить права суперпользователя и выполнить команду reboot. Но даже это может привести к следующим проблемам.

- Возможно, придется дать права суперпользователя (для выполнения команды перезагрузки) и разрешить заходить в производственную среду всем, кому может понадобиться проделать эту операцию.
- Будучи авторизованными на вашем продуктовом сервере, эти люди могут случайно выполнить другую команду, которая выведет сервер из строя.

- В конце концов, кто-то может зайти на сервер с правами суперпользователя, имея отнюдь не благие намерения, и проделать какое-либо вредоносное для сервера действие, например запустить на Linux команду rm-rf.
- Скорее всего, у вас не будет никаких записей, что за действие было сделано, кем и когда.

Поэтому вместо ручного процесса перезагрузки сервера можно внедрить автоматический. Кроме собственно перезагрузки, вы можете получить следующие дополнительные преимущества.

- Уменьшится необходимость в выдаче данных для входа на ваши продуктовые серверы, в результате чего снизится вероятность как ошибок, так и намеренных вредоносных действий.
- Можно фиксировать все действия, предпринятые для выполнения перезагрузки.
- Можно фиксировать, кто запросил выполнение перезагрузки.
- Можно проверять, есть ли у лица, запросившего перезагрузку, необходимые для этого права (скажем, вы можете дать права строго для выполнения перезагрузки определенной группе лиц без выдачи им прав доступа к чему-то еще).
- Можно удостовериться, что перед перезагрузкой сервера выполнены все остальные необходимые действия. Например, сервер временно удален из балансировщика нагрузки, все запущенные приложения корректно завершены и т. д.

Как видите, автоматизация этого процесса позволяет избежать серьезных ошибок и дает возможность контролировать, кем и какие действия были предприняты.

Copyrighted material

Часть III
Сервисы и микросервисы

Сервис — обособленная закрытая система, обеспечивающая функционирование отдельной бизнес-возможности и используемая при разработке одного или нескольких крупных продуктов.

Copyrighted material

11 Зачем нужны сервисы

Когда-то все приложения были огромными обособленными многокомпонентными монолитами, которые сосредотачивали в себе абсолютно *все* бизнес-возможности приложения. Для реализации улучшения какой-нибудь возможности какой-то разработчик должен был внести изменения в целое приложение, как и все остальные специалисты, которые трудились над ним. В результате все они работали в одном и том же пространстве, наступая друг другу на пятки и внося противоречивые изменения, что приводило к сбоям, проблемам и ошибкам.

Сейчас при использовании сервисно-ориентированной архитектуры создаются индивидуальные сервисы, каждый из которых отвечает за конкретную бизнес-задачу, а затем отдельные сервисы связываются между собой для обеспечения бизнес-логики всего приложения.

Монолитное приложение

На рис. 11.1 изображено обособленное крупное приложение с многокомпонентной сложной архитектурой.

Именно так будет выглядеть большинство приложений, если они изначально сконструированы как монолитные и остаются такими при дальнейшем развитии. Как вы можете видеть на рис. 11.1, над пересекающимися областями приложения работают пять независимых

Copyrighted material

команд разработки. Понять, кто над каким фрагментом приложения трудится в данный момент, невозможно, и легко представить себе, к каким конфликтам и проблемам в коде это может привести! Качество кода и, как следствие, качество и доступность приложения неизбежно снижаются. Кроме того, различным командам разработки становится труднее и труднее вносить изменения: приходится учитывать работу других команд, решать конфликты между изменениями и стараться не допустить ущерба для всей организации.

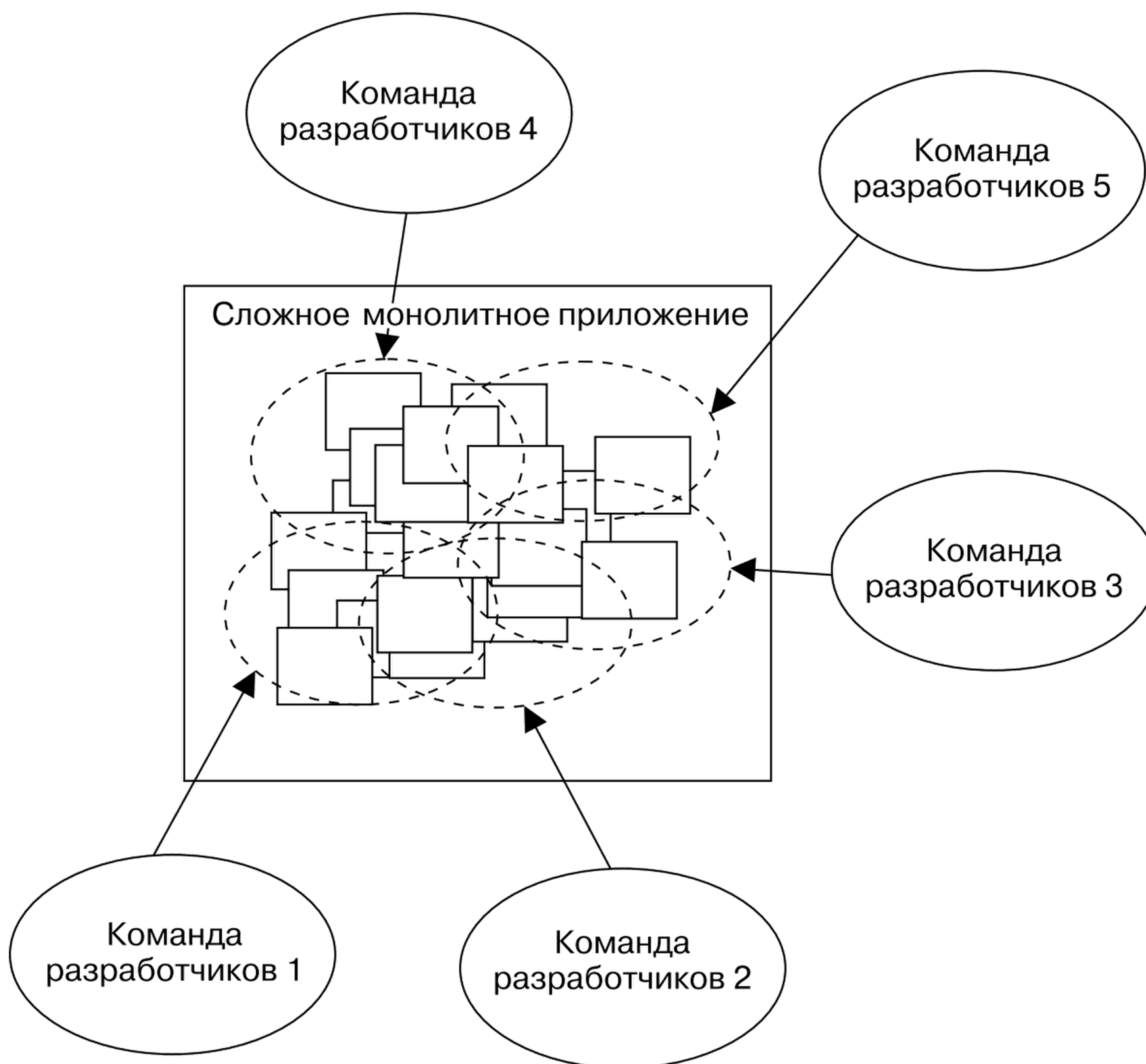


Рис. 11.1. Большое и сложное монолитное приложение

Сервисно-ориентированное приложение

На рис. 11.2 представлено то же самое приложение, организованное как набор сервисов.

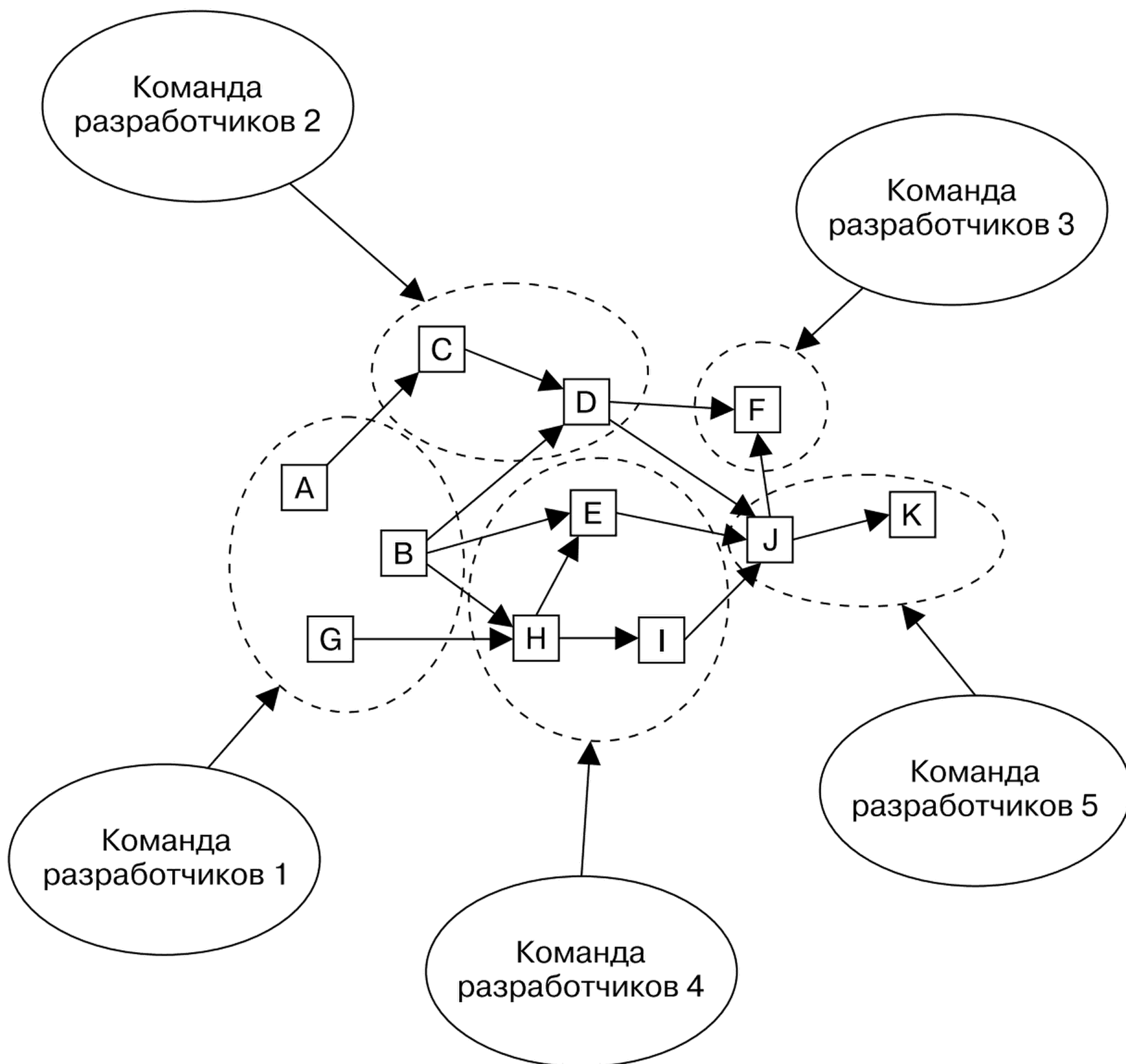


Рис. 11.2. Большое и сложное приложение

Принадлежность каждого сервиса ясно определена, и у каждой команды есть только своя четко очерченная область ответственности.

Сервисно-ориентированные архитектуры предоставляют возможность разделить приложение на несколько отдельных доменов, каждым из которых управляет определенная группа лиц внутри организации. В результате образуются отдельные зоны ответственности, что критически важно для создания крупномасштабных приложений: необходимые задачи могут быть выполнены для каждого сервиса в отдельности, что исключает риск повлиять на результат труда программистов из других групп, работающих над тем же приложением.

При создании крупномасштабных приложений сервисно-ориентирование обеспечивает следующие преимущества.

- *Решения о масштабировании*. Решения о масштабировании могут приниматься на низком уровне, что оптимизирует эффективность и организацию системы.
- *Разделение обязанностей для эффективной концентрации*. Вы можете закреплять определенные функциональности за конкретными командами, они будут отвечать только за масштабирование и доступность своей локальной области, что обеспечивает необходимый эффект для системы в целом.
- *Локализация сложности*. Когда архитектура основана на сервисах, вы можете воспринимать их как черные ящики — только те, кто отвечает за работу этого сервиса, должны разбираться в его сложном устройстве. Другим разработчикам достаточно лишь знать о возможностях, которые сервис предоставляет, им нет необходимости понимать внутренний механизм его работы. Такое разделение сложной информации облегчает создание сложных приложений и позволяет эффективно управлять ими.
- *Тестирование*. Приложения, основанные на сервисах, проще тестировать, чем монолитные, что повышает их надежность.

При всем этом, однако, если разграничение сервисов спроектировано неверно, сервисно-ориентированные архитектуры могут увеличить сложность системы в целом. Это может привести к снижению возможности масштабируемости и ухудшению доступности системы. Очень важно выбирать сервисы и разграничивать их между собой со всей возможной тщательностью.

Преимущества выделенного владения сервисами

Давайте рассмотрим пару сервисов. На рис. 11.3 можно видеть два сервиса, которыми владеют две разные команды. Левый сервис использует возможности, которые предоставляет правый.

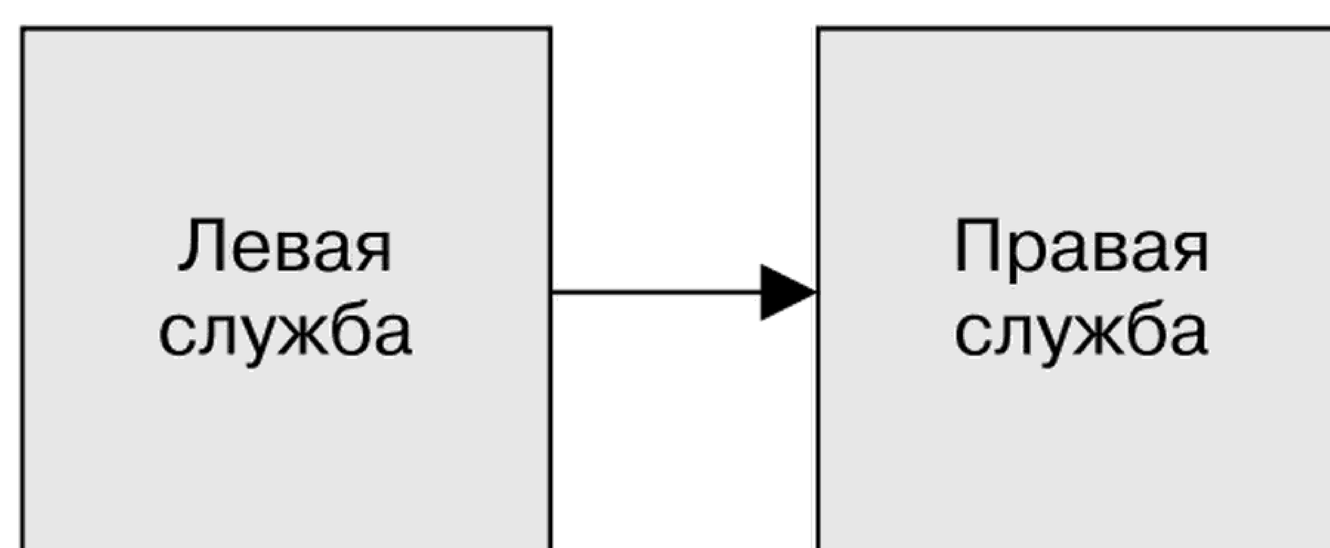


Рис. 11.3. Пара сервисов

Давайте рассмотрим эту ситуацию с точки зрения владельца левого сервиса. Очевидно, что команда должна хорошо разбираться в коде, структуре, компонентном составе, взаимосвязях, зависимостях и других особенностях своего сервиса. Но что им нужно знать о правом сервисе? Для начала следующее:

- возможности, предоставляемые этим сервисом;
- как задействовать эти возможности (синтаксис API);
- смысл и результаты вызова этих возможностей (семантика API).

Это базовая информация, которую должна знать команда левого сервиса. А что им не требуется знать? Гораздо больше! Не нужно знать:

- является ли правый сервис единичным сервисом или конструкцией из нескольких подсервисов;
- от каких сервисов зависит выполнение своих обязанностей правым сервисом;
- на каком языке (или языках) программирования написан правый сервис;
- какие оборудование или системная инфраструктура необходимы для работы правого сервиса;
- даже кто именно управляет правым сервисом (а вот информация о том, как связаться с владельцем сервиса при возникновении проблем, безусловно, пригодится).

Правый сервис может быть настолько простым или сложным, насколько требуется (рис. 11.4). Но, с точки зрения команды левого сервиса, можно рассматривать его просто в виде черного ящика

(рис. 11.5). Зная только способ взаимодействия с этим ящиком (API), они могут полноценно использовать все его возможности.

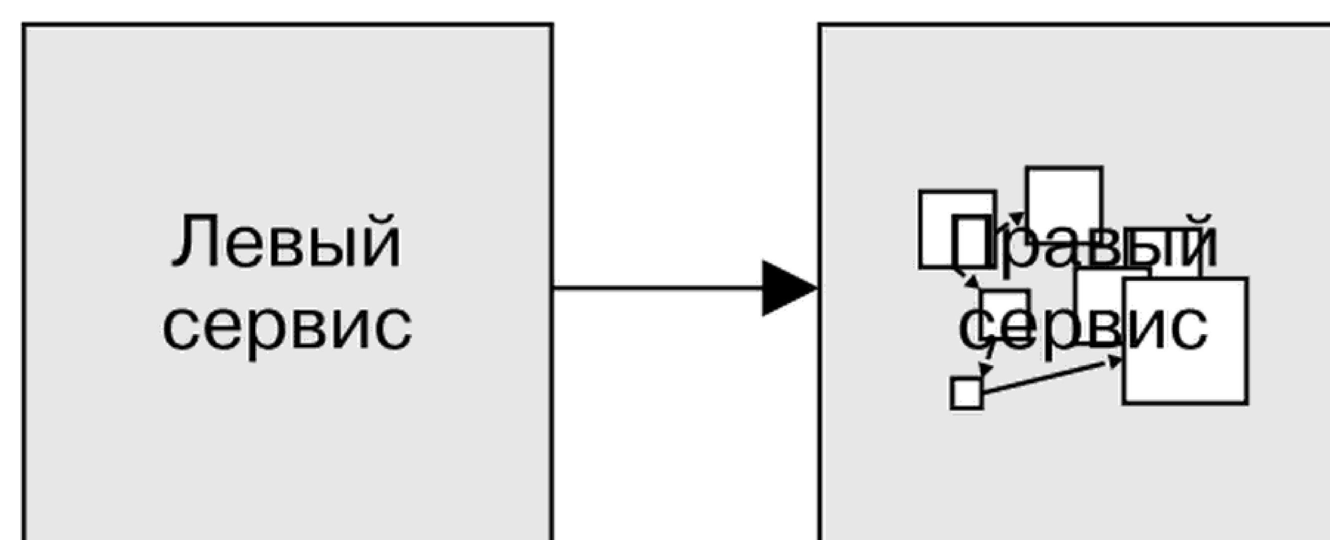


Рис. 11.4. Что находится внутри правого сервиса

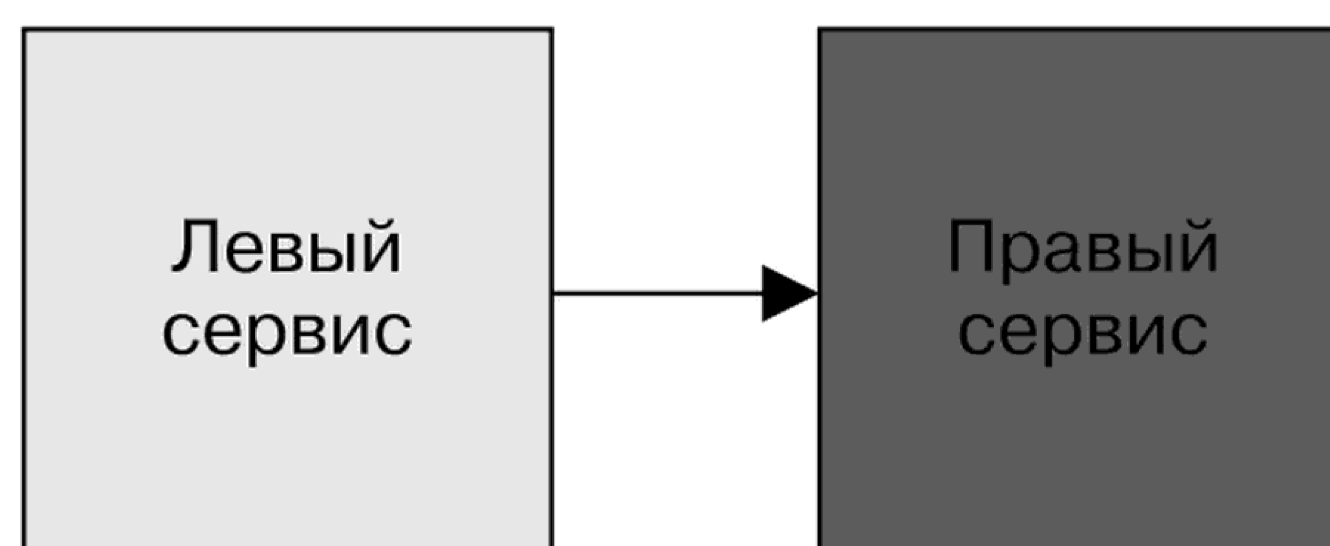


Рис. 11.5. Знать устройство правого сервиса для взаимодействия с ним не нужно

Для взаимодействия левому сервису нужен *контракт* с правым сервисом. В контракте указано все, что нужно левому сервису для того, чтобы использовать правый.

Контракт состоит из двух частей.

- *Возможности сервиса (API).*
  - Что делает сервис?
  - Способ обращения к сервису и значение каждого вызова.
- *Ответственность сервиса.*
  - Как часто может быть использован API?
  - Когда он может быть использован?
  - Насколько быстро реагирует API?
  - Может ли API адаптироваться?

Вся информация, содержащаяся в контракте, сообщает владельцам правого сервиса о том, как функционирует левый. Пока сервисы

взаимодействуют в соответствии с этим контрактом, левый сервис не должен знать или беспокоиться о том, как именно правый выполняет свои обязательства.

Последняя часть контракта, *ответственность*, называется *соглашением сервисного уровня* (Service-Level Agreement, SLA). Это чрезвычайно важный компонент, позволяющий левому сервису зависеть от работы правого, не зная ничего о том, как правый сервис работает.

Мы подробно обсудим SLA в главе 18.

Когда владение сервисами четко разделено, каждая команда концентрируется только на тех фрагментах системы, за которые несет прямую ответственность, и предоставляет *контракты API* для тех сервисов, чья работа зависит от их подопечных. Такое разделение ответственности существенно облегчает рост вашей организации и увеличение количества команд, а поскольку существенного взаимодействия между командами не требуется, удаленность команд (организационная или физическая) не так уж важна. Регулярно обновляя контракты, вы можете обеспечивать рост своей организации для создания более крупных и сложных приложений.

Преимущества масштабирования

Разные части вашего приложения требуют разного масштабирования. Компонент, который генерирует главную страницу приложения, используется гораздо чаще, чем тот, что отвечает за страницу настроек.

Используя сервисы с API и заключая API-контракты между ними, вы можете определить и реализовать необходимую степень масштабирования для каждого сервиса в отдельности. Это значит, что если ваша главная страница вызывается чаще всего, вы можете улучшить оборудование для запуска этого сервиса, а развитие сервиса страницы настроек пока отложить.

Управляя масштабированием каждого сервиса по отдельности, вы можете получить следующие преимущества:

Copyrighted material

- обеспечивать более точное масштабирование, привлекая к решениям о масштабировании команду, которая хорошо знает сервис и его возможности;
- не масштабировать один компонент только потому, что этого требует другой, и таким образом экономить ресурсы системы;
- передать ответственность за решения о масштабируемости той команде, которая лучше всего разбирается в нуждах сервиса, так как владеет им.

Copyrighted material

12 Использование микросервисов

Сервис предоставляет определенные возможности, которые используются остальной частью приложения. В качестве примера можно привести платежные сервисы (с их помощью покупатели оплачивают заказы), сервисы управления учетными записями (они управляют создающим эти записи компонентом) или сервисы уведомлений (они обеспечивают оповещение пользователей об определенных событиях).

Сервис — автономный компонент, и ключевое слово в этом определении — «автономный». Сервисы должны соответствовать следующим критериям.

- *Основываться на собственной базе кода.* У каждого сервиса есть собственная база кода, обособленная от остальных баз.
- *Управлять собственными данными.* Сервис, участвующий в управлении приложением, хранит всю необходимую ему информацию в собственной базе данных. Единственный способ получить к ней доступ — через определенное сервисом API. Ни один сервис не может получить доступ к данным или информации о состоянии другого сервиса.
- *Предоставлять возможности другим сервисам.* Сервис имеет четко определенный набор возможностей и предоставляет их

Copyrighted material

другим сервисам в вашем приложении. Другими словами, он предоставляет API.

- *Пользоваться возможностями других сервисов*. Сервис использует четко определенный набор возможностей, предоставляемых другими сервисами по конкретному стандарту. Другими словами, он использует API других сервисов.
- *Иметь единственного владельца*. Каждый сервис принадлежит определенной команде разработки в организации и обслуживается только ею. Эта команда может владеть несколькими сервисами и отвечать за них, но несколько команд не могут владеть одним сервисом.

Что должно быть сервисом?

Как же решить, какой фрагмент приложения или системы должен быть выделен в отдельный сервис?

Хороший вопрос, но дать на него единственно верный ответ невозможно. Некоторые компании, «сервирующие» свои приложения, разделяют их на множество (сотни или даже тысячи) крошечных микросервисов. Другие — на несколько довольно больших сервисов. Единственного верного решения не существует, но все же заметна тенденция к уменьшению микросервисов и увеличению их количества. Такие технологии, как Docker, обеспечивают существование топологий с большим количеством микросервисов.

В этой книге терминами *«сервисы»* и *«микросервисы»* обозначается одно и то же понятие.

Разделение на сервисы

Как определить, где будут проходить границы сервисов? Организация и культура компании, а также специфика приложения играют в этом не последнюю роль.

Далее приведены некоторые рекомендации, которые могут помочь в принятии этого решения. Помните: это лишь советы, а не строгие

Copyrighted material

правила и со временем, по мере развития индустрии, они могут меняться и трансформироваться. Однако они могут по крайней мере задать направление мысли о сервисах и о том, что может быть сервисом.

- *Разделение по бизнес-требованиям*. Есть ли какие-нибудь конкретные бизнес-требования, подсказывающие место границы (финансовая отчетность, безопасность или юридическое регулирование)?
- *Разделение по зонам ответственности*. Если какая-либо команда владеет какой-то определенной функциональностью независимо от других (размещаясь в другом городе, на другом этаже или даже под управлением отдельного менеджера), возможно, граница проходит здесь?
- *Разделение по роду данных*. Возможно, в вашей системе какие-либо данные естественным образом отделяются от остальных? Если так, не перегрузит ли систему размещение их в другом дата-центре?
- *Разделение возможностей или информации*. Предоставляет ли какая-то часть приложения определенные возможности, используемые другими сервисами? Требуют ли эти возможности использования общих данных?

Давайте подробно рассмотрим каждую рекомендацию в отдельности.

Рекомендация 1. Разделение по бизнес-требованиям

В некоторых случаях место проведения границы продиктовано наличием определенных бизнес-требований. Это могут быть требования обеспечения безопасности, соблюдения законодательства и другие специфические бизнес-нужды.

Пример 12.1. Обработка платежей

Представим себе какую-либо систему, где принимаются платежи от клиентов через кредитные карты. Как нужно собирать, обрабатывать и хранить информацию о картах и осуществленных платежах?

Copyrighted material

Хорошим стратегическим решением для бизнеса было бы поместить обработку кредитной карты в сервис, обособленный от остальной части системы.

Отделение критически важной бизнес-логики, в том числе обработки платежей с кредитных карт, в отдельный сервис имеет следующие преимущества.

- *Нормативно-правовые требования*. Законодательство предъявляет значительно более строгие требования к хранению информации о кредитных картах, чем ко всем остальным данным, необходимым бизнесу. Выделение платежей в отдельный сервис упрощает задачу по обеспечению иных правил хранения информации.
- *Безопасность*. Возможно, по соображениям безопасности вам понадобятся дополнительные брандмауэры (межсетевые экраны) между этими серверами.
- *Валидация*. Может понадобиться проведение в производственной среде дополнительных проверок для подтверждения того, что безопасность данной функциональности обеспечивается более строго, чем в остальной части системы.
- *Ограничение доступа*. Поскольку доступ к секретной информации о платежах, в частности к данным кредитных карт, должен иметь строго ограниченный круг лиц, возможно, потребуется ограничить доступ к серверам с функциональностью. Как правило, все сотрудники организации доступа к таким системам не имеют.

Рекомендация 2. Разделение по зонам ответственности

Приложения становятся все более и более сложными, а с увеличением сложности растет и количество людей, задействованных в разработке, в том числе разного рода узких специалистов. Координация между командами становится все сложнее по мере увеличения количества разработчиков, команд и офисов.

Copyrighted material

Сервисы — это способ разделить между различными командами владение небольшими обособленными друг от друга модулями.

ОБЩАЯ РЕКОМЕНДАЦИЯ

Одним сервисом должна владеть и управлять одна команда, оптимальный размер которой — 3–8 разработчиков. Команда должна отвечать за все аспекты работы сервиса.

Таким образом вы снизите зависимость от взаимодействия между командами. Отдельной команде намного проще управлять отдельным сервисом и при необходимости развивать его.

КОМАНДА-ВЛАДЕЛЕЦ

Как уже было сказано, за управление каждым сервисом должна нести ответственность только одна команда, но в то же время команда может управлять несколькими сервисами. Самое главное, чтобы все аспекты работы одного сервиса находились в руках одной команды. Это значит, что команда отвечает за разработку, тестирование, развертывание, производительность и доступность данного сервиса.

Однако эта команда может вполне успешно управлять более чем одним сервисом в зависимости от объема и сложности действий, необходимых для этого. Кроме того, если несколько сервисов схожи между собой, очень удобно, когда они находятся в ведении одной команды.

Команда может владеть или управлять несколькими сервисами, но сервис не может находиться под управлением нескольких команд.

Разделение команд по соображениям безопасности

Порой представляется разумным ограничить доступ людей к коду и конфиденциальной информации, хранящимся на определенном сервисе. Особенно важно это для сервисов, имеющих определенные нормативно-правовые ограничения, как в примере 12.1. Ограни-

чение доступа к сервисам с секретной информацией может значительно снизить вероятность возникновения проблем с утечкой этих данных. Если это ваш случай, можно физически ограничить доступ к коду, данным и системам, хранящимся на сервисе, так, чтобы доступ к нему имели только сотрудники, непосредственно обеспечивающие работу сервиса.

Кроме того, разделение хранения конфиденциальных данных на несколько сервисов, за каждый из которых отвечает отдельная команда, в принципе снижает риск компрометации этих данных: маловероятно, что информация пострадает сразу на нескольких сервисах, которыми владеют отдельные команды.

Пример 12.2. Разделение данных по соображениям безопасности

В примере 12.1 номера кредитных карт могут храниться на одном сервисе, а остальная информация, необходимая для их использования (платежный адрес и код CCV), — на другом. Разделив эту информацию между двумя сервисами, каждый из которых находится в ведении отдельной команды, вы снижаете вероятность того, что какой-либо сотрудник умышленно или случайно допустит попадание этих данных к мошенникам.

Более того, можно не хранить номера кредитных карт на своих сервисах вообще, а передать эту задачу сервисам обработки платежных карт, предлагаемым другими компаниями. Таким образом, даже если безопасность одного из ваших сервисов будет нарушена, данные кредитных карт не пострадают.

Рекомендация 3. Разделение по роду данных

Одно из основных требований к сервисам заключается в том, что управляемое состояние сервиса и данные, хранящиеся на нем, должны быть отделены от других данных. По разным причинам проблематично иметь несколько независимых баз кода, каждая из которых управляет одним и тем же набором данных. Разделение кода и владения им

Copyrighted material

эффективно только в том случае, если вы одновременно разделяете и данные.

На рис. 12.1 можно видеть сервис А, пытающийся получить доступ к информации, хранящейся на сервисе Б. Это корректный путь получения данных сервисом А от сервиса Б: сервис А делает вызов через API к сервису Б, а затем сервис Б самостоятельно запрашивает для него в своей базе данных необходимую информацию.

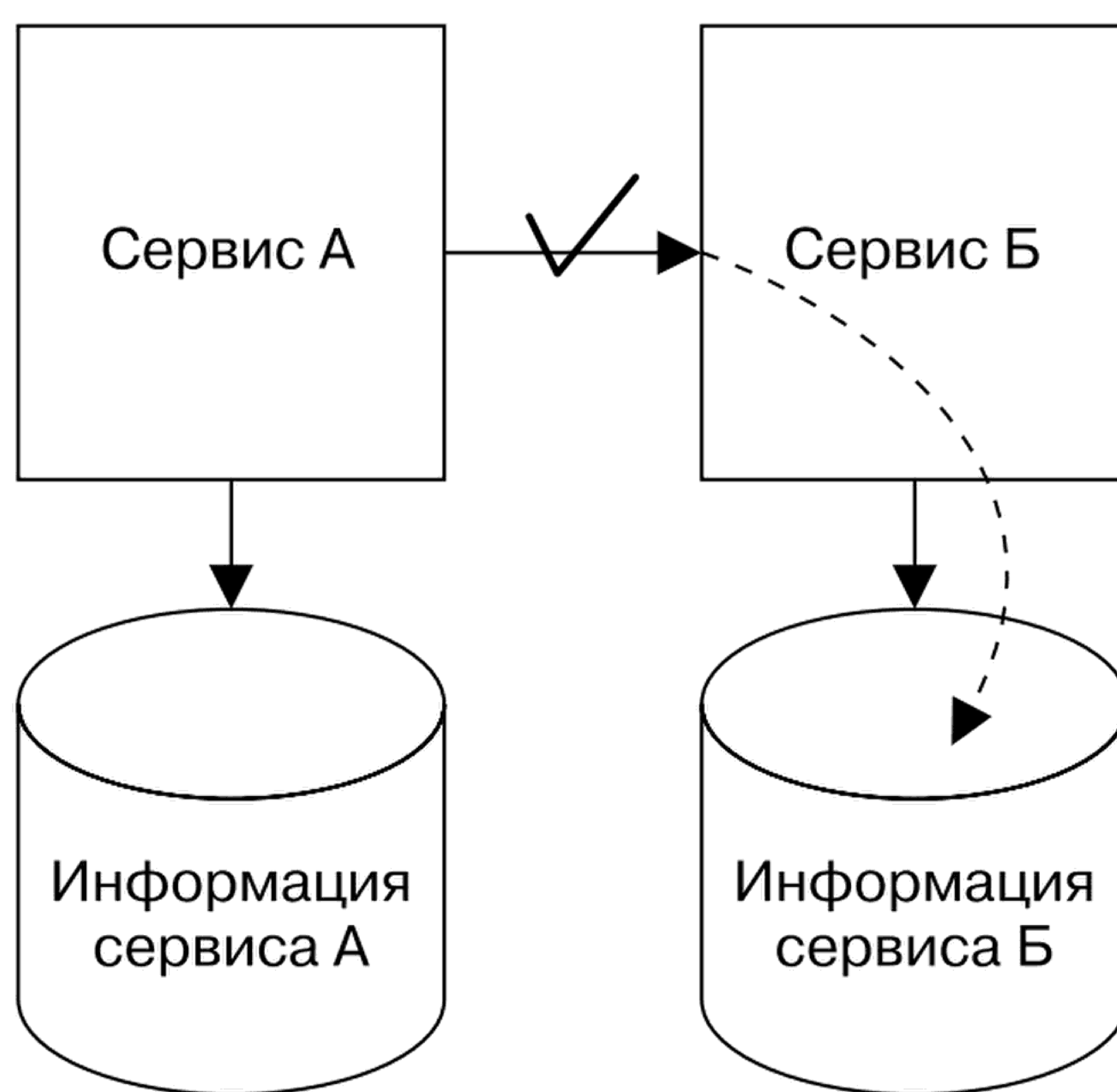


Рис. 12.1. Корректный способ получения информации

Если же вместо этого сервис А попытается напрямую получить данные сервиса Б без использования API последнего (рис. 12.2), могут возникнуть самые разные проблемы. Этот тип интеграции данных требует куда более тесной координации между сервисами А и Б, чем было бы разумно, и, кроме того, может в дальнейшем вызвать проблемы при миграции структуры базы данных и хранении информации. В целом, возможность прямого доступ сервиса А к данным сервиса Б без учета бизнес-логики последнего может вызвать серьезные проблемы, связанные с различием версий данных или искажением этих данных, поэтому нельзя этого допускать.

Как можно видеть, определение границы разделения данных играет важную роль в определении границы разделения сервисов. Должен ли данный сервис быть владельцем своих данных и предоставлять

доступ к ним только через внешние интерфейсы? Если да, то перед вами хорошая возможность провести границы сервиса. Если нет — лучше поискать другую.

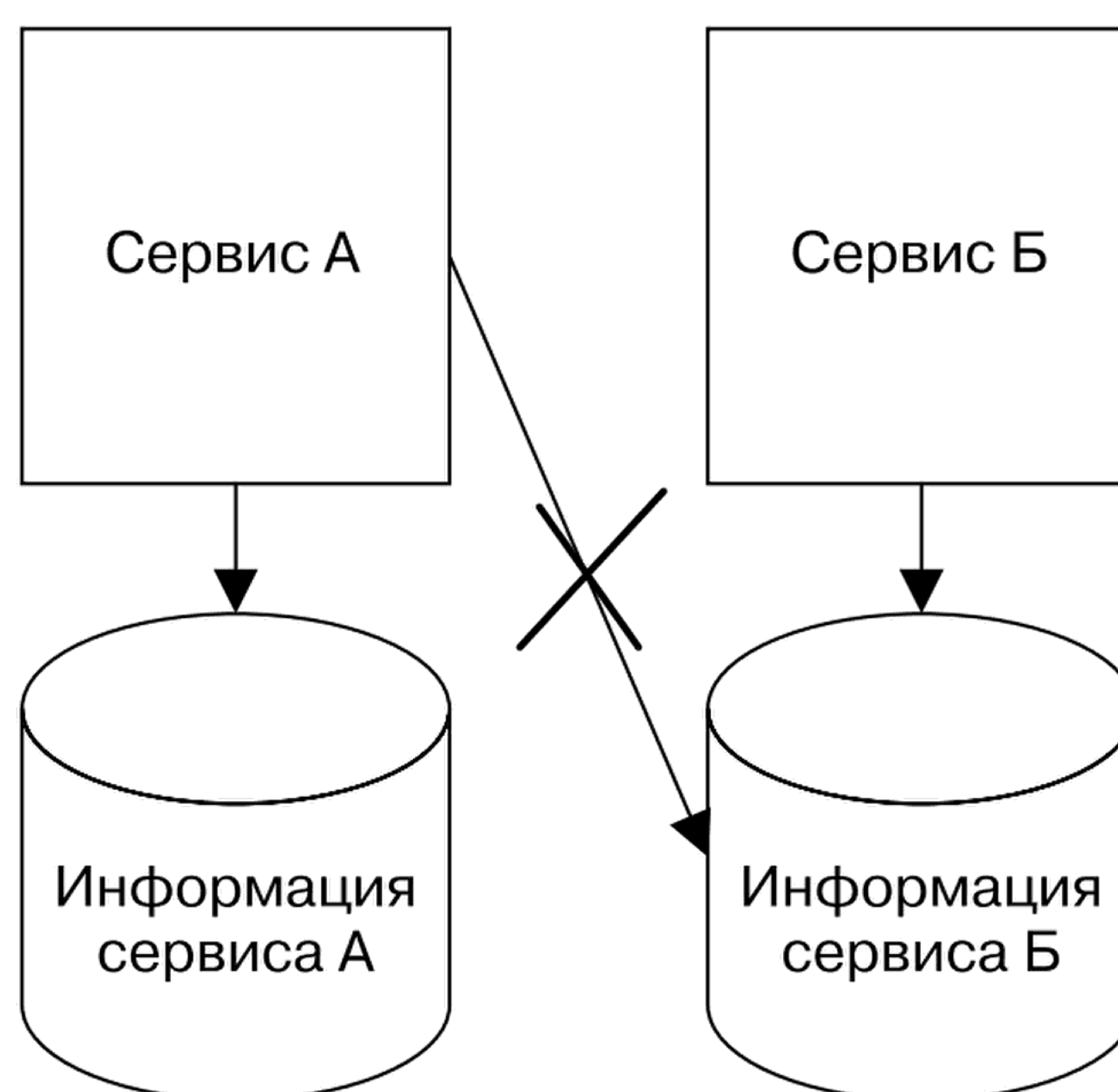


Рис. 12.2. Некорректный способ получения информации

Сервис, которому нужны данные, находящиеся у другого сервиса, должен получать их только через публичные интерфейсы, предоставляемые сервисом, являющимся владельцем данных.

Рекомендация 4. Разделение возможностей или информации

Иногда разумно создать сервис только потому, что он будет отвечать за определенный набор возможностей и данные, которые необходимы для работы нескольким другим сервисам. Хороший пример — сервис пользовательской идентификации, содержащий информацию обо всех пользователях вашей системы (рис. 12.3).

В работе сервиса по управлению данными не обязательно участвует сложная бизнес-логика, но он выполняет важнейшую задачу хранения общей информации, связанной со всеми пользователями системы. От этой информации зависит работа огромного количества других сервисов. Иметь централизованный сервис, ответственный

за управление и предоставление другим сервисам важной информации, очень полезно.

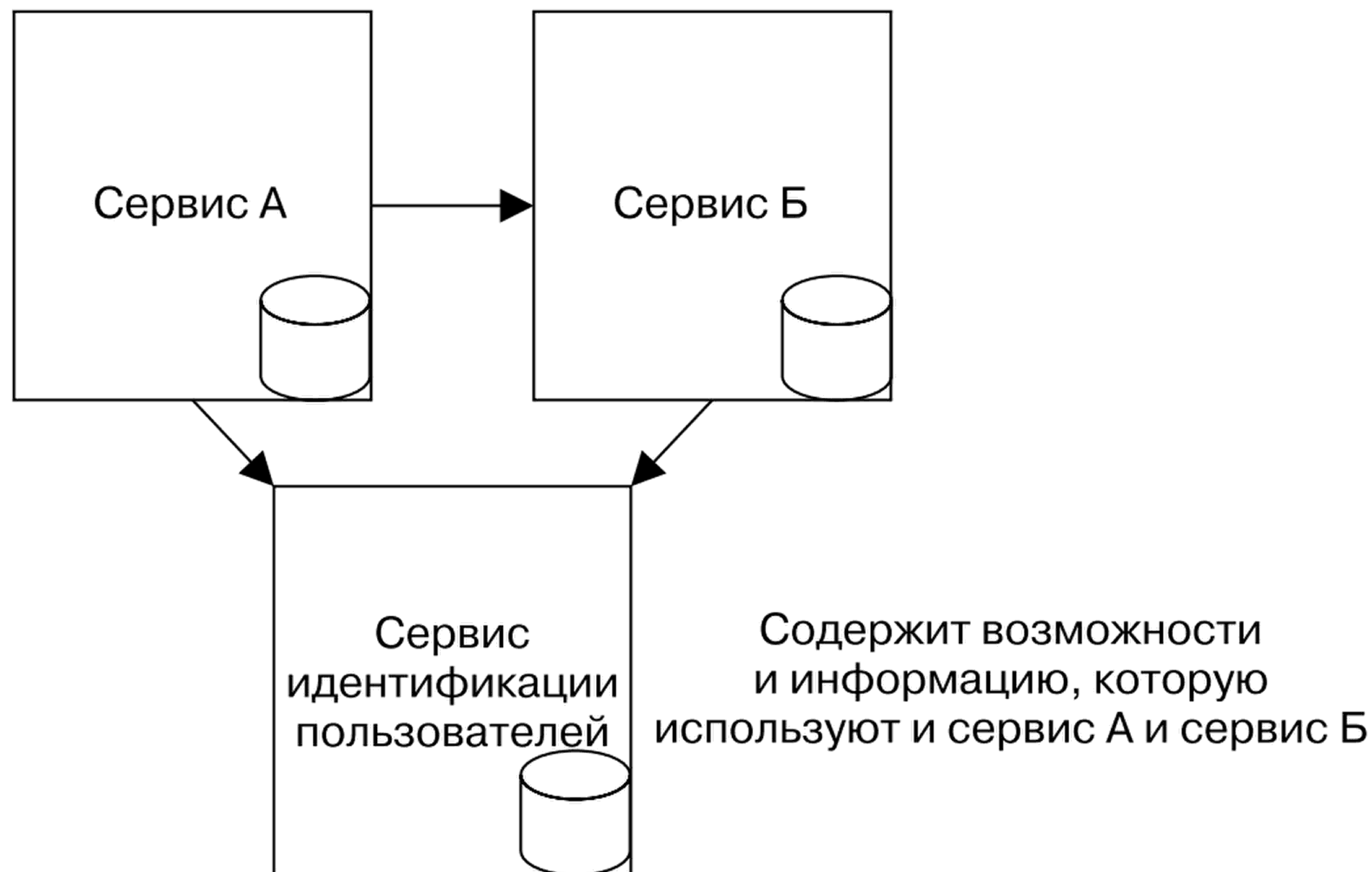


Рис. 12.3. Сервис для совместного использования данных другими сервисами

Сочетание критериев

В приведенных рекомендациях вы можете найти некоторые базовые критерии для выявления границ сервисов, но часто решающим фактором служит сочетание нескольких критериев. Например, выделение идентификации пользователей в отдельный сервис имеет смысл с точки зрения владения данными и разделения возможностей, но в перспективе командного управления это может быть неразумно. Возможно, следует хранить в отдельном сервисе или сервисах информацию, которая должна иметь в базе данных связи с идентификацией пользователей.

Хороший пример такой информации — поисковые предпочтения пользователя, которые, как правило, являются частью пользовательского профиля, но не используются нигде за пределами инфраструктуры поиска. В связи с этим имеет смысл хранить такие данные в сервисе поисковой идентификации, обособленном от сервиса пользовательской идентификации. Это решение будет

оправдано соображениями сложности информации или даже производительности[1].

Хоть и с учетом приведенных критериев, в конечном итоге вам придется полагаться на собственные решения. И конечно, следует учитывать бизнес-логику, специфические требования конкретной компании, а также особые нужды бизнеса.

Не переходите границ разумного

Порой, разделяя свое приложение на сервисы, можно увлечься и зайти слишком далеко. Не следует доводить до абсурда проведение границ сервисов согласно приведенным ранее критериям и создавать излишнее количество сервисов.

Допустим, вместо того, чтобы предоставить простой сервис пользовательской идентификации, вы решили разделить его на еще несколько более мелких сервисов. Например, так:

- сервис управления именами пользователей;
- сервис управления физическими адресами;
- сервис управления электронными адресами;
- сервис управления домашними адресами;
- сервис управления... и т. п.

Скорее всего, такое разделение на сервисы будет чрезмерным[2].

Разделение приложения на слишком большое количество сервисов может привести к нескольким проблемам, включая снижение производительности всего приложения. Главное, о чем следует помнить, разделяя какой-либо фрагмент функциональностей на несколько

---

[1] Нерационально хранить в сервисе пользовательской идентификации поисковые предпочтения, если они используются только в нескольких особых случаях.

[2] Хотя нет, слова «скорее всего» беру обратно — это определенно чрезмерное разделение на сервисы.

сервисов: в данный момент вы снижаете сложность отдельных сервисов (как правило) и повышаете сложность приложения в целом.

Как правило, сложность сервиса прямо пропорциональна размеру. Однако, чем больше у вас сервисов, тем больше сил приходится тратить на координацию их между собой и тем сложнее становится архитектура приложения в целом.

Наличие в системе чрезмерно большого количества сервисов может привести к появлению в вашем приложении следующих проблем.

- *Сложно понять общую картину.* Становится труднее удерживать в уме всю архитектуру приложения, так как оно оказывается все сложнее и сложнее.
- *Возникла благоприятная среда для сбоев и ошибок.* Чем больше независимых компонентов должны работать вместе, тем больше возможностей для появления ошибок и сбоев при их взаимодействии.
- *Сложно вносить изменения в сервисы.* Если от каждого сервиса в отдельности зависит работа большого количества других сервисов, то возрастает вероятность того, что изменения в одном сервисе негативно повлияют на работу остальных.
- *Возрастает количество зависимостей.* У каждого сервиса оказывается все больше зависимостей от других сервисов. А чем больше зависимостей, тем больше ошибок может возникнуть при взаимодействии.

Большинство этих проблем решается тщательной проработкой граничных интерфейсов взаимодействий между сервисами, но полностью избавиться от них таким образом нельзя.

Соблюдение баланса

Как видите, выбор необходимого количества сервисов и определение верного размера каждого из них — нелегкая задача: приходится балансировать между преимуществами создания большого количества сервисов и недостатками, вызванными возрастанием сложности системы в целом.

Copyrighted material

Если сервисов окажется слишком мало, им будут свойственны практически все недостатки монолитного приложения: над одним сервисом приходится работать многим разработчикам, а сами сервисы оказываются слишком сложными.

Если же их окажется слишком много, то отдельные сервисы будут очень простыми, а вот приложение в целом и взаимодействие между сервисами станут чрезмерно сложными. Я слышал об экспериментальном приложении, где среди сервисов были микросервисы «Да» и «Нет». Они не делали ничего, кроме возвращения этих булевых результатов. Это, разумеется, совершенно экстремальный пример. Тем не менее, как бы ни было здорово определить единственный *верный* размер сервиса, он всегда будет зависеть от специфики вашего приложения и вашей компании. Поэтому всегда нужно помнить о балансе сложности приложения и сервисов при определении сервисов в архитектуре.

Выявив этот баланс именно для своего приложения, своей организации и ее особенностей, вы сможете извлечь максимальную выгоду из основанной на сервисах среды.

Copyrighted material

13 Обработка отказов сервисов

Одна из опасностей, подстерегающих вас при построении крупного приложения, основанного на микросервисах, — необходимость обработки отказов их работы. Чем больше у вас сервисов, тем выше вероятность того, что какой-нибудь из них выйдет из строя, и тем больше будет отказавших сервисов, чья работа зависит от поврежденного.

Каскадные аварии сервисов

Допустим, вы владелец какого-либо сервиса. Он зависит от одних сервисов, а другие зависят от него. На рис. 13.1 изображен наш сервис с несколькими зависимостями от сервисов А, Б и В, а также несколькими сервисами, чья работа зависит от него (потребитель 1 и потребитель 2).

Рис. 13.1. Наш сервис, его зависимости и потребители

Copyrighted material

Что произойдет, если одна из этих зависимостей перестанет работать? На рис. 13.2 проиллюстрирован выход из строя сервиса А.

Рис. 13.2. Наш сервис, одна из зависимостей которого не работает

Стоит не уследить, и наш сервис перестанет работать, а это, в свою очередь, приведет к выходу из строя потребителей 1 и 2. Таким образом, авария станет каскадной (рис. 13.3).

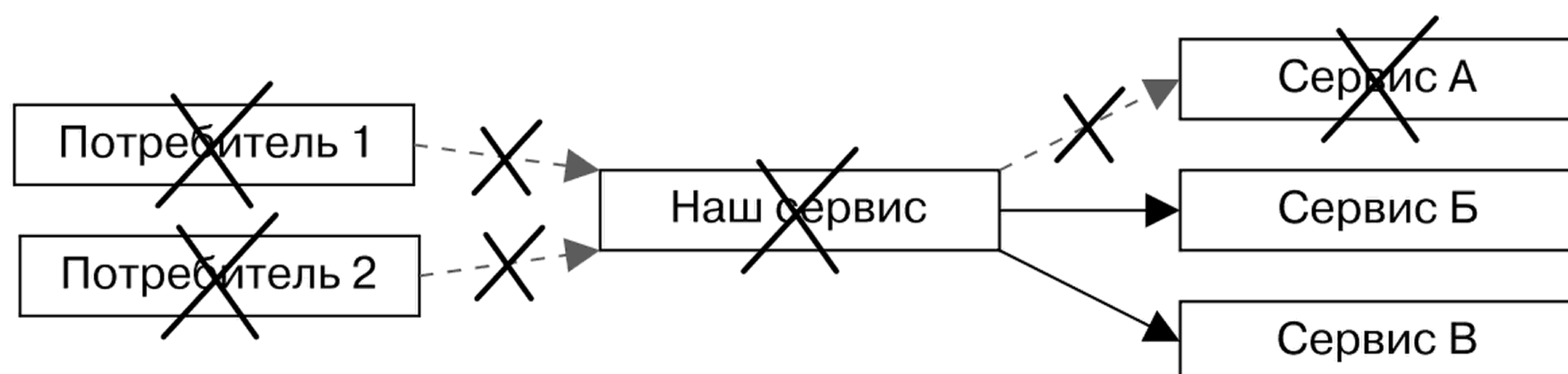


Рис. 13.3. Каскадная авария

Вот так отказ одного сервиса в системе может вызвать серьезнейшие проблемы во всем приложении.

Что можно предпринять для предотвращения такого рода каскадных аварий? Иной раз — ничего, так как выход из строя какой-либо зависимости автоматически ведет к отказу вашего сервиса (и тех, кто зависит от вас), так как эта зависимость жизненно важна. Иногда ваш сервис просто прекращает работу, если нет нужной зависимости.

Но, к счастью, так получается не всегда. На практике, как правило, есть масса возможностей сохранить работоспособность сервиса в случае отказа зависимости. В этой главе мы обсудим несколько техник.

Copyrighted material

Реагирование на отказ сервиса

Что нужно предпринять, если сервис, от работы которого вы зависите, выходит из строя? Так как вы разработчик сервиса, ваша реакция на отказ зависимости должна быть:

- предсказуемой;
- понятной;
- адекватной.

Давайте рассмотрим эти характеристики по одной.

Предсказуемая реакция

Для успешного создания зависимости одних сервисов от других очень важно наличие предсказуемой реакции. Кроме того, она должна соответствовать конкретным обстоятельствам и запросам. Только так можно избежать негативного влияния на все аспекты вашего приложения при каскадных авариях сервисов. Даже тривиальный поначалу сбой при каскадном развитии может стать большой проблемой, если вы не подготовились как следует.

Таким образом, даже если одна из нисходящих зависимостей вашего сервиса дала сбой, вы все еще отвечаете за генерацию предсказуемого ответа, только теперь это должно быть сообщение об ошибке. Это совершенно приемлемая реакция, если в вашем API есть вероятность генерации ошибочного ответа.

Ответ, сообщающий об ошибке, — не то же самое, что непредсказуемая реакция. Непредсказуемая реакция означает, что она непонятна сервисам, которые вы обслуживаете. А информация об ошибке — это корректная реакция, означающая, что вы оказались неспособны обработать конкретный запрос. Это две разные вещи.

Если ваш сервис получил запрос на выполнение операции 3 + 5, ожидается, что в результате он вернет какое-то число, в частности 8. Это предсказуемая реакция. Если сервис получил запрос на операцию 5 / 0, предсказуемым ответом будет

требованиям этого контракта, даже если ваши собственные зависимости работают неверно. Совершенно неприемлемо нарушать соглашение между вами и потребителями только по той причине, что нисходящие зависимости нарушают их контракт с вами. Напротив, следует позаботиться, чтобы в таком согласованном интерфейсе были предусмотрены все неожиданные случаи на вашей стороне, включая нарушение работы зависимостей.

Адекватная реакция

Ваша реакция должна соответствовать тому, что приключилось с сервисом. Если вы получили запрос «Сколько будет 3 + 5?», то не должны возвращать в качестве ответа «Красный», даже если все зависимости не работают. Приемлемыми в таких ситуациях могут быть ответы «К сожалению, вычислить результат не удалось» или «Пожалуйста, попробуйте снова», но ни в коем случае не «Красный».

К чему может привести неадекватная реакция API

Да, это звучит банально, но, возможно, вас удивит количество ситуаций, когда не соответствующая ситуации реакция может вызвать проблемы. Представьте, например, что сервис хочет получить список всех аккаунтов с истекшим сроком действия, которые можно удалить. Как показано на рис. 13.4, вы можете обратиться к сервису просроченных аккаунтов, который вернет вам список всех аккаунтов, готовых к удалению, а затем удалить все эти аккаунты.

Если сервис просроченных аккаунтов по какой-то причине не может сгенерировать корректный ответ, он должен вернуть отсутствие результатов или сообщение «К сожалению, выполнить запрос не удалось».

Представьте, что было бы, если бы вместо адекватного ответа сервис вернул список всех аккаунтов в системе! Это точно не прошло бы незамеченным.

Рис. 13.4. Неадекватная реакция API

Выявление сбоев

Как определить, что с зависимостью произошел сбой? Это зависит от вида сбоя. Вот несколько примеров сбоев, о которых не стоит забывать.

- *Бесполезный результат.* Ответ может быть непонятным и негодным для дальнейшей работы: в нем содержатся непонятные данные в нераспознаваемом формате. Это может говорить о синтаксических ошибках в формате ответа или неверно выбранном формате.

- *Результат при возникновении фатальной ошибки*. Ответ в этом случае понятен, он означает, что при обработке запроса возникла серьезная ошибка. Как правило, это говорит о сбое не при коммуникации, а на самом сервисе. Возможно также, что отправленный сервису запрос оказался непонятным.

- *Результат понятен, но не соответствует необходимому*. Из ответа можно понять, что операция была успешно выполнена без серьезных ошибок, но возвращенные данные не соответствуют ожидаемым.

- *Результат выходит за границы допустимого*. Из ответа можно понять, что операция была успешно выполнена без серьезных ошибок. Возвращенные данные адекватны и получены в нужном формате, но их значения выходят за пределы допустимого. Например, вы отправили сервису запрос о количестве дней, минувших с начала года, а получили в ответ, например, число 843. Этот результат понятен, формально годен для дальнейшей работы и сбоев при его получении не обнаружено, но совершенно ясно, что он находится вне допустимого диапазона.

- *Ответ не получен*. Запрос был отправлен, но никакого ответа на сервис не пришло. Это могло случиться из-за проблем в сетевой коммуникации, сбоя на сервисе или выхода сервиса из строя.

- *Ответ пришел с задержкой*. Запрос был отправлен, а ответ — получен. Ответ оказался полезным, адекватным и находится в допустимом диапазоне, но пришел он позже, чем ожидалось. Как правило, это говорит о перегрузке либо сервиса, либо сети или же о поглощении ресурсов сервиса из-за другой проблемы.

Проблемы перечислены здесь, начиная от самой легкой в обнаружении до самой сложной. Получив ответ, негодный к использованию, вы сразу понимаете, что работать с ним нельзя, и можете предпринять соответствующие действия. Заметить, что результат запроса не соответствует ожидаемому, может быть, чуть сложнее, как и определить необходимые в этом случае действия, но все-таки позаботиться об этом можно.

Если вы не получили ответа вообще, сложно понять, когда и как нужно реагировать, потому что нет никакого стимула для вашей реакции. Но если все, что вы можете сделать, — это сгенерировать сообщение об ошибке для сервисов-потребителей, простой тайм-аут при операциях с зависимостями может помочь в обнаружении отсутствия ответа.

КАК ОТЛОВИТЬ ОТВЕТЫ, КОТОРЫХ НЕТ?

Это не рецепт на все случаи жизни. Например, что вы будете делать, если, как правило, сервису требуется 50 мс на ответ, но в разных случаях это время может колебаться от 10 до 500 мс? На какое время установить тайм-аут? Ожидаемый ответ — на любое, превышающее 500 мс. Но что делать, если по контракту с сервисом-потребителем вы обязаны отреагировать в течение 150 мс? Так что простая установка тайм-аута 500 мс — не лучшее решение, потому что вы просто перекладываете решение своей проблемы с зависимостью на плечи сервиса-потребителя, что нарушает требования предсказуемости и понятности.

Как можно решить эту проблему? Один из возможных ответов — использование шаблона прерывания цепи. При таком шаблоне написания кода ваш сервис постоянно отслеживает обращения к своим зависимостям и запоминает, сколько их завершились успешно, а сколько — ошибочно (или по тайм-ауту). Если достигнут определенный уровень сбоев, прерыватель размыкает цепь и заставляет ваш сервис предположить, что зависимость не работает. В результате сервис прекращает посылать запросы или ожидать ответы. Таким образом он может немедленно выявить ошибку и предпринять соответствующие действия, выполнив требования к скорости ответа сервису-потребителю согласно SLA между ними.

Затем можно периодически проверять состояние зависимости, посылая запросы к другому сервису. Если он начинает успешно на них отвечать (то есть уровень сбоев падает), прерыватель восстанавливает цепь и ваш сервис может возобновить работу с зависимостью.

Если ответа сервиса приходится ждать слишком долго (в отличие от случая, когда он не приходит вообще), это, пожалуй, обнаружить сложнее всего. Прежде всего, как отличить «долго» от «слишком долго»? Однозначного ответа дать нельзя, и простого использования базовых тайм-аутов (с прерывателями цепи или без них), как правило, недостаточно для эффективной обработки ситуации. Иногда долгое ожидание ответа может быть достаточно скорым, чтобы сгенерировать нестабильные результаты. Помните: *предсказуемость* ответа — важнейшая характеристика вашего сервиса и зависимость, которая произвольным образом сбоит (из-за медленных ответов или неверно выбранных тайм-аутов), подрывает вашу способность дать предсказуемый ответ тем, кто зависит от вас.

ВЫЯВЛЕНИЕ ЗАДЕРЖЕК В ЗАВИСИМОСТЯХ: ВЕРСИЯ ДЛЯ ПРОДВИНУТЫХ

Может оказаться полезным более продвинутый механизм тайм-аутов в сочетании с прерыванием цепи и похожими шаблонами. Например, вы можете создать специальные сегменты памяти для хранения последних данных о производительности вызовов по данной зависимости. Каждый раз, когда вы обращаетесь к зависимости, в таком сегменте сохраняется информация о вызове и о том, сколько времени потребовалось для получения ответа. Результаты сохраняются в сегменте только для заданного временного периода, а затем используются для формулирования правил запуска прерывания цепи. Эти правила могут выглядеть, например, так:

- если вы получили 500 запросов в минуту, которые заняли дольше 150 мс, запускается разрыв цепи;
- если вы получили 50 запросов в минуту, которые заняли дольше 500 мс, запускается разрыв цепи;
- если вы получили пять запросов в минуту, которые заняли дольше 1000 мс, запускается разрыв цепи.

Такая многоуровневая техника позволит вам раньше отлавливать более серьезные задержки, не игнорируя при этом менее серьезные.

Соответствующие действия

Что предпринять, если произошла ошибка? Это зависит от ее типа. Далее вы найдете несколько известных шаблонов, которые могут помочь вам обрабатывать ошибки различных типов.

Постепенная деградация

Если зависимость вашего сервиса вышла из строя, может ли сервис продолжать работу без нее? Можно ли выполнять свои задачи в отсутствие ответов от поврежденного сервиса? Если ваш сервис может хотя бы частично выполнять то, что от него требуется, не получая ответов на свои запросы к поврежденному сервису, это и есть пример *постепенной деградации*.

Постепенная деградация означает, что сервис старается выполнить наибольший объем работы, возможный при отсутствии необходимых результатов деятельности поврежденного сервиса.

Пример 13.1. Сниженная функциональность

Представим себе веб-приложение, генерирующее коммерческий сайт, где продаются футболки. Допустим, что там есть сервис, предоставляющий URL для изображений, которые будут показаны на сайте. Если приложение вызывает сервис изображений, но тот оказывается поврежден, что может сделать приложение? Один из вариантов — приложение продолжает отображать для пользователя запрошенный продукт, но без изображений (или с сообщениями «Изображение недоступно»). Таким образом, приложение продолжает функционировать как интернет-магазин, но без возможности показать изображения продуктов.

Это все-таки лучше, чем полная неработоспособность интернет-магазина с отображением для пользователя сообщения об ошибке по причине недоступности сервиса изображений.

В данном примере мы наблюдаем *снижение функциональности*.

Очень важно, чтобы сервис (или приложение) продолжал функционировать с максимально возможной пользой, даже если в его распоряжении нет всех необходимых данных из-за поврежденной зависимости.

Постепенный откат

Естественно задуматься, что же будет, если результатов, необходимых для продолжения работы, не хватает. Запрос просто даст сбой. Но все же можете ли вы сделать хоть что-то полезное для своих потребителей вместо простой отправки сообщения об ошибке?

Пример 13.2. Постепенный откат

Продолжая обсуждение ситуации, описанной в примере 13.1, предположим, что сервис, предоставлявший все детали данного продукта, вышел из строя. Это значит, что сайт не может отобразить вообще никакой информации о запрошенном продукте. Разумеется, показывать пустую страницу не стоит, в этом нет никакой пользы для клиентов. Отображение сообщения «К сожалению, произошла ошибка» — тоже не лучшая идея.

Вместо этого вы можете отобразить страницу с извинениями за проблему и ссылками на самые популярные продукты, доступные на сайте. Разумеется, это не то, чего хотел покупатель, но все-таки он получит определенную пользу и по крайней мере не увидит простую страницу с ошибкой.

Если вы вообще не можете выполнить запрос, следует перейти от изначальной задачи к тому, что будет хоть чем-то полезно потребителю. Это и есть пример *постепенного отката*.

Сдавайтесь как можно раньше

А если ваш сервис не может функционировать без ответа поврежденного сервиса? Если нет вариантов снижения функциональности или постепенного отката? Не получив результатов от поврежденного сервиса, вы не сможете сделать ничего полезного. В этом случае остается только отказать в выполнении запроса.

Если вы обнаружили, что нет никакого способа хоть как-то выполнить запрос, тут же прекращайте его обработку. Не стоит тратить время на выполнение каких-то других действий или задач, связанных с этим запросом.

Copyrighted material

Естественный вывод из этого правила: надо выполнять как можно больше проверок поступающих запросов и делать это как можно раньше. Таким образом вы удостоверитесь, что запрос по крайней мере возможно выполнить при дальнейших действиях.

Представим себе сервис, который получает два числа, а затем делит одно на другое. Как вы знаете, деление на ноль невозможно. Поэтому, получив запрос, к примеру, 3 / 0 и попытавшись вычислить его, рано или поздно вы обнаружите, что получить результат невозможно и в любом случае придется сообщить об ошибке.

Поэтому, зная, что деление на ноль в любом случае не даст результата, просто сразу же проверяйте все данные, поступающие с запросом. Если делитель равен нулю, немедленно возвращайте ошибку, так как все дальнейшие попытки выполнить вычисление будут бесполезными.

Почему же так важно сдаваться как можно раньше? Вот несколько причин.

- *Экономия ресурсов*. Если запрос в итоге не будет выполнен, все предварительная работа, проделанная перед обнаружением ошибочности запроса, окажется напрасной. Если в числе прочего вы сделали много вызовов к другим сервисам, то может оказаться, что потрачено значительное количество ресурсов, а в результате получена ошибка.
- *Быстрое реагирование*. Чем раньше вы определите, что запрос не может быть обработан, тем быстрее отправите этот результат обратно. Таким образом, тот, кто отправил вам этот запрос, сможет скорее предпринять дальнейшие действия и принять необходимые решения.
- *Сложность ошибки*. Иногда, если вы позволите ошибочному запросу продвинуться дальше, выявление ошибочности или выяснение причин ошибки может усложниться. Рассмотрим пример с делением 3 на 0. Вы могли бы немедленно обнаружить, что вычисление результата невозможно, и вернуть соответствующий

100 000 не входило в допустимый диапазон входных данных для этого запроса.

Однако сервис аккаунтов все-таки продолжал попытки выполнить этот запрос... продолжал и продолжал...

Эти действия изначально были обречены на неудачу, так как у сервиса просто не было достаточных ресурсов для выполнения такого огромного запроса. Обработав первые несколько тысяч аккаунтов, он прекратил работу и выдал простое сообщение об ошибке.

Вызывающий сервис, тот, который сгенерировал некорректный запрос, получил сообщение об ошибке и решил, что должен повторить запрос. И повторял его снова и снова.

Сервис аккаунтов раз за разом обрабатывал тысячи аккаунтов только для того, чтобы выбросить эти результаты прочь и выдать сообщение об ошибке. Это происходило снова и снова. Повторяющиеся некорректные запросы потребляли огромное количество доступных ресурсов — так много, что даже корректные запросы к сервису не могли быть обработаны, становились в очередь и получали немедленный отказ.

Простейшая проверка поступившего запроса на сервисе аккаунтов (скажем, попадание запрошенного количества аккаунтов в допустимый диапазон) предотвратила бы этот огромнейший и совершенно бесполезный расход ресурсов. Кроме того, если бы в сообщении об ошибке при этом указывалось, что ошибка является перманентной и вызвана неправильным аргументом, то вызывающий сервис прекратил бы повторные попытки, изначально обреченные на неудачу.

Установите ограничения сервиса

Естественный вывод из этой истории: всегда устанавливайте ограничения в соответствии с возможностями сервиса. Если вы знаете, что ваш сервис не может обработать, скажем, больше 5000 аккаунтов за раз, установите это ограничение в сервисном контракте, а затем проверяйте все запросы и отказывайте тем, чьи входные данные превышают этот предел.

Copyrighted material

Часть IV
Масштабирование приложений

Даже улитку можно масштабировать.

Copyrighted material

14 Запас на две ошибки

Вроде бы все идет хорошо...

Хочу поделиться с вами услышанной однажды историей.

Мы хотели узнать, как изменение одного из параметров базы данных MySQL повлияет на производительность, но беспокоились, как бы этот эксперимент не привел к выходу из строя продуктовой базы данных. В то же время мы не хотели получить аварию в производственной среде, поэтому решили сначала внедрить изменение на резервной базе данных, являющейся точной копией продуктовой. В тот момент она была не нужна...

Звучит разумно, не правда ли? Вы и сами наверняка слышали такие рациональные аргументы.

Проблема в том, что на самом деле резервная база данных *была нужна* для кое-чего. Ее назначение — хранить и при необходимости предоставлять точную копию данных производственной среды. Если эта задача не выполняется, резервная база бесполезна.

Итак, резервная база данных была использована для экспериментов с различными значениями настроек. К чему это привело? К тому, что она стала все больше *отличаться* от основной продуктовой базы, так как ее настройки постоянно менялись.

Copyrighted material

В конце концов случилось неизбежное: продуктовая база данных вышла из строя. Резервная база данных сделала то, для чего она изначально предназначалась: заступила на место продуктовой и попыталась выполнить ее задачу, но ей это не удалось. Настройки резервной базы данных оказались настолько отличающимися от настроек продуктовой, что она была не способна надежно обрабатывать тот же объем трафика, что обрабатывала основная база.

Резервная база данных вскоре вышла из строя, и сайт прекратил работу.

Это реальная история, повествующая о самых лучших намерениях. Была резервная база данных — точная копия продуктовой. Но из-за того, что к резервной базе не относились с той же осторожностью, что и к основной, она утратила возможность выполнить свое основное предназначение — заместить основную базу данных в случае выхода из строя последней.

Два минуса не дают плюс, две ошибки не компенсируют друг друга, и две проблемы не решат друг друга сами. Выход из строя основной базы данных в сочетании с плохим управлением резервным сервером не приведет ни к чему хорошему.

Что такое запас на две ошибки?

Если вы когда-нибудь запускали радиоуправляемые самолетики, то, возможно, слышали выражение «Держи свой самолет на две ошибки выше».

Обучаясь обращению с радиоуправляемым самолетом и особенно выполнению акробатических трюков, вы очень быстро понимаете смысл этого выражения. Высота зависит от ошибок. Вы делаете ошибку — самолет теряет высоту. Если самолет находится слишком низко, может случиться беда. Поэтому удерживать его на две ошибки выше означает, что высота должна позволять выправить самолет после двух независимых ошибок.

Почему именно двух ошибок? Все просто. Вы хотите управлять своим самолетом на такой высоте, чтобы можно было выправить его

положение, если (или когда) сделаете ошибку. Представьте теперь, что вы сделали ошибку и потеряли высоту. Во время исправления ситуации вы также хотите быть достаточно высоко, чтобы можно было выровняться после еще одной ошибки. Подумайте: во время восстановления ситуации вы, как правило, нервничаете и ведете себя нетипично, в этом случае очень легко сделать еще одну ошибку. В результате, находясь недостаточно высоко, самолет может разбиться.

В то же время, летая на две ошибки выше, вы всегда имеете запасной план восстановления после ошибки, даже если прямо сейчас восстанавливаетесь после еще одной ошибки. Точно такой же подход очень важен при создании высокодоступных крупномасштабных приложений.

Как можно создать запас на две ошибки в приложении? Для начала нужно выявить аварийные сценарии, с которыми может столкнуться ваше приложение, а затем подробно рассмотреть их и разработать планы восстановления. Затем убедиться, что планы восстановления не содержат ошибок или других дефектов, — иначе говоря, проверить их работоспособность. Если выяснится, что план восстановления не работает, — значит, плана восстановления у нас нет.

Запас на две ошибки на практике

Это был лишь один потенциальный сценарий применения принципа двух ошибок (на самом деле их куда больше). Давайте рассмотрим несколько примеров, чтобы увидеть, какую роль играет этот принцип в наших приложениях.

Потеря одного из узлов

Вот пример сценария, где трафик обрабатывается через веб-сервис.

Пример 14.1. Сколько требуется узлов?

Допустим, вы работаете над сервисом, который должен обрабатывать 1000 запросов в секунду (з./с). Давайте предположим также, что единичный узел нашего сервиса может обрабатывать 300 з./с.

Copyrighted material

Сколько узлов нужно, чтобы покрыть потребности трафика?

Простая арифметика подсказывает нам следующее:

$$\text{Необходимое количество узлов} = \frac{\text{Общее количество запросов}}{\text{Количество запросов на узел}},$$

где *необходимое количество узлов* — количество узлов, необходимое для обработки указанного количества запросов; *общее количество запросов* — максимальное количество запросов к сервису, предусмотренное проектом; *количество запросов на узел* — ожидаемое среднее количество запросов к каждому узлу, которое тот сможет обработать.

Подставим числа:

$$\text{Необходимое количество узлов} = \frac{1000\ \text{з./с}}{300\ \text{з./с}} = 3{,}3 = 4\ \text{узла}.$$

Необходимое количество узлов равно 4.

Вам нужны четыре узла в сервисе для обработки 1000 з./с ожидаемой нагрузки. Используя обратный расчет, увидим, что каждый узел может обработать:

$$\text{Запросов на узел} = \frac{\text{Количество запросов}}{\text{Количество узлов}} = \frac{1000\ \text{з./с}}{4\ \text{узла}} = 250\ \text{з./с/узел}.$$

Каждый узел будет обрабатывать 250 з./с, что оставляет нам хороший запас с учетом предела допустимой нагрузки на узел 300 з./с.

В вашей системе четыре узла. Вы можете обработать ожидаемый трафик, не опасаясь потери одного из них. Вы предусмотрели возможность обработки выхода из строя узла. Точно? *Точно???*

На самом деле, конечно, нет. Если вы потеряете узел на пике трафика, сервис в конце концов выйдет из строя. Почему? Да потому, что в случае потери одного узла оставшиеся три вынуждены будут

$$\text{Запросов на узел} = \frac{\text{Количество запросов}}{\text{Количество узлов}} = \frac{1000\ \text{з./с}}{4\ \text{узла}} = 250\ \text{з./с/узел.}$$

Поскольку эта величина меньше допустимой нагрузки одного узла (300 з./с), мощности все еще достаточно для обработки всего трафика даже в случае потери одного из узлов.

Проблемы во время обновлений

Обновления и плановое обслуживание могут привести к довольно необычным проблемам. Давайте рассмотрим пример 14.4.

Пример 14.4. Обновление приложения

Предположим, у вас есть сервис со средним трафиком 1000 з./с. Допустим также, что каждый узел в сервисе может обрабатывать 300 з./с трафика. Как мы обсудили в примере 14.1, четыре узла — необходимый минимум для работы сервиса. А чтобы обработать ожидаемый трафик и выдержать нагрузку в случае потери одного из узлов, надо установить пять.

А сейчас предположим, что вы выполняете обновление программного обеспечения сервиса, запущенного на этих узлах. Чтобы сохранить полную работоспособность сервиса во время обновления, вы решаете выполнить *последовательное развертывание*.

Попросту говоря, последовательное развертывание означает, что вы обновляете один узел за раз, временно останавливая его, чтобы выполнить обновление. После того как первый узел успешно обновлен и снова может обрабатывать трафик, вы переходите ко второму узлу, тоже на время прерывая его работу. Вы повторяете процедуру, пока все пять узлов не будут обновлены.

Поскольку в любой момент времени из строя выводится только один узел, у вас всегда есть как минимум четыре узла, обрабатывающих трафик. Поскольку четырех узлов достаточно для обработки всего

трафика, сервис сохраняет работоспособность на все время обновления.

Прекрасный план! Вы создали систему, которая не только способна пережить потерю одного из узлов, но и может быть обновлена последовательно, без необходимости отключения. Но что произойдет, если один из узлов выйдет из строя *во время* обновления? Получится, что на одном из узлов сбой, а другой остановлен и обновляется. Остается только три узла для обработки всего трафика, чего, как мы знаем, недостаточно. В результате сервис полностью или частично прекращает работу.

Минутку: а какова вероятность того, что узел выйдет из строя именно во время обновления?

Сколько раз обновление заканчивалось неудачно? Фактически можно утверждать, что узлы *чаще* отказывают во время обновлений, чем в любое другое время. Обновление и отказы узлов — отнюдь не строго независимые события.

Какой вывод можно сделать? Даже если вы думаете, что у вас достаточный запас надежности для обработки различных видов отказов, но при этом есть вероятность, что возникнут одновременно несколько проблем, потому что они связаны друг с другом, это означает, что никакого запаса нет, а доступность под угрозой.

Так сколько же узлов требуется для системы из примера 14.1, чтобы обработать 1000 з./с, притом что каждый узел может обработать 300 з./с?

- *Четыре узла*. Могут обработать весь трафик, но при отказе одного узла система не работает.
- *Пять узлов*. Система работает при выходе из строя одного узла или его остановке для планового обслуживания или обновления.
- *Шесть узлов*. Система работает при выходе из строя двух узлов или отказе одного узла во время плановой остановки другого для обслуживания или обновления.

Отказоустойчивость дата-центра

Давайте немного масштабируем проблему и рассмотрим отказоустойчивость и избыточность дата-центра.

Пример 14.5. Крупный сервис

Давайте предположим, что теперь ваш сервис обрабатывает 10 000 з./с. При том, что каждый узел по-прежнему может обработать 300 з./с, для работы нужны 34 узла без учета запаса на отказы и плановое обслуживание.

Попробуем обеспечить отказоустойчивость и задействуем всего 40 узлов (каждый из которых будет обрабатывать 250 з./с), что обеспечивает избыток мощности. Можно потерять шесть узлов и все еще сохранять полную работоспособность.

Можно зайти дальше и разделить эти 40 узлов между четырьмя дата-центрами для обеспечения дополнительной надежности. Теперь отказы дата-центров или узлов нам не страшны.

Точно?

Хороший вопрос. Да, мы можем пережить отказы отдельных узлов, так как заранее запаслись шестью (40 – 34 = 6) дополнительными. А что, если прекратит работу дата-центр целиком?

Если отдельный дата-центр выйдет из строя, мы потеряем четверть своих серверов. В этом случае у нас окажется 30 узлов вместо 40. Теперь каждому узлу придется обрабатывать не 250, а 334 з./с. Поскольку это значение превышает допустимую мощность рассматриваемого узла, возникают проблемы с доступностью.

Хотя мы используем несколько дата-центров, выход из строя хотя бы одного из них означает, что нет возможности обработать возросший трафик. Мы думали, что устойчивы к потере дата-центра, а оказалось, что нет.

Так сколько серверов на самом деле нужно?

Какое количество серверов позволит нам пережить потерю одного дата-центра? Давайте разберемся.

Copyrighted material

Мы знаем, что при тех же условиях, что и в примере 14.5, для обработки всего трафика необходимы как минимум 34 работающих сервера. Если мы будем использовать четыре дата-центра, сколько серверов понадобится для надежного обеспечения отказоустойчивости дата-центров?

Получается, что требуется всегда иметь хотя бы 34 работающих сервера, даже если один из четырех дата-центров выходит из строя. Это значит, что нужно иметь 34 сервера, разделенных между тремя дата-центрами:

$$\text{Узлов на дата-центр} = \\ = \frac{\text{Минимальное количество серверов}}{\text{Минимальное количество дата-центров} - 1} = \\ = \frac{34}{4-1} = 11{,}333 = 12 \text{ серверов/дата-центр.}$$

Поскольку нам нужны 12 серверов на дата-центр с учетом того, что каждый из них однажды может выйти из строя, требуется 12 серверов в *каждом* дата-центре.

$$\text{Всего узлов} = \text{Узлов на дата-центр} \times 4 = 48 \text{ узлов.}$$

Понадобится 48 узлов, чтобы гарантировать, что у вас будут 34 работающих узла даже в случае отказа одного из дата-центров.

Как может повлиять на расчеты изменение количества дата-центров? Рассмотрим пример 14.6.

Пример 14.6. Различное количество дата-центров

Что, если у нас всего два дата-центра? Повторим расчеты:

$$\text{Узлов на дата-центр} = \\ = \frac{\text{Минимальное количество серверов}}{\text{Минимальное количество дата-центров} - 1} = \\ = \frac{34}{2-1} = 34;$$

Всего узлов = Узлов на дата центр × 2 = 68.

Если у вас два дата-центра, нужны 68 узлов. А как насчет других вариантов? Если у вас:

- *четыре дата-центра*, для обеспечения их отказоустойчивости необходимы 48 узлов;
- *шесть дата-центров*, для обеспечения их отказоустойчивости необходимы 42 узла.

Напрашивается парадоксальный на первый взгляд вывод: чтобы обеспечить возможность продолжения работы после выхода из строя целого дата-центра, вам требуется тем меньше узлов, тем больше у вас дата-центров, хотя, если не задумываться, это кажется неестественным.

Следует запомнить: хотя этот пример может быть не вполне применим к реальным ситуациям, его основная мысль работает. Тщательно подходите к разработке планов отказоустойчивости. Интуиция может вас подвести, а если это произойдет, вам грозят проблемы с доступностью.

Скрытые общие свойства — угроза отказа системы

Часто считается, что какие-то аварийные сценарии независимы друг от друга и вероятность их одновременного наступления невысока, но может оказаться, что между ними есть связь. Это значит, что они могут произойти одновременно, а в некоторых ситуациях этого даже следует ожидать.

Пример 14.7. Стеллажирование

Допустим, ваш сервис работает на четырех узлах. Вы стараетесь предотвращать проблемы, поэтому задействуете шесть узлов — достаточно, чтобы сохранить работоспособность при одновременном плановом обслуживании одного узла и отказе другого.

Однажды электричество гаснет, и вы идете к генератору. И только сейчас понимаете, что единственный способ попасть в гараж, где хранится ваш генератор, — пройти через управляемую электричеством дверь, которая, конечно, не работает, потому что электричества нет.

Упс...

Само наличие запасного плана не означает, что вы можете реализовать его в случае необходимости.

Точно такие же проблемы существуют и в мире сервисов. Может ли какой-то сбой сервиса усложнить ремонт этого самого сервиса, потому что это вызовет другие проблемы, казалось бы не связанные с изначальной? Например, если ваш сервис вышел из строя, можно ли развернуть на нем обновленную версию? Что, если авария произойдет на сервисе доставки обновлений? Что произойдет в случае отказа сервиса, который вы используете для мониторинга производительности других сервисов?

Вывод: убедитесь, что ваши планы решения проблем будут работать даже в случае наступления других проблем. Зависимости между проблемой и способом ее решения могут привести к проблемам с доступностью.

Управление приложениями

Запас на две ошибки в нашем контексте означает следующее: *не ограничивайтесь поверхностным анализом аварийных режимов*. Рассматривайте проблему глубже. Убедитесь в том, что у вас нет зависимых аварийных режимов, а приготовленные механизмы восстановления действительно способны восстановить работу системы в случае сбоя.

Кроме того, не стоит игнорировать проблемы. Они не решатся сами собой, зато могут повлиять на ваши планы обеспечения доступности. Не стоит считать, что выход из строя резервной базы данных — некритическая проблема, только потому, что это резервная база. Восстановите ее функционирование столь же тщательно, как вы сделали бы с основной базой, ведь они обе важны для работы системы.

Как говорит один мой друг, если что-то связано с производственной средой, это тоже производственная среда. Не считайте что-либо за пределами производственной среды малозначимым.

Это сложный материал. Скрытые проблемы или неочевидные зависимости обнаружить нелегко. Выделите время для анализа сложившейся у вас ситуации и решения найденных проблем.

Космический корабль

Закончу эту главу замечательным примером независимой, избыточной, многоуровневой и самовосстанавливающейся системы. Фактически это было одно из первых крупномасштабных программных приложений, где принципам обеспечения избыточности и обработки сбоев придавалось огромное значение. Иначе было нельзя, ведь от этого зависела жизнь астронавтов.

Я говорю о «Спейс Шаттл» (Space Shuttle) — американском многоразовом транспортном космическом корабле.

В программе «Спейс Шаттл» наблюдались серьезные проблемы с механикой, которые полностью осветить здесь невозможно. Но вот компьютерная система, встроенная в корабль, представляла собой передовой уровень техник избыточности и независимого восстановления после ошибок.

Основная компьютерная система «Спейс Шаттл» состояла из пяти компьютеров. Четыре из них были абсолютно одинаковыми, на них было установлено идентичное программное обеспечение, а пятый был иным. Мы поговорим о нем позже.

На четырех основных компьютерах была запущена одна и та же программа, отвечающая за критически важные этапы миссии, такие как взлет и приземление. На этих компьютерах хранились одни и те же данные, было установлено одно и то же программное обеспечение, и предполагалось, что они сгенерируют идентичные результаты. Все четыре выполняли одни и те же расчеты и постоянно сравнивали результаты. Если в какой-то момент один из компьютеров выдавал результат, отличающийся от результатов других, все четыре

15 Владение сервисами

В главе 12 мы говорили о том, что владеть сервисом и обслуживать его должна одна команда разработчиков внутри вашей организации, но не обсуждали подробно, что это значит. В этой главе мы рассмотрим понятие *«владение сервисами»*, а также поговорим о том, что необходимо для работы *отдельной команды, владеющей сервисной архитектурой*.

Отдельная команда, владеющая сервисной архитектурой

Что такое отдельная команда, владеющая сервисной архитектурой (ОКВСА)? ОКВСА — важный руководящий принцип для больших организаций, где работает много команд разработки, которые владеют и управляют сервисами, обеспечивающими работу одного или нескольких приложений.

Что же для приложения или организации означает иметь ОКВСА? Чтобы считаться ОКВСА, необходимо соответствовать следующим критериям.

- Архитектура приложения должна быть сконструирована на основе сервисов или микросервисов.
- За создание и обслуживание приложения отвечают несколько команд разработки.

Copyrighted material

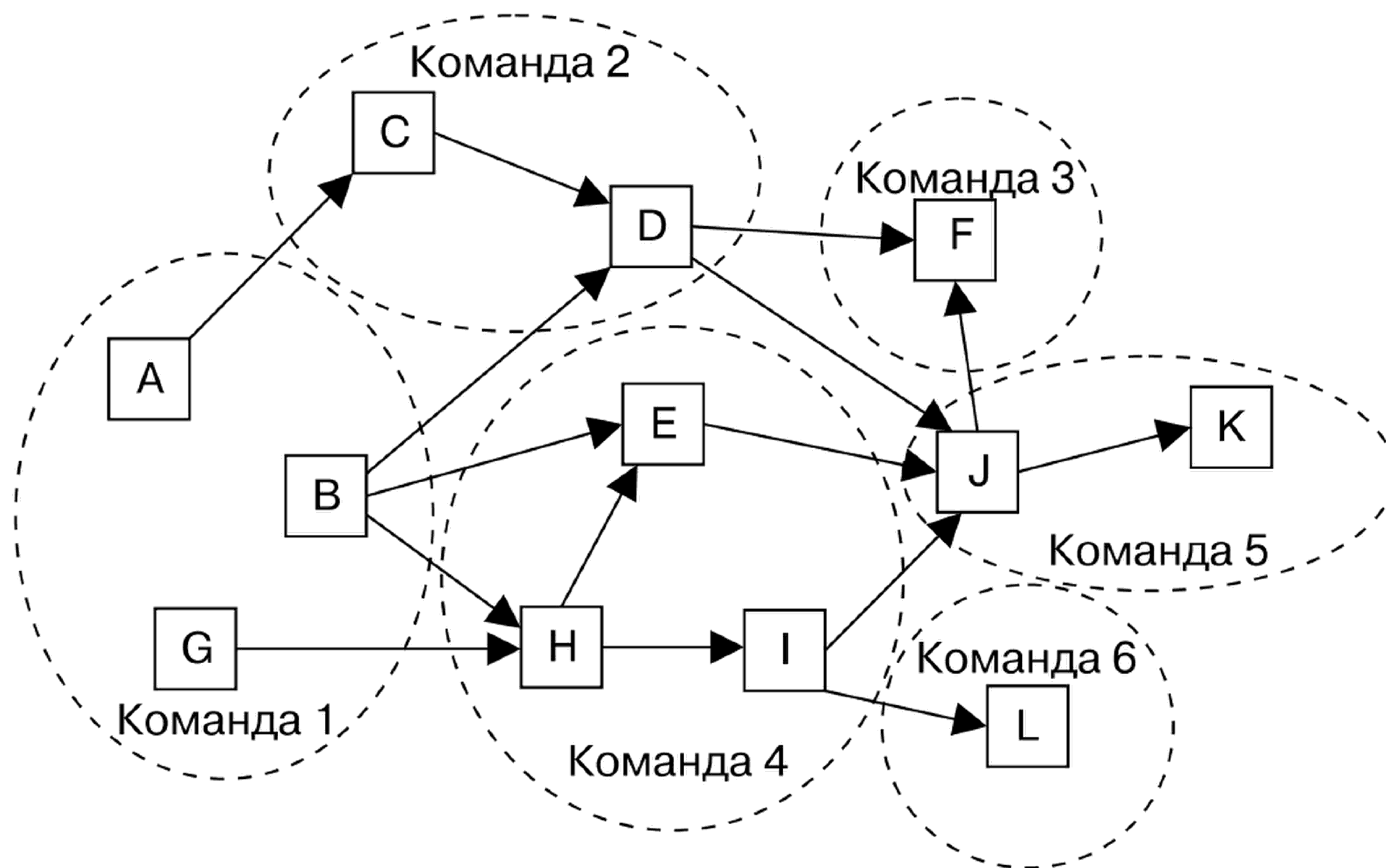


Рис. 15.1. ОКВСА-организация и ОКВСА-приложение

Получается, что у каждого из аспектов приложения есть четкая принадлежность. Легко определить, кто отвечает за любую часть приложения и с кем можно связаться в случае возникновения вопросов, проблем или необходимости изменений.

На рис. 15.2 показан пример приложения и организации, не соответствующих принципам ОКВСА.

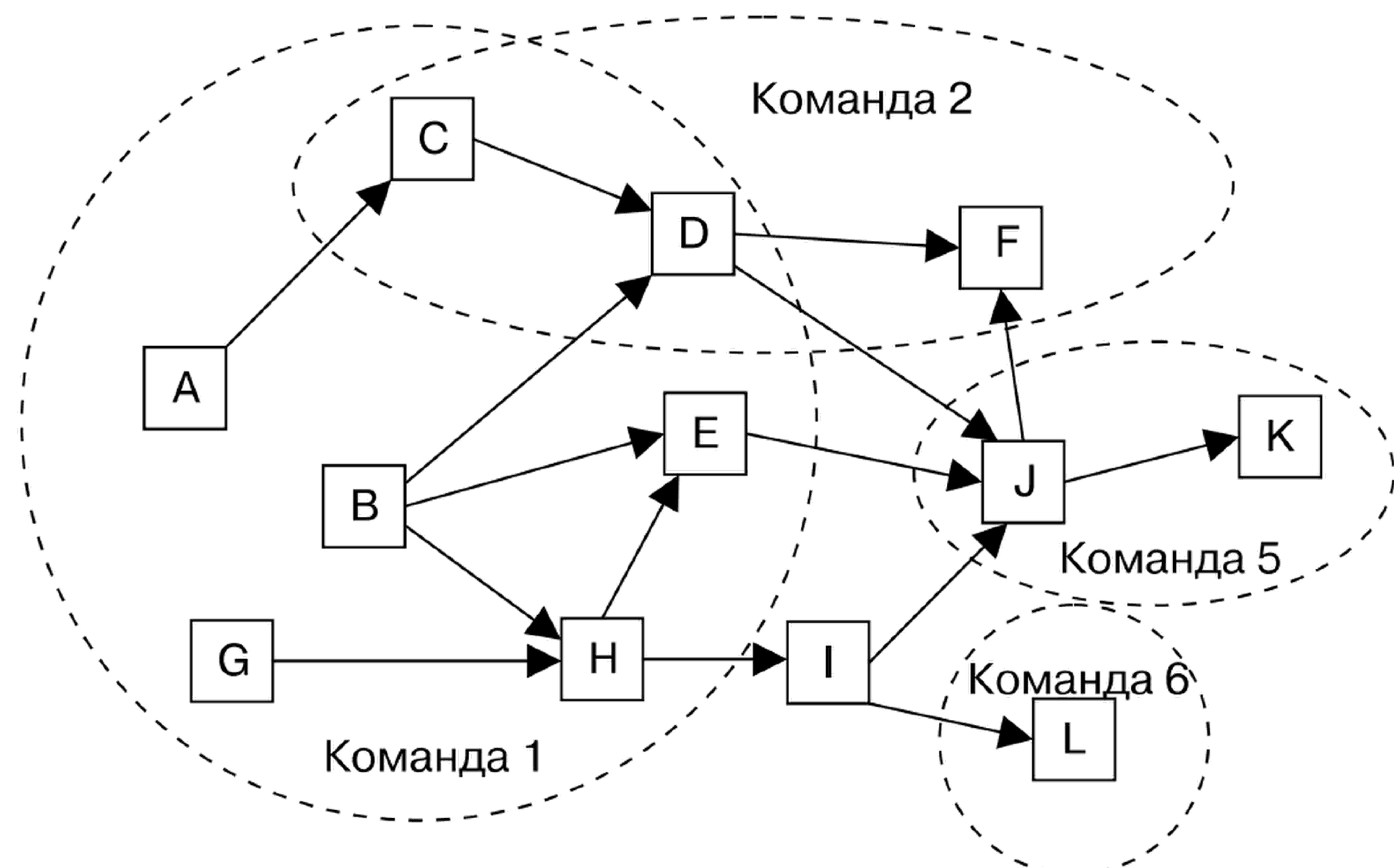


Рис. 15.2. Приложение и организация, не соответствующие принципам ОКВСА

Можно отметить несколько аспектов. Прежде всего, сервис I никому не принадлежит, в то время как сервисами C и D управляют несколько команд.

Четкой принадлежности нет. Если вам нужно, чтобы что-то было сделано в сервисах C и D, непонятно, кто отвечает за это. Если в каком-то из этих сервисов происходят проблемы, куда бежать? Что будет, если надо что-то сделать с сервисом I? К кому обращаться? Таким образом, отсутствие четкой принадлежности и разделения ответственности делает управление сложным приложением еще сложнее.

Преимущества организаций и приложений, соответствующих принципам ОКВСА

По мере роста приложения растет и его сложность. ОКВСА-приложения могут вырасти до больших размеров и находиться под управлением бо́льших команд разработки, чем не-ОКВСА. Так, масштаб может увеличиваться, при этом сохраняются четкость структуры, а также строго задокументированные и поддерживаемые интерфейсы.

ОКВСА-организация может разрабатывать более крупные и сложные приложения, чем не-ОКВСА. Это объясняется тем, что ОКВСА разделяет сложность системы между несколькими командами разработки эффективнее, сохраняя четкую принадлежность и рамки ответственности.

Что значит быть владельцем сервиса

В ОКВСА-организации команда, владеющая сервисом, полностью ответственна за все аспекты его работы. Эта команда может зависеть от помощи других команд (например, от техников, отвечающих за работу оборудования), но несет полную ответственность за работу своего сервиса.

Copyrighted material

В задачи команды могут входить следующие сферы.

- *Проектирование API*. Дизайн, реализация, тестирование и управление версиями всех API, внешних и внутренних, которыми управляется сервис.
- *Разработка сервиса*. Проектирование, реализация и тестирование бизнес-логики и бизнес-возможностей сервиса.
- *Данные*. Управление всеми данными, которыми владеет и управляет сервис, их схемой и представлением, шаблонами доступа и жизненным циклом.
- *Развертывание*. Процесс определения того, когда (и если) требуются обновление сервиса и доставка на сервис нового программного обеспечения, включая верификацию и откат всех узлов сервиса, а также обеспечение его доступности во время развертывания.
- *Окна развертывания*. Определение того, когда развертывание безопасно, а когда — нет. Эта задача включает внедрение практики плановых отключений сервиса на уровне компании или продукта, а также окон развертывания, специфичных для данного сервиса.
- *Изменения в производственной среде*. Все изменения производственной среды, нужные сервису (например, настройки балансировщика нагрузки и регулировка системы).
- *Среда*. Управление производственной средой, включая среду для разработки, стейджинговую и предпродуктовую среды для сервиса.
- *Мониторинг*. Ответственность за состояние мониторинга и отслеживание всех аспектов работы сервиса, включая SLA. Последовательный и регулярный контроль мониторинга.
- *Оперативное реагирование при инцидентах*. Обеспечение оповещений в случае, когда система функционирует неподобающим образом. Обеспечение постоянного дежурства, чтобы кто-то из членов команды всегда был доступен для обработки инцидентов. Обработка инцидентов внутри оговоренных SLA границ.
- *Отчетность*. Внутренняя отчетность для других команд (отвечающих за восходящие либо нисходящие зависимости), а также отчетность об оперативном состоянии сервиса.

Copyrighted material

16 Классы сервисов

В процессе работы больших сложных приложений со множеством сервисов могут возникнуть проблемы с доступностью. Отказ одного сервиса может привести к отказам других сервисов, чья работа зависит от него. В результате может произойти каскадная авария и все приложение выйдет из строя. Особенно неприятно сознавать, что изначально сбой произошел в сервисе, работа которого была абсолютно некритична для приложения, но это привело к отказу жизненно необходимых сервисов.

Сложность приложения

Иногда может выйти из строя самый маленький и незначительный сервис (рис. 16.1).

Но из-за этого может прекратить работу все приложение (рис. 16.2).

Существует много способов предотвратить отказы зависимых сервисов в таком случае. Некоторые из них мы обсудили в главе 13. Однако обеспечение отказоустойчивости между сервисами увеличивает сложность и удорожает приложение, и это не всегда необходимо. Посмотрим на рис. 16.3: что произойдет, если сервис Г неважен критически для сервиса А? Почему сервис А должен выйти из строя, если отказывает сервис Г?

Copyrighted material

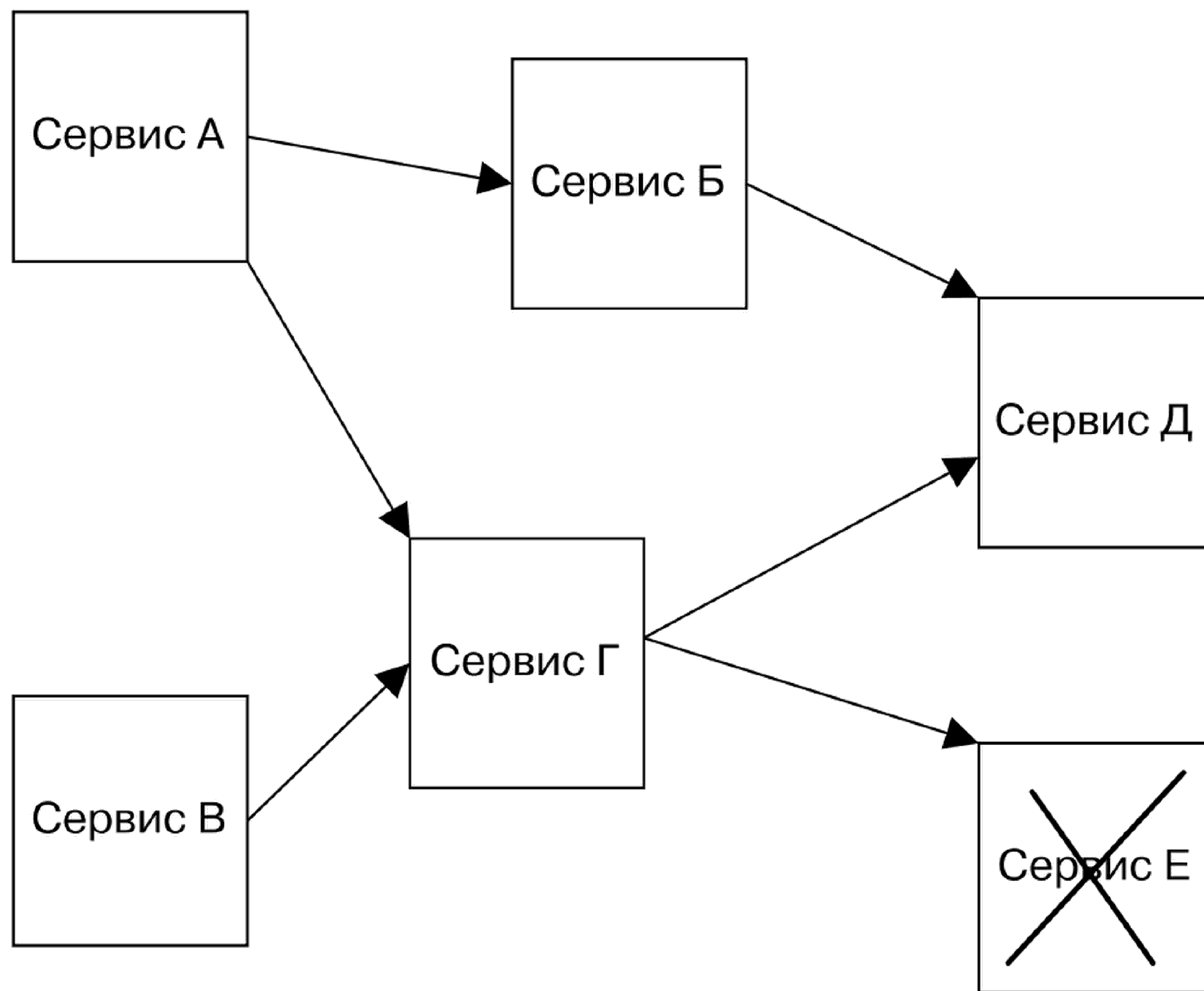


Рис. 16.1. Сбой одного сервиса...

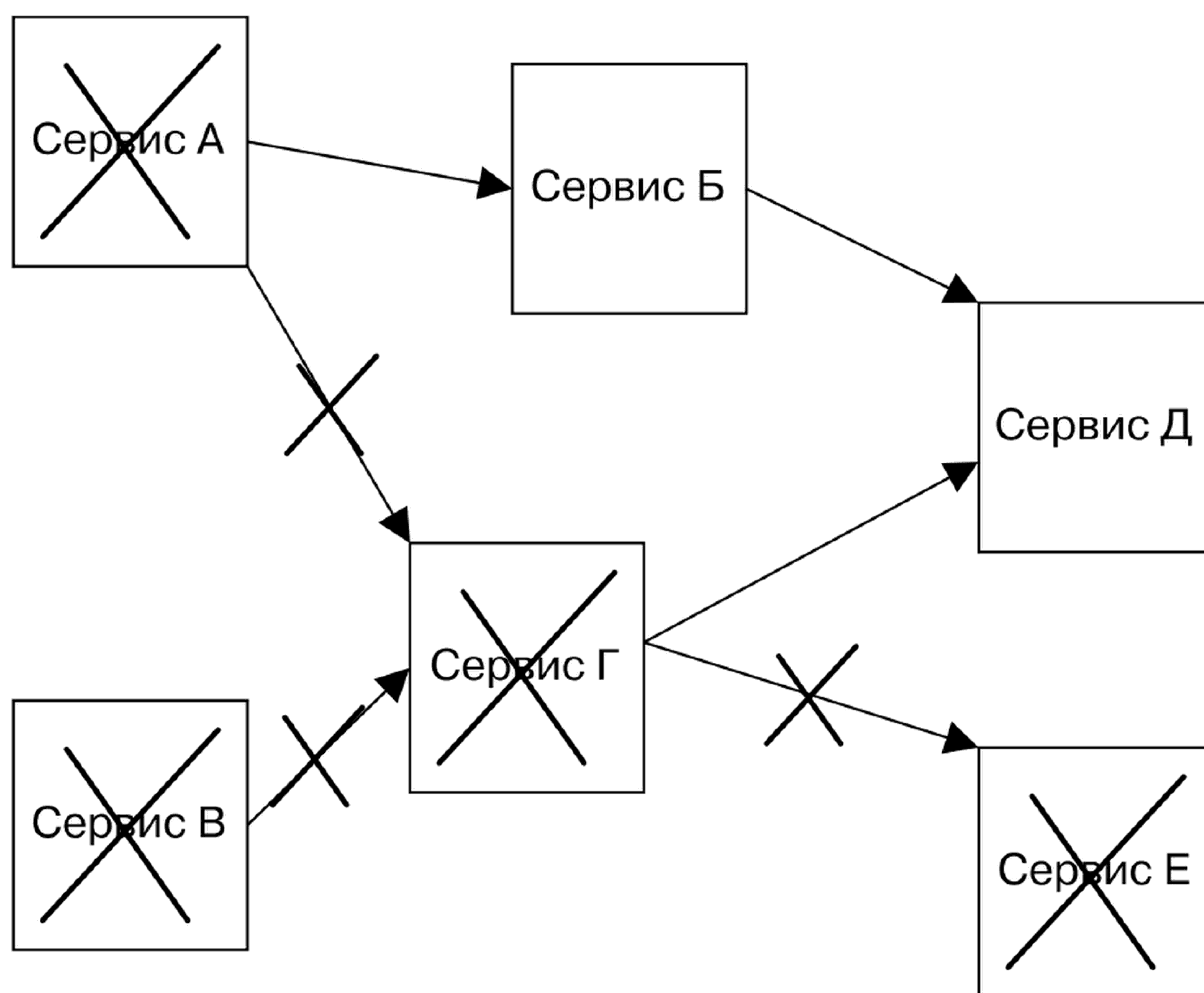


Рис. 16.2. ...Может вызвать каскадную аварию

Как узнать, когда связь между зависимыми сервисами критически важна, а когда — нет? Один из способов найти ответ — классы сервисов.

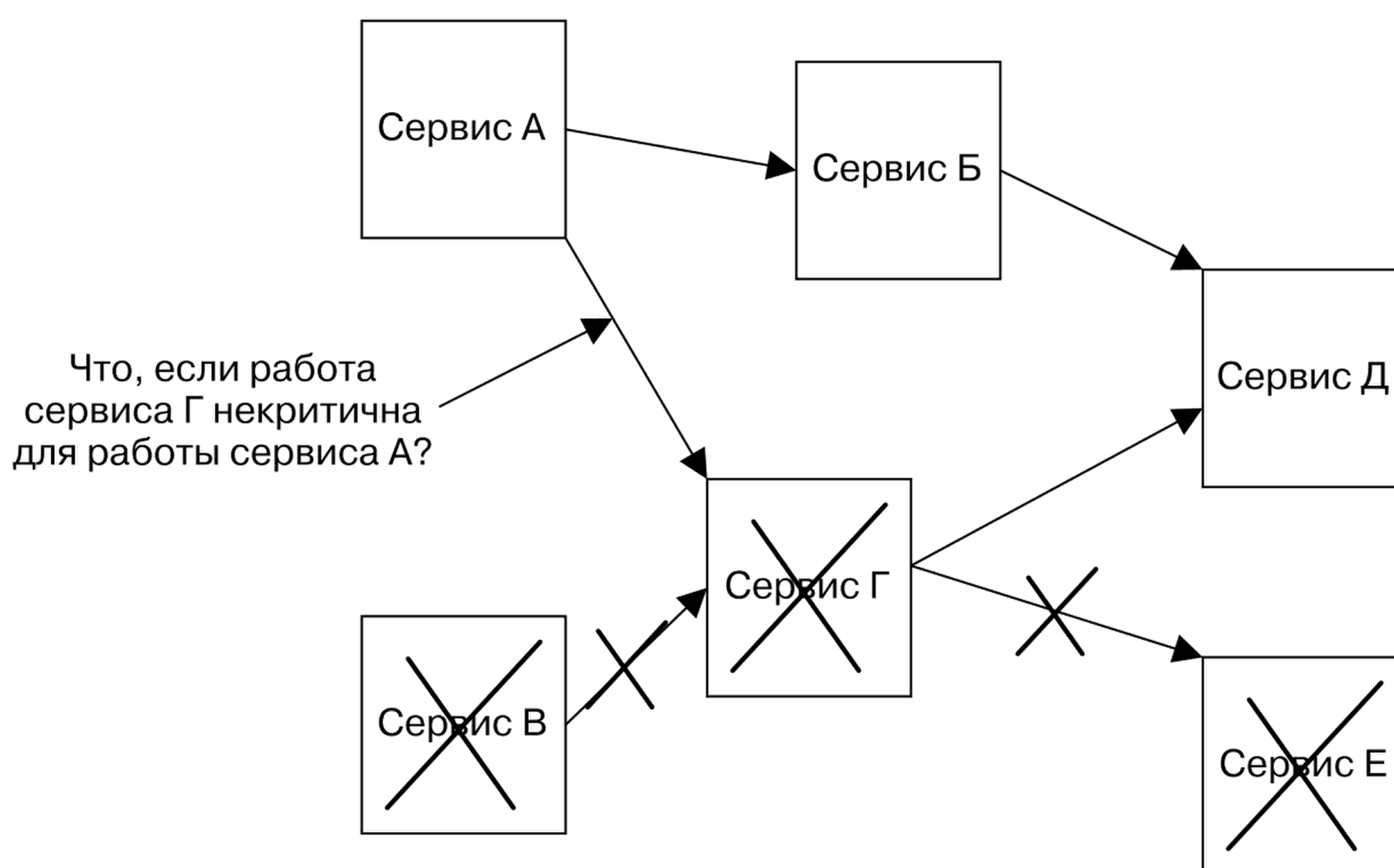


Рис. 16.3. Что, если работа сервиса Г некритична?

Что же такое классы сервисов

Класс сервиса — это просто *метка*, присвоенная сервису и означающая, насколько важен данный сервис для функционирования вашего бизнеса. Классы сервисов помогают отделить жизненно важные сервисы от тех, чья работа нужна и полезна, но не критически необходима.

Сравнивая классы зависимых сервисов, вы можете решить, какие зависимости сервиса более важные, а какие — менее.

Присвоение сервисам меток сервисных классов

Всем сервисам в вашей системе независимо от их размера необходимо назначить класс. Далее приведены рекомендации, которые вы можете использовать для начала (можно внести в них необходимые корректировки, чтобы они точно соответствовали нуждам бизнеса).

Класс 1

Сервисы класса 1 — важнейшие в вашей системе. Сервису можно присвоить класс 1, если его отказ оказывает значительный негативный эффект на клиентов или прибыль компании.

Вот примеры сервисов класса 1.

- *Сервис авторизации*. Позволяет вашим пользователям войти в систему.
- *Обработчик кредитных карт*. Обрабатывает платежи клиентов.
- *Сервис прав доступа*. Определяет, какие функциональности вашей системе каким пользователям доступны.
- *Сервис обработки заказов*. Позволяет клиентам купить продукцию на вашем сайте.

Сбой сервиса класса 1 имеет самые тяжелые последствия для компании.

Класс 2

Сервисы класса 2 также важны для вашего бизнеса, но менее критичны, чем относящиеся к классу 1. Отказ работы сервиса класса 2 может привести к получению клиентом неприятного опыта взаимодействия с вашей системой, но все-таки не лишает его возможности с ней работать.

К сервисам класса 2 относятся также те, которые значительно влияют на внутренние бизнес-процессы, но клиентов напрямую не касаются.

Вот несколько примеров сервисов класса 2.

- *Сервис поиска*. Сервис, обеспечивающий работу функции поиска на сайте.
- *Сервис комплектации заказов*. Сервис, с помощью которого сотрудники складов собирают заказы и отправляют их покупателям.

Сбой в работе сервиса класса 2 негативно повлияет на покупателей, но не приведет к отказу всей системы.

Класс 3

Сервис класса 3 — это сервис, имеющий незначительное, незаметное или почти незаметное влияние на клиентский опыт взаимодействия либо ограниченно влияющий на ваши бизнес или систему.

Вот несколько примеров сервисов класса 3.

- *Сервис пользовательских значков*. Сервис, отображающий значок или аватар пользователя на странице сайта.
- *Сервис рекомендаций*. Сервис, отображающий альтернативные продукты, которыми мог бы заинтересоваться покупатель, на основе истории просмотров.
- *Сервис «Сообщение дня»*. Сервис, отображающий сообщения или предупреждения для покупателей вверху веб-страницы.

Возможно, покупатели даже не заметят выхода из строя сервиса класса 3.

Класс 4

В случае отказа сервиса класса 4 никакого негативного влияния на опыт взаимодействия покупателей или финансовые показатели не наблюдается.

Вот несколько примеров сервисов класса 4.

- *Сервис генерации отчетов о продажах*. Сервис, генерирующий еженедельный отчет о продажах. Этот отчет важен, однако краткосрочный отказ сервиса его генерации не вызовет значительных проблем.
- *Сервис рекламной рассылки*. Сервис, генерирующий электронные письма, регулярно рассылаемые вашим покупателям. Если он какое-то время не работает, письма могут быть отправлены с задержкой, но, как правило, это не оказывает значительного негативного влияния на клиентов.

Copyrighted material

страницы товаров и в целом использовать сайт при неработающей строке поиска. Как правило, работать с сайтом в этом случае некомфортно, но все-таки можно.

- *Сервис базы данных каталога (класс 1)*. База данных, где хранится сам каталог.

  Это сервис класса 1, так как, если база не работает, ни один товар не удастся отобразить.

- *Сервис редактирования каталога (класс 3)*. Сервис, с помощью которого ваши сотрудники добавляют новые предметы в каталог или обновляют уже имеющиеся.

  Этому сервису можно присвоить класс 3, так как его задача некритична для того, чтобы клиенты могли успешно сделать заказ. Правда, невозможность добавления новых товаров в каталог негативно повлияет на ваш бизнес, но все-таки немедленного или прямого воздействия на клиентов нет. Значит, непродолжительный отказ работы сервиса допустим.

- *Сервис оплаты (класс 1)*. Это сервис, который формирует процесс оплаты для ваших покупателей. Без него они не смогут ничего у вас приобрести.

  Это сервис класса 1, так как его выход из строя значительно влияет как на клиентов (они не могут ничего купить), так и на бизнес (вы не получаете прибыль, если клиенты ничего не покупают).

- *Сервис отправки заказов (класс 3)*. Это сервис, отвечающий за упаковку и отправку заказов покупателям (упростим для примера). Без него ваши покупатели не получат товары, которые заказали.

  Казалось бы, это должен быть сервис класса 1, потому что своевременная доставка заказов — жизненно важный аспект вашего бизнеса. Но давайте подумаем: если отправка заказов будет невозможна в течение часа или около того, как это повлияет на клиентов? А на бизнес? В большинстве случаев эффект окажется крайне незначительным — задержка отправки на час слабо повлияет на то, когда покупатели получат свои заказы. Возможно определенное негативное воздействие на ваш бизнес, так как

Copyrighted material

сотрудники, упаковывающие заказы, какое-то время не смогут выполнять свою работу. Таким образом, поскольку влияние на бизнес незначительно, а на клиентов — отсутствует вовсе, можно присвоить этому сервису класс 3.

- *Еженедельный отчет о заказах (класс 4)*. Это сервис, который собирает данные о заказах и формирует еженедельный отчет для менеджмента и финансового отдела.

  Это сервис класса 4, так как никакого воздействия на клиентов он не оказывает. Задержка же отчета на короткое время может немного повлиять на ваш бизнес, но, скорее всего, незначительно.

Руководствуясь этими примерами, вы можете сформировать ряд сервисных классов для собственных сервисов.

Что дальше?

Разобравшись в сути классов, вы сможете назначить соответствующие метки сервисных классов всем сервисам в своем приложении. Но после того, как они будут классифицированы, что делать с этими метками? Какую пользу они приносят? Поговорим об этом в главе 17.

Copyrighted material

Реагирование

Когда в системе возникает проблема, ваша реакция на нее зависит от следующих двух факторов:

- серьезности проблемы;
- класса сервиса, в котором она произошла.

Серьезная проблема с сервисом класса 1 должна считаться более важной, чем серьезная проблема с сервисом класса 3. Это понятно. Но даже если с сервисом класса 1 случилась проблема средней критичности, она должна получить более высокий приоритет, чем очень серьезная проблема с сервисом класса 3. Это демонстрирует рис. 17.1.

Рис. 17.1. Реагирование в соответствии с классом сервиса и критичностью проблемы

Чем критичнее проблема или выше класс сервиса (то есть чем ниже номер класса), тем быстрее нужно реагировать. Параллельные линии на рис. 17.1 означают реакцию одного уровня. Если с сервисом класса 1 возникла проблема низкой или средней критичности, то требуется тот же уровень реагирования, что и на очень серьезную проблему с сервисом класса 3. А проблемы с сервисом класса 4 почти никогда не требуют немедленного реагирования. Более того, проблема низкой критичности с сервисом класса 2 потребует того же уровня реакции, что и очень серьезная проблема с сервисом класса 4.

Вы можете использовать эту информацию для формирования различных аспектов способов реагирования, например таких.

- Какие проблемы с какими сервисами требуют немедленного прямого оповещения об аварии?
- Каковы ожидаемые последствия согласно SLA?
- Каков иерархический путь для медленной реакции или не сразу проявляющихся последствий?
- Каков необходимый временной период реагирования (круглосуточно или только в рабочее время)?
- Потребуются ли изменения в производственной среде или экстренная доставка изменений?
- Каковы SLA, регламентирующие работу вашего сервиса с точки зрения доступности и реагирования?

Зависимости

Если вы создаете сервис, то связь между классом, который вы назначили своему сервису, и сервисными классами ваших зависимостей чрезвычайно важна. На рис. 17.2 показаны зависимости между уровнями вашего сервиса и его зависимостей.

Если ваш сервис более высокого класса (номер его класса меньше), чем сервис, от которого вы зависите, значит, это зависимость критического уровня важности. Если же класс вашего сервиса ниже (номер его класса больше), чем у сервиса, от которого вы зависите, то зависимость некритична.

Copyrighted material

Сервис, от которого вы зависите	Ваш сервис: Уровень 1	Уровень 2	Уровень 3	Уровень 4
Уровень 1	Критическая зависимость			
Уровень 2				
Уровень 3				
Уровень 4				Некритическая зависимость

Рис. 17.2. Критичность сервисных зависимостей

Критическая зависимость

Если, как в примере 17.1, зависимость является критической, вам как разработчику сервиса очень важно решать проблемы с вашей зависимостью так, чтобы влияние на ваш сервис было минимальным.

Вы отвечаете за то, чтобы ваш сервис сохранял максимально возможную работоспособность в случае отказа критической зависимости. Это объясняется тем, что сервис, от которого вы зависите, обладает более низким классом (номер класса больше) и, скорее всего, его надежность и доступность ниже, чем у вашего сервиса.

Пример 17.1. Критическая зависимость

Посмотрите на приложение, показанное на рис. 16.4, и обратите внимание на сервис клиентской части, которому присвоен класс 1. Когда этот сервис старается отобразить для клиента страницу с информацией о деталях конкретного товара, ему надо выяснить цену этого товара. Для этого он вызывает сервис расчета цен и стоимости доставки (СРЦД).

Copyrighted material

класс 4. Для получения информации, необходимой для формирования отчета о заказах, этот сервис вызывает сервис обработки заказов, относящийся к классу 1.

Что будет, если сервис обработки заказов вышел из строя? Что предпринять сервису формирования отчетов о заказах? Вероятно, проще всего ему тоже попросту прекратить работу. Поскольку сервису обработки заказов присвоен класс 1, любые проблемы с ним должны быть решены как можно быстрее, с высочайшим приоритетом — намного выше, чем заслуживает отказ сервиса формирования еженедельных отчетов о заказах.

Поэтому сервис отчетов о заказах может не предпринимать ничего особенного в случае отказа сервиса обработки заказов. Для него абсолютно приемлемо просто перестать функционировать, если сервис обработки заказов недоступен.

Резюме

Сервисные классы являются удобным способом отражения уровня критичности сервиса для его владельцев, зависимостей и потребителей. Они позволяют представить ожидания от работы сервиса в простом для понимания и обмена информацией виде. Простота уменьшает вероятность ошибок, а сервисные классы представляют собой несложную модель, позволяющую обсуждать ожидания.

Copyrighted material

- *Отправка.* Клиенты ожидают, что получат свои заказы довольно быстро. Вы можете отразить это в виде указания времени с момента получения заказа до отправки или периода времени, в течение которого товар доставляется клиенту, например: «Мы отправляем все заказы в течение 24 ч».

Все это — примеры SLA. Хотя их типы и значения очень разные, по сути, у них всех одна цель: они выражают ожидания клиентов от вашего приложения.

Вы можете измерить производительность каждого из этих аспектов своего приложения, когда оно функционирует и взаимодействует с клиентами. Можно также построить графики и диаграммы, отражающие изменения этих параметров во времени. Но SLA — это согласованный предел, в рамках которого производительность вашего сервиса может считаться соответствующей ожиданиям. Диаграмма, приведенная на рис. 18.1, иллюстрирует производительность магазина с точки зрения наличия товаров на складе (этот параметр равен отношению количества продуктов, находящихся на складе, к их ассортименту в каталоге в каждый момент времени).

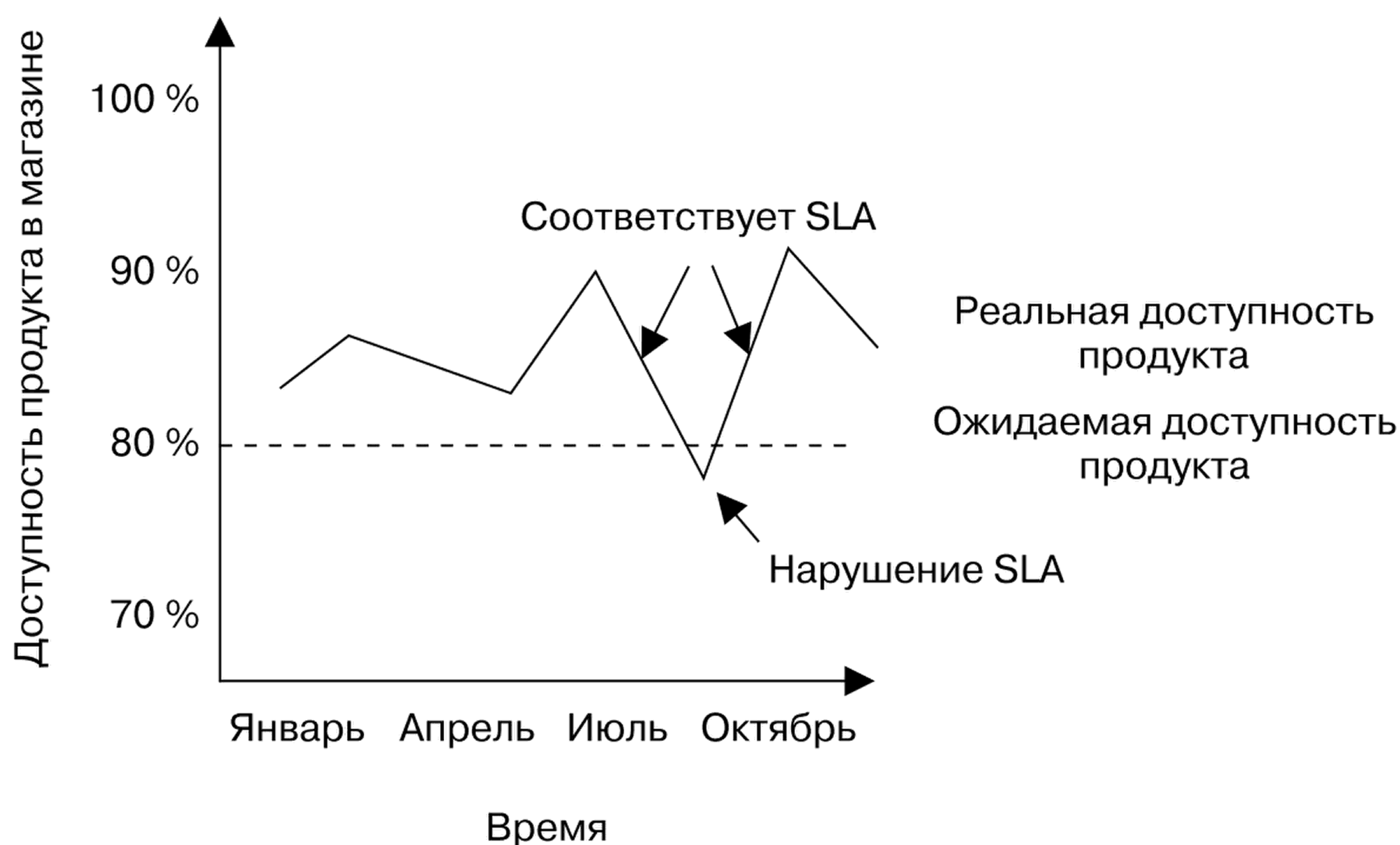


Рис. 18.1. Пример проверки соответствия производительности требованиям SLA

Но SLA могут быть использованы также между отдельными сервисами внутри вашего приложения. Можете применять их как механизмы взаимодействия и обмена ожиданиями между владельцами и потребителями отдельных сервисов.

Почему внутренние SLA так важны

Внутренние SLA чрезвычайно важны для обеспечения работоспособности и сопровождаемости сложных многосервисных систем. Почему? Ну хотя бы по той причине, что сервис не может выполнять свои обязательства перед пользователями, если сервисы, от которых он зависит, не выполняют свои обязательства перед ним. Обратитесь к главе 15 за более подробной информацией.

Как вы можете обеспечить для клиентов скорость реакции 50 мс, если сервис, от которого вы зависите, реагирует на ваш запрос лишь за 90 мс?

Как вы можете обеспечить 99%-ную доступность, если сервис, от которого вы зависите, дает только 90 %?

SLA создают доверие

Суть SLA — формирование и последовательное распространение уверенности. Будучи убежденными в том, что сервис, от которого вы зависите, всегда будет соответствовать ожиданиям, вы можете уверенно формировать ожидания от своего сервиса.

Пример 18.2. Формирование уверенности

Снова вспомним интернет-магазин, изображенный на рис. 16.4. Допустим, вы и ваша команда отвечаете за сервис расчета цены и стоимости доставки. Ваши клиенты — сервис клиентской части сайта и сервис оплаты. Одна из основных операций, за которые вы отвечаете, — поиск цены данного продукта по его номеру. Поскольку эти сервисы используют данную величину для генерации веб-страницы, которую видят конечные пользователи, им нужно, чтобы цена была найдена очень быстро. Ваша команда устанавли-

вает соглашение, согласно которому определение цены длится не более 20 мс на один запрос.

При этом вы понимаете: чтобы иметь возможность выполнять это требование, вам нужен быстрый доступ к сервису каталожного хранилища, где хранятся данные, с помощью которых вычисляется цена. Вы не уверены, сможет ли этот сервис предоставить вам информацию настолько быстро, чтобы вы смогли выполнить свое обязательство насчет 20 мс. Этим сервисом владеет другая команда. Как вы можете быть уверены, что она сможет соответствовать вашим требованиям производительности? У вас есть два варианта.

Во-первых, можно связаться с командой-владельцем и с помощью ее членов разобраться в работе их сервиса, вникнуть в проблемы обеспечения производительности. Затем познакомиться с командой и проанализировать, смогут ли они, по вашему мнению, сделать все как полагается. Разумеется, это очень дорогой, невежливый, да и непрактичный для большой организации путь.

Во-вторых, можно провести переговоры с командой-владельцем и договориться о SLA по вопросу производительности их сервиса. Допустим, что в результате переговоров они согласились обеспечивать ответ за 10 мс. Если они и в самом деле будут выдавать результат за 10 мс, вы сможете выполнить свое обязательство (20 мс) перед потребителями.

Поскольку другая команда соответствует своему SLA, это позволяет и вам соответствовать своему.

Вы можете отслеживать соответствие производительности другой команды SLA, чтобы оценить, как у них идут дела. Если команда долгое время соблюдает SLA, ваше доверие к ним растет и вы можете сконцентрироваться на своем сервисе и собственных задачах по обеспечению гарантий в 20 мс для потребителей.

SLA помогают в диагностике проблем

SLA представляют собой отличный инструмент для локализации проблем в сложной системе. Если у сервиса наблюдаются какие-либо проблемы, одно из первых действий, которые необходимо

Copyrighted material

предпринять, — проверка, выполняет ли сервис, от которого вы зависите, свои обязательства согласно SLA. Если он не соответствует требованиям, у вас есть отличная стартовая точка для диагностики проблем, возникших у вашего сервиса.

Пример 18.3. Обнаружение проблемы

Вернемся к интернет-магазину, изображенному на рис. 16.4. Допустим, что вы и ваша команда — владельцы сервиса расчета цены и стоимости доставки, как описано в примере 18.2.

А сейчас представьте себе, что вас разбудили звонком посреди ночи. Ваш сервис генерирует цены все медленнее и медленнее, что сильно мешает покупателям магазина. Вы выясняете, соответствует ли ваша производительность заявленным 20 мс, и обнаруживаете, что каждый поиск занимает в среднем 500 мс. Разумеется, это значительно тормозит работу клиентской части сайта и покупатели недовольны.

Но что же вызвало эту проблему? Что-то не так с сервисом? Или проблема произошла с каким-то из тех, от которых вы зависите?

Может быть, это и в самом деле ваш сервис — что-то с оборудованием или с чем-нибудь еще. Но прежде, чем тратить время на поиск проблем на своей стороне, следует проверить производительность тех сервисов, от которых вы зависите.

Зная, что работа вашего сервиса зависит от сервиса каталожного хранилища, а они обещали вам давать ответ на каждый запрос за 10 мс, вы проверяете их производительность на соответствие SLA. Выясняется, что вместо 10 мс вызовы к этому сервису занимают целых 400 мс. Получается, что проблемы с производительностью на стороне этой команды. Вы проверяете, известно ли им об этом, и обнаруживаете, что их сотрудники уже в курсе и работают над проблемой.

Убедившись, что причина проблем с производительностью, скорее всего, в этом, вы начинаете отслеживать, как движется работа другой команды по исправлению проблемы. Это куда более разумно, чем тратить драгоценное время на поиск проблем на своей стороне.

Если у вас есть четко определенные SLA со всеми сервисами, от которых зависит работа вашего, отследить, где именно случилась проблема: у вас или у кого-то из них, довольно просто.

Измерение производительности для SLA

Есть много способов измерять производительность сервиса. Конкретные методы могут и должны варьироваться в зависимости от нужд и требований владельцев и потребителей сервиса. Вот несколько примеров типов измерений производительности.

- *Время ожидания.* Измерение времени, которое требуется сервису на то, чтобы обработать запрос и вернуть результат. Как правило, измеряется в милли- или микросекундах. Для потребителя очень важно знать, сколько времени потребуется на обработку его запроса, так как эта величина входит в общую продолжительность обработки запросов самим потребителем. Этот тип SLA упоминается в примерах 18.2 и 18.3.
- *Объем трафика.* Это измерение количества запросов, которые сервис может обработать за какой-то период. Как правило, единицей измерения служит количество запросов в секунду. Таким образом владелец сервиса может узнать, какого объема трафика можно ожидать от потребителя.
- *Продолжительность исправного состояния.* Эта величина означает, сколько времени подряд сервис, как предполагается, будет работоспособен, правильно и без серьезных проблем исполняя свои функции. Обычно вычисляется как процентное соотношение, означающее, какую долю от заданного периода времени (как правило, дня, месяца или года) сервис был доступен.
- *Количество ошибок.* Измерение, показывающее, сколько ошибок генерирует сервис в течение заданного времени. Как правило, измеряется как процентное отношение количества необработанных запросов к общему числу запросов, сделанных за данный период.

Copyrighted material

Пределы SLA

В SLA, как правило, определяется определенный *уровень оперативности*, который должен быть достигнут. Если актуальная производительность укладывается в этот предел, можно считать, что наша работа *соответствует SLA*. Если же производительность выходит за его рамки, считается, что *SLA нарушены*. Величина этого предела является SLA как таковым.

Например, утверждение «Количество запросов не должно превышать 1000 з./с» устанавливает предел SLA, ограничивающий допустимый объем трафика на какой-либо сервис. Другой пример: «Реакция на запрос должна поступать менее чем за 20 мс» — устанавливает предел SLA допустимой задержки обработки запроса на сервисе. Вы можете установить пределы SLA для большинства типов измерений производительности.

Верхний перцентиль SLA

Пределы SLA хороши, когда вы можете измерить какую-либо величину, а также получить гарантию того, что она будет соответствовать заданному пределу постоянно. SLA такого типа отлично подходят для отражения доступности, времени нахождения в исправном состоянии и количества ошибок.

Другой тип измерений SLA — *верхний перцентиль SLA*. Он нужен для измерения производительности события, если эта величина значительно меняется в различных ситуациях.

Верхние перцентили SLA очень полезны для измерения таких, например, показателей, как время обработки вызова. Количество времени, которое требуется сервису для обработки запроса и выдачи результата, может значительно различаться. Как правило, нам не нужно, чтобы время обработки *каждого* запроса укладывалось в заданные пределы, нам важно, чтобы *большая часть* запросов была обработана за требуемое или меньшее время.

Верхний перцентиль SLA равен процентной доле полученных результатов, которые находятся выше или ниже заданной вели-

чины. SLA в данном случае, как правило, формулируются следующим образом: «ВП <*процентная доля*> меньше, чем <*заданная величина*>». Например, «ВП90 не превышает 20 мс». Это следует понимать так: «Обработка 90 % запросов занимает не более 20 мс на каждый».

Зачастую мы вычисляем производительность какого-либо события, например время обработки запроса на сервисе, и указываем ее в виде верхнего перцентиля этого сервиса, как описано в примере 18.4.

Пример 18.4. Время обработки запроса на сервисе в верхнем перцентиле

Допустим, у нас есть сервис, который обрабатывает вызовы. Вот какие величины задержки обработки вызовов мы получили за определенный период времени (табл. 18.1).

Таблица 18.1. Актуальные величины задержки обработки вызовов

Время запроса	**Задержка, мс**
$T + 1$ с	5
$T + 2$ с	10
$T + 3$ с	20
$T + 4$ с	30
$T + 5$ с	15
$T + 6$ с	8
$T + 7$ с	12
$T + 8$ с	45
$T + 9$ с	12
$T + 10$ с	22
$T + 11$ с	4
$T + 12$ с	8

Продолжение ☞

Copyrighted material

Таблица 18.1 *(продолжение)*

Время запроса	**Задержка, мс**
T + 13 с	12
T + 14 с	15
T + 15 с	14
T + 16 с	28
T + 17 с	21
T + 18 с	32
T + 19 с	15
T + 20 с	22

На диаграмме это можно отобразить так.

Используя эти данные, мы можем вычислить *наибольшие величины* (верхние перцентили) задержки на сервисе.

- *ВП 90*. Это величина, которую не превышают 90 % значений задержки. В данном случае 90 % данных составляют 18 величин. Убрав две наибольшие величины в таблице (45 и 32 мс), мы получаем 18 величин, наибольшая из которых — 30 мс. Это значит, что *ВП 90 составляет 30 мс*.

- *ВП 80*. Это величина, которую не превышают 80 % значений задержки. Следовательно, нужно удалить 20 % значений (четыре наибольших: 45, 32, 30 и 28 мс). Самое большое значение из 16 оставшихся — 22 мс. Это значит, что *ВП 80 составляет 22 мс*.

Поступая аналогично, мы можем получить еще несколько величин ВП, характеризующих наши данные:

- ВП 95 = 32 мс;
- ВП 90 = 30 мс;
- ВП 80 = 22 мс;
- ВП 50 = 14 мс.

Иногда могут быть полезны и другие значения, например:

- $ВП_{макс}$ = 45 мс (наибольшее значение);
- $ВП_{мин}$ = 4 мс (наименьшее значение);
- $ВП_{ср}$ = 18 мс (среднее значение).

Разумеется, верхние перцентили беспрерывно меняются со временем. Поэтому, вычислив эти величины, вы можете установить пределы SLA, чтобы нормировать ожидания. Например, продолжая рассматривать пример 18.4, можно установить для вашего сервиса SLA ВП 90 < 35 мс. Если это так, сервис соответствует данному SLA. Однако, если установить следующую величину SLA, то ВП 80 < 20 мс и сервис не будет ей соответствовать (актуальная величина ВП 80 — 22 мс). Значит, сервис нарушает свое SLA.

Группы задержек

Иногда SLA имеют вид связанных между собой групп величин. Например, сервис должен быть способен гарантировать определенную величину задержки, но только если объем вызовов остается в определенных пределах. Например, SLA могут быть сформулированы так:

- «ВП 90 задержек вызова < 25 мс, если объем трафика < 250 з./с»;
- «ВП 90 задержек вызова < 30 мс, если 250 з./с < объем трафика < 400 з./с».

Copyrighted material

Сколько и какие внутренние SLA установить

Разрабатывая свой сервис, вы можете задаться вопросом: сколько внутренних SLA нужно разработать для него?

Прежде всего, их должно быть как можно меньше. Осознавать значение SLA и их влияние становится намного сложнее по мере того, как растет их количество.

Убедитесь, что вы охватили все критически важное для вашего сервиса. У вас должны быть SLA для всех основных функциональностей, особенно для тех, что чрезвычайно важны для бизнеса.

Надо обсудить SLA со всеми потребителями ваших сервисов: *SLA, не соответствующие нуждам потребителя, нельзя назвать релевантными*. Однако при этом приложите все силы к тому, чтобы можно было *использовать одни и те же SLA для всех потребителей*. Очень желательно, чтобы у вашего сервиса был один набор SLA, удовлетворяющий нуждам всех потребителей. Наличие набора SLA, индивидуальных для каждого потребителя, значительно усложняет вам жизнь и не приносит никаких преимуществ.

Указывайте в SLA только то, что вы реально можете отслеживать и о чем по необходимости отправлять оповещения. Нет никакого толку от установления в качестве SLA какого-либо показателя, если вы не можете проверить, соответствует ли ему ваша работа. Кроме того, необходимо позаботиться об оповещениях о нарушении SLA, которые свидетельствуют о наличии в вашем сервисе каких-то проблем.

Подумайте о том, чтобы отслеживать величины, которые вы установили как свои внутренние SLA, и получать о них оповещения. Эти данные могут быть очень полезными при обнаружении и локализации проблем с сервисом, даже если они не нарушают ваши обязательства перед потребителями.

Вам пригодится информационная панель, где отображаются все ваши SLA и их текущие показатели, благодаря чему вы, бросив один взгляд, сможете узнать, все ли у вас в порядке. Полезно

Copyrighted material

19 Непрерывное совершенствование

Если ваше приложение имеет успех, вам понадобится масштабировать его, чтобы справляться с увеличивающимися объемами трафика. Это требует задействования более специфических сложных механизмов, и приложение может пострадать под тяжестью возрастающей нагрузки.

Как правило, разработчики приложения поначалу не уделяют масштабируемости достаточно внимания. Мы часто считаем, что сделали все необходимое для того, чтобы масштабировать приложение в соответствии с самыми смелыми представлениями о будущей нагрузке. Однако на практике все идет далеко не так гладко, и различного рода сбои чем дальше, тем сильнее затрудняют получение больших объемов трафика и большего количества данных.

Как мы можем улучшить масштабируемость приложения даже в том случае, если уже приближаемся к пределам его нагрузки? Очевидно, чем раньше мы возьмемся за этот вопрос в жизненном цикле приложения, тем легче его потом можно будет масштабировать. Однако в любой точке жизненного цикла можно применить множество способов, которые облегчат масштабирование приложения.

В этой главе рассматриваются несколько таких методов.

Copyrighted material

Регулярно проверяйте приложение

В частях I и II подробно раскрыта тема поддержки высокодоступного приложения и управления свойственными ему рисками. Перед тем как обдумывать методы масштабирования приложения, нужно добиться хороших показателей доступности, а также наладить управление рисками. Пока это не сделано, от остальных действий будет мало толку. Если вы не внесете изменения сейчас, заранее, то в дальнейшем в процессе масштабирования приложения обнаружите, что потеряли контроль над его работой. Начнут появляться совершенно неожиданные проблемы, которые приведут к перебоям и потерям данных, а также значительно повлияют на возможность развития приложения. К тому же по мере того, как увеличиваются трафик и количество данных, масштаб этих проблем также возрастет. Поэтому прежде всего добейтесь высокого уровня доступности и обеспечьте управление рисками, а затем переходите к чему-то другому.

Микросервисы

В части III рассказывалось об архитектуре, основанной на сервисах и микросервисах. Несмотря на то что вам придется принимать множество различных архитектурных решений и рассматривать множество архитектурных направлений, как можно раньше откажитесь от монолитной или полимонолитной архитектуры и перейдите к какой-либо форме сервисно-ориентированной архитектуры.

Владение сервисами

Если вы переходите на сервисно-ориентированную архитектуру, выбирайте модель, в которой предусмотрено владение сервисами и отдельные команды разработки отвечают за все аспекты работы своих сервисов. Такая монополия улучшит возможность масштабирования приложения до подходящего размера с точки зрения сложности кода, объема трафика и размеров массива данных. О владении сервисами более подробно рассказывается в главе 15.

Copyrighted material

должны храниться в других серверах и хранилищах, поближе к тем сервисам, которым они нужны.

Локализация данных предоставляет вам следующие преимущества.

- *Уменьшение размера отдельных массивов данных*. Поскольку ваши данные разделены по массивам, каждый массив сам по себе невелик. Это означает снижение взаимодействия с данными, в результате чего масштабировать массив становится легче. Это называется *функциональным разбиением*. Вы разделяете данные, базируясь на их функциях, а не на размере всего массива.
- *Ограниченный доступ*. Обычно, получив доступ к данным, размещенным в базе или хранилище данных, вы видите всю имеющуюся информацию в пределах одной или нескольких записей. Но для большинства операций, как правило, не нужно так много данных. Используя несколько массивов данных уменьшенного размера, вы снижаете количество ненужных данных, поступающих с вашими запросами.
- *Оптимизация методов доступа*. Разделение данных на несколько массивов позволит выбрать оптимальный тип хранилища данных для каждого массива. Например, конкретному массиву может потребоваться реляционное хранилище данных, а может быть достаточно обычного хранилища типа «ключ — значение».

Партиционирование данных

Под *партиционированием* данных может подразумеваться множество вещей. В данном случае я имею в виду разделение данных определенного типа на партиции (сегменты) в соответствии с определенными параметрами. Это часто делается с целью обеспечения работы нескольких баз данных для хранения больших или высокочастотных по запросам массивов данных, с чем не справится одиночная база данных.

Существуют и другие типы партиционирования данных (например, упомянутое функциональное партиционирование), однако в этом разделе информация сфокусирована на партиционировании по ключевым параметрам.

Copyrighted material

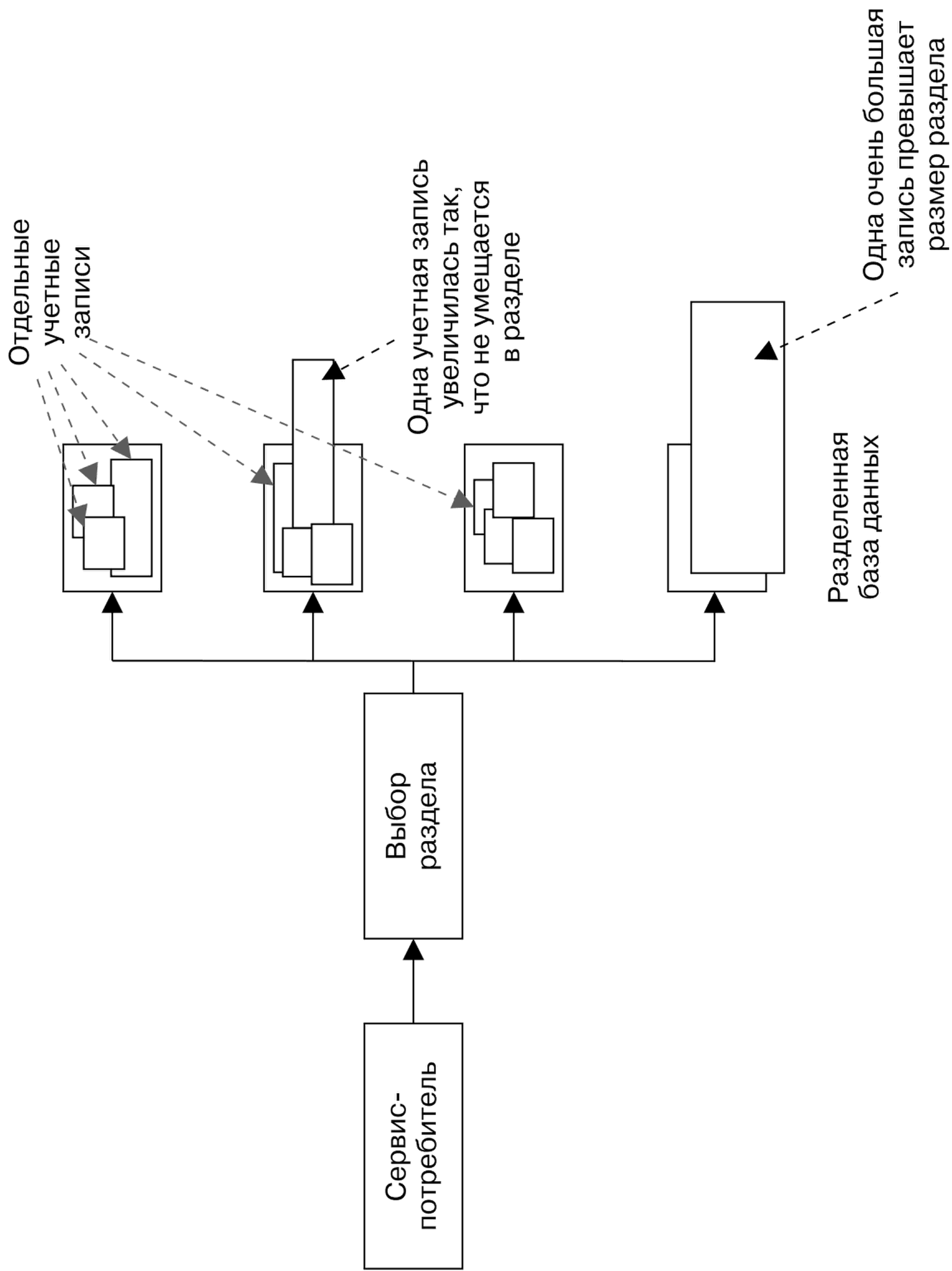


Рис. 19.2. Пример превышения размерами учетных записей возможностей своих партиций

Copyrighted material

Самым лучшим критерием партиционирования будет такой параметр, который обеспечит максимально равномерное разделение данных по размерам. Увеличение разделов в размерах должно быть независимым и равномерным настолько, насколько это возможно (рис. 19.3). При этом перепартиционирование может понадобиться только в том случае, если все разделы равномерно увеличились и стали слишком велики, чтобы партиционирование могло их правильно обработать.

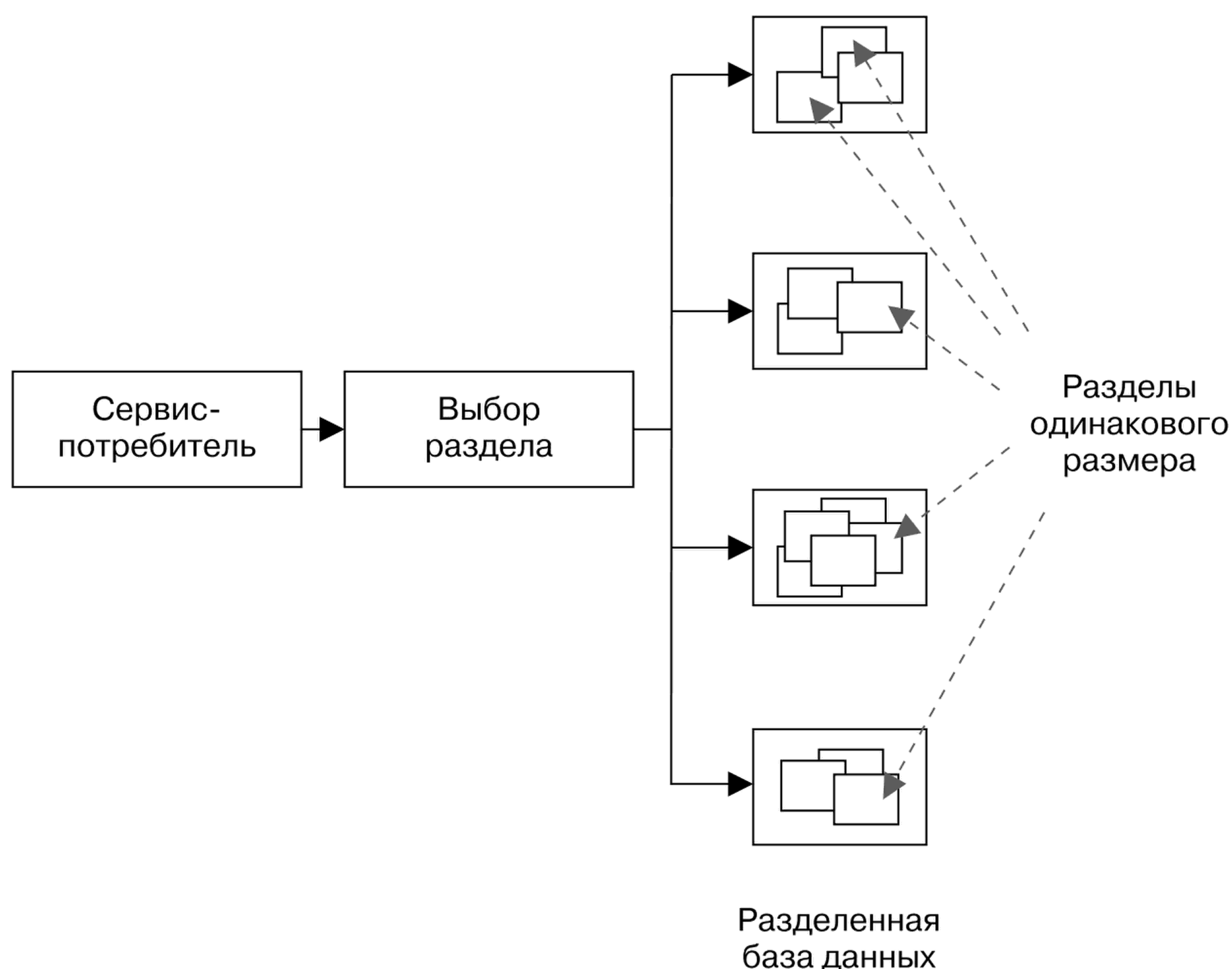


Рис. 19.3. Пример равномерного разбиения базы данных на разделы одинакового размера

Эффективной схемой партиционирования является та, что основана на выборе критерия, позволяющего получить значительное количество небольших элементов. Затем эти маленькие партиции можно отправить в более крупные партиционированные базы данных. Позднее, если потребуется перепартиционирование, вы можете

просто обновить распределение и отправить отдельные небольшие элементы в новые разделы, при этом выполнять полное перепартиционирование для всей системы не нужно.

Значение непрерывного совершенствования

Большинство современных приложений сталкиваются с ростом обрабатываемого ими трафика, размеров и сложности самого приложения, а также с увеличением количества людей, работающих над приложением.

Очень часто мы склонны игнорировать тревожные сигналы об этих изменениях, но рано или поздно болезни роста достигнут определенного уровня, когда неизбежно придется решать проблемы. Но к этому моменту, как правило, становится слишком поздно: проблема разрослась до угрожающих размеров и большинство простых техник, способных облегчить ее решение, применить невозможно. Поэтому нужно задумываться о перспективе роста приложения раньше, чем оно дорастет до проблемных размеров. Таким образом вам удастся предупредить большинство проблем, а также безопасно справиться с болезнями роста при разработке и развитии приложений.

Copyrighted material

Часть V
Облачные сервисы

Прогноз погоды: «Облачность будет увеличиваться...»

Copyrighted material

20 Облака и перемены в них

Облачные вычисления раз и навсегда изменили наши представления о разработке и запуске приложений. Но по мере того, как менялось наше видение построения приложений в облаках, менялись и сами облака, а также наши представления о них.

Что изменилось в облаках

В последнее десятилетие облако претерпело значительное развитие. Облачные провайдеры теперь предлагают гораздо более продвинутые продукты. Они предоставляют сейчас не только возможность хранения файлов и вычислительные мощности. AWS предлагает 50 уникальных сервисов для обеспечения широкого ряда вычислительных нужд.

Какие же изменения в облаке наиболее интересны для нас и наших приложений? Далее мы обзорно рассмотрим их.

Согласованность с сервисно-ориентированными архитектурами

Как уже говорилось, популярность архитектур, основанных на сервисах и микросервисах, значительно возросла в последние годы.

Copyrighted material

Перевод приложений на какую-то форму сервисно-ориентированной архитектуры становится стандартной техникой уменьшения технического долга и упрощения поддержки приложений.

Поскольку компании, перенося свои приложения в облака, стараются думать наперед, это действие становится частью полноценной стратегии модернизации продукта, которая может включать в себя переход к ультрасовременным типам архитектуры приложений. В последние годы самые современные типы архитектуры подразумевают использование микросервисов и других сервисно-ориентированных архитектур в рамках стратегии модернизации. Это объясняется тем, что технологии наподобие Docker сделали доступными для использования архитектуры, основанные на микросервисах, при разработке приложений.

Учитывая это, облачные провайдеры начали предоставлять качественно организованные предложения наподобие EC2 Container Service для управления контейнерами, основанными на микросервисах.

Небольшие специализированные сервисы

По мере модернизации наших приложений и перемещения их в облако мы начинаем внимательнее присматриваться к облачным сервисам и их расширениям, которые могут быть полезны для приложений. Возможности, ранее обеспечиваемые внутри приложений, теперь размещаются в облаке.

Крупное облако может предоставить такие возможности, как базы данных, сервисы кэширования, сервисы диспетчеризации и логгирования, сеть доставки контента (Content Delivery Network, CDN), а также перекодировка.

Все внимание приложению

Появление облака освободило ресурсы, предназначенные для создания инфраструктуры, необходимой для запуска приложений, и управление ею, и позволило уделять больше внимания критическим

аспектам приложения и его среды. Взяв на себя часть функций по управлению приложением, облако позволило нам заняться самыми важными аспектами запуска приложений.

Микростартапы

Благодаря облаку стало возможно существование совсем маленьких стартапов, которые зачастую организует и которыми управляет всего один человек. Это произошло, потому что в облаке существуют недорогие и масштабируемые технологии вычислений и другие возможности.

Для отдельных людей, которых осенила какая-то идея, как никогда просто стало реализовать ее и даже получить прибыль. Возможность создания компьютерной экосистемы без необходимости вкладывать средства в дорогую инфраструктуру позволяет новым, свежим идеям быстрее проникать на рынок. В частности, мобильные приложения и онлайн-игры получают от этого огромную выгоду.

Эти стартапы быстро выводят приложения в онлайн, и в случае как успеха, так и провала вложения невелики. Для тех, кому повезло, облако предоставляет возможность легко и с небольшими затратами выполнить масштабирование, благодаря чему компании могут вкладывать в инфраструктуру лишь столько, сколько требуют нужды бизнеса. Теперь запускать небольшие стартап-компании и управлять ими намного проще с финансовой точки зрения.

Безопасность и обеспечение законности

На ранних этапах развития облаков проблемы обеспечения безопасности были одними из основных причин, из-за которых компании отказывались переносить свои приложения в облака. Зная о необходимости обеспечения безопасности, облачные провайдеры сейчас предоставляют лучшие возможности в этой области. Облачные компании предлагают также гарантии безопасности в виде исполнения нормативных требований, таких как PSI, SOC или HIPPA.

Copyrighted material

Благодаря этим изменениям, а также заметным улучшениям высокоуровневого обеспечения безопасности компании больше не рассматривают вопросы безопасности как препятствие для перемещения в облако.

Изменения продолжаются

Перемены неизбежны. Облачные вычисления изменили наш взгляд на разработку и запуск приложений: мы начали создавать небольшие узкоспециализированные сервисы, узнали, как справляться с большим количеством данных, меньше фокусируемся на инфраструктуре приложений и больше сосредотачиваемся на них самих. Небольшие компании получили шанс на развитие, неся свежие новые идеи и взгляды в наш мир. А стандарты безопасности стали привычны во всем, что мы делаем.

Облачный сервис развивается, мотивируя нас использовать его и взаимодействовать с ним все чаще. Вероятно, эта тенденция будет развиваться, и мы должны постоянно адаптироваться для того, чтобы идти в ногу со временем. Только тогда наши приложения продолжат развиваться и расширяться.

Хорошо это или плохо, но облако изменило и продолжает изменять всех нас.

Copyrighted material

21 Распределение облака

Все мы знаем, насколько выгодно распределять данные приложения между несколькими дата-центрами. Такой же принцип применим и к самим облакам. Если мы помещаем фрагменты приложений или приложения целиком в облако, нужно проследить, *где* именно они будут расположены. Размещение приложений в облаке так же важно, как и размещение в обычных дата-центрах. В особенности это важно для масштабирования.

Однако узнать, *насколько* распределено ваше приложение в облаке, не так-то легко. Намеренно увеличить степень распределения приложения в облаке еще сложнее. Некоторые компании — провайдеры облаков не предоставляют подробную информацию о том, где географически запускается ваше приложение.

Крупные провайдеры наподобие AWS не скажут вам точно, где именно запускается приложение, но по крайней мере дадут достаточно возможностей сделать выводы об этом самостоятельно. Зная архитектурное устройство AWS, можно интерпретировать и понять эту информацию, а также извлечь из нее определенные преимущества.

Архитектура AWS

Для начала обсудим некоторые понятия, используемые внутри экосистемы AWS.

Copyrighted material

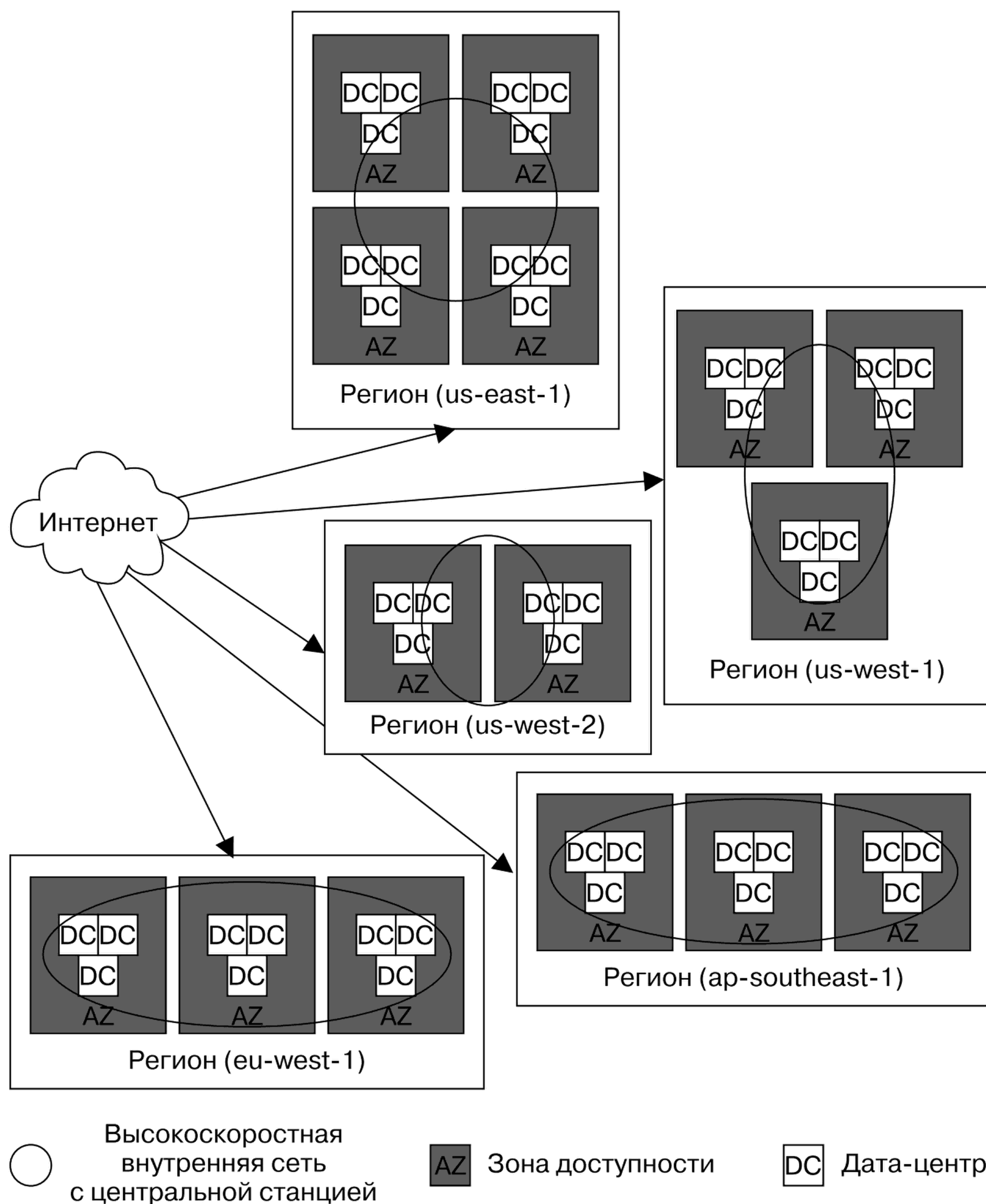


Рис. 21.1. Архитектура дата-центра AWS

Отдельный регион AWS состоит из одной или нескольких зон доступности AWS. Сами зоны внутри одного региона соединены через высокоскоростную внутреннюю сеть с центральной станцией (рис. 21.2). В результате обеспечиваются доступ друг к другу

любых двух серверов внутри региона и, как следствие, их равная производительность без участия зоны доступности, в которой они расположены. Данная ЗД состоит из двух или более дата-центров в зависимости от своего размера.

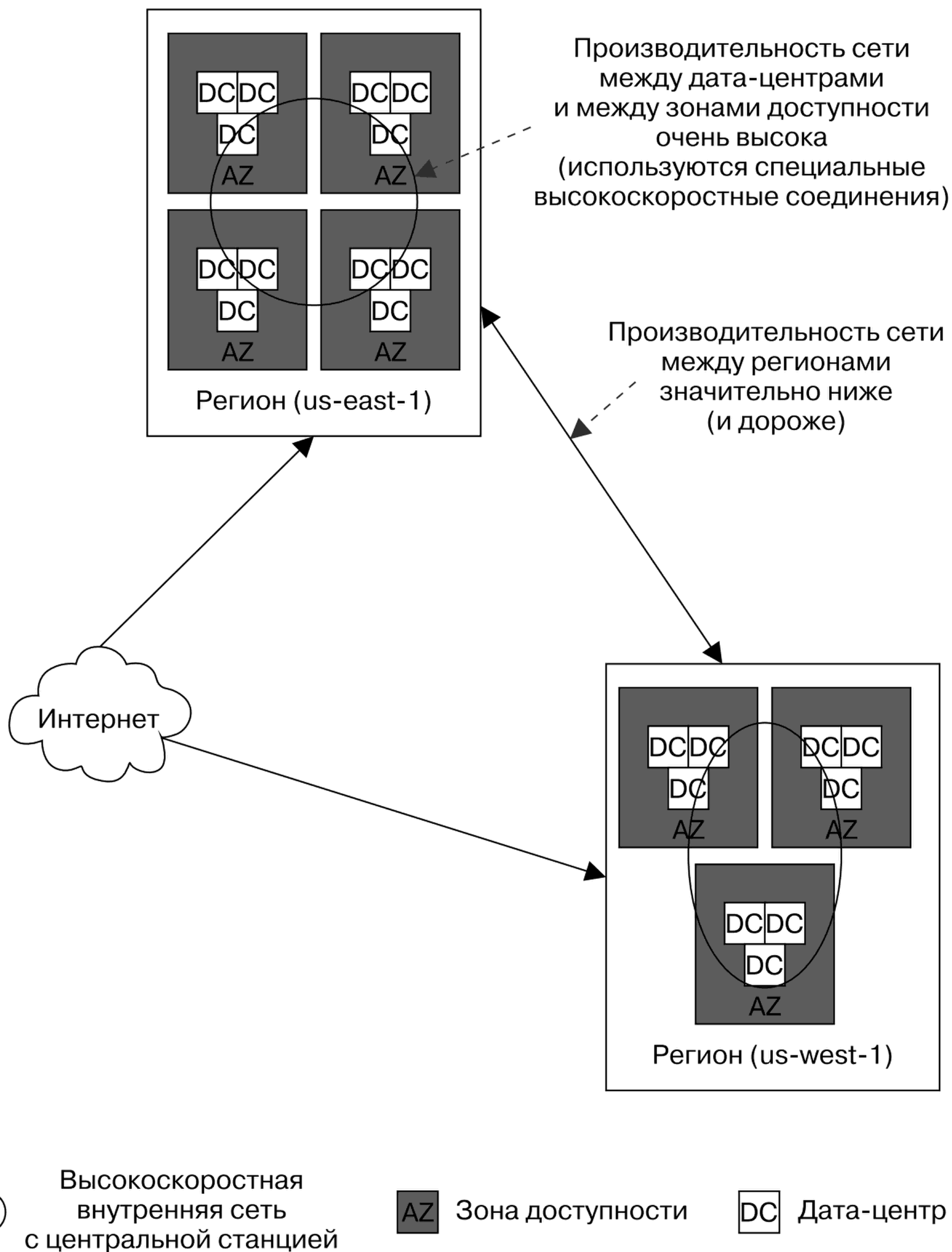


Рис. 21.2. Производительность сети региона AWS и зоны доступности

Copyrighted material

Как можно видеть на данной схеме, сетевая топография служит для упрощения разработки приложения внутри отдельного региона, но при этом распределенного по зонам доступности. Такое распределение необходимо для того, чтобы иметь возможность в случае аварии подключить резервную систему, что может быть актуально для отдельных дата-центров. А для компонентов очень важно быть напрямую связанными друг с другом посредством высокоскоростного соединения, не принимая во внимание зону, в которой они находятся.

Регионы организованы так, чтобы все приложение могло находиться внутри одного региона и не было необходимости в организации высокоскоростного канала связи с компонентами, располагающимися в другом. Если приложение все-таки нужно разместить в нескольких регионах, то, как правило, в работу запускаются несколько его независимых копий, каждая в своем регионе. Таким образом несколько географических регионов получают доступ к приложению локально, без использования дорогостоящих международных каналов связи (рис. 21.3). Данная схема поддерживается моделью расчета стоимости трафика сети AWS, которая, как правило, позволяет организовать бесплатный трафик между зонами доступности в пределах одного региона, в то время как трафик между несколькими регионами или из региона в Интернет оплачивается соответствующим образом.

Такая архитектура выбрана не только по соображениям стоимости, но и из-за величины задержек (сетевая задержка между регионами намного больше, чем между зонами доступности). Кроме того, эта структура позволяет приложению соответствовать законодательству различных стран (примером может служить зона безопасности Европейского союза)[1].

---

[1] Зона безопасности Европейского союза (EU Safe Harbor) — набор принципов приватности, регулирующих передачу информации о гражданах ЕС на территории вне ЕС. Место, где расположены данные, может иметь значение из-за необходимости соответствовать местным законам, и регионы AWS дают возможность разработки приложений при выполнении этих требований.

Copyrighted material

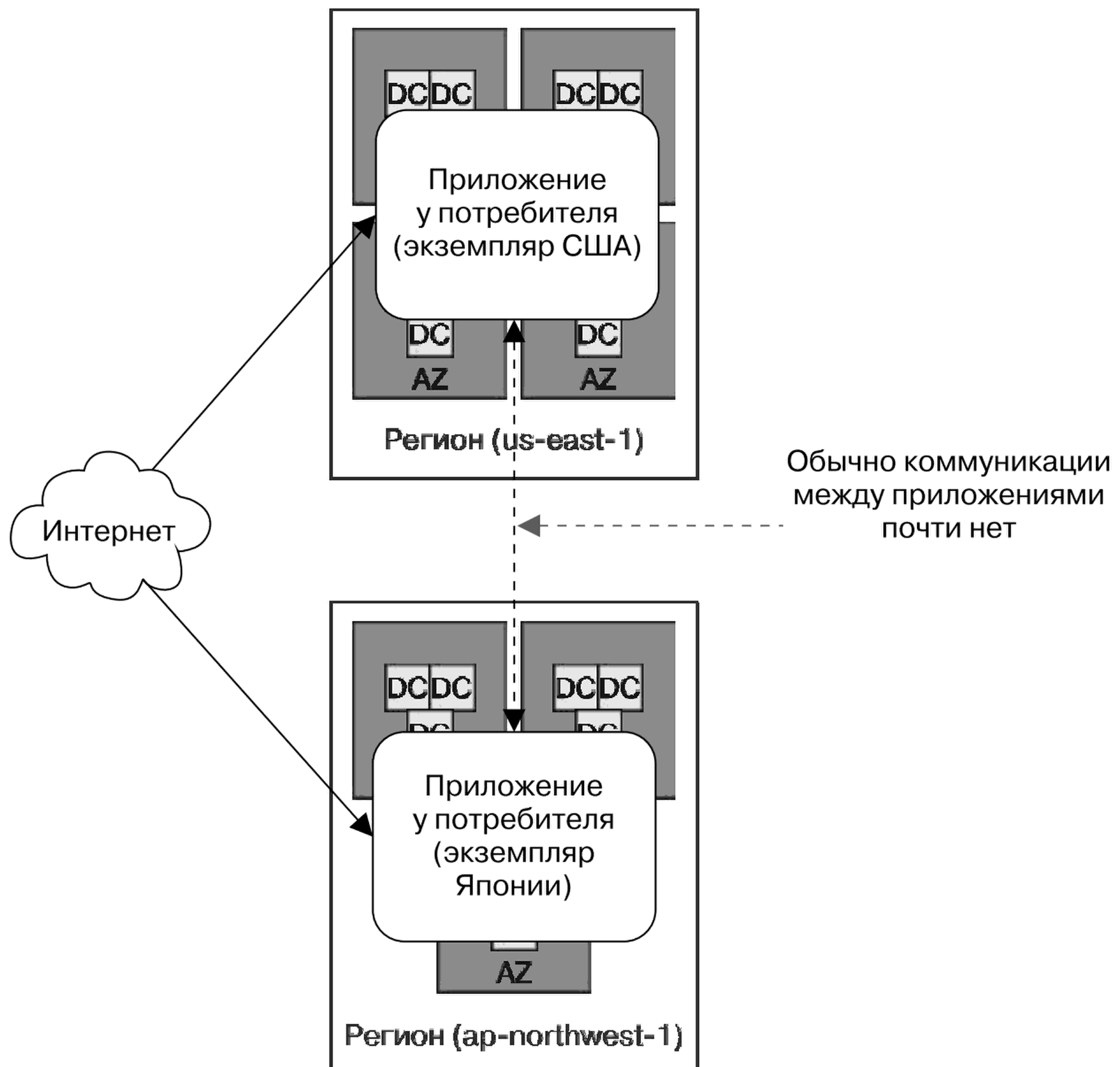


Рис. 21.3. Клиентская архитектура

Зоны доступности и дата-центры не одно и то же

Можно смело считать, что внутри одной учетной записи экземпляры EC2, расположенные в разных зонах доступности (допустим, us-east-1a и us-east-1b), находятся в различных дата-центрах.

Но это не обязательно так, если вы используете несколько учетных записей AWS. Если вы создадите экземпляр EC2 в учетной записи 1, находящейся в зоне доступности us-east-1a, и экземпляр EC2 — в учетной записи 2, находящейся в зоне доступности us-east-1b, физически

эти два экземпляра вполне могут оказаться на одном и том же сервере в одном дата-центре.

Почему так происходит? Дело в том, что названия зон доступности не строго соответствуют определенным дата-центрам. Напротив, дата-центр для зоны доступности us-east-1a в одной учетной записи может быть иным, чем дата-центр для us-east-1a в другой учетной записи.

Когда вы создаете учетную запись AWS, приложение в случайном порядке генерирует схему соответствия названий зон доступности определенным дата-центрам[1]. Это значит, что us-east-1a для двух различных учетных записей может размещаться в совершенно разных местах (табл. 21.3). Сначала мы рассматриваем случайный список дата-центров, произвольно пронумерованных от 1 до 8, внутри одного региона, а затем приводим возможное соответствие между этими дата-центрами и названиями зон доступности для четырех воображаемых учетных записей.

Таблица 21.3. Карта случайного распределения зон доступности

Дата-центр	Учетная запись AWS 1	Учетная запись AWS 2	Учетная запись AWS 3	Учетная запись AWS 4
Дата-центр 1	us-east-1a	us-east-1d		us-east-1e
Дата-центр 2	us-east-1a	us-east-1c	us-east-1a	us-east-1a
Дата-центр 3	us-east-1b	us-east-1a	us-east-1d	us-east-1d
Дата-центр 4	us-east-1c		us-east-1a	us-east-1b
Дата-центр 5	us-east-1d	us-east-1b	us-east-1c	us-east-1c
Дата-центр 6	us-east-1e		us-east-1b	
Дата-центр 7			us-east-1e	
Дата-центр 8		us-east-1e		

[1] Разумеется, на самом деле порядок не случаен, а определен специальным алгоритмом. Кроме того, фактически эта схема не определяется до тех пор, пока в конкретной учетной записи не потребуется использовать конкретную зону доступности или регион.

Copyrighted material

Из этой таблицы можно сделать несколько выводов. Во-первых, какая-то зона доступности для одного из аккаунтов фактически может быть распределена по разным дата-центрам. Это значит, что два разных экземпляра EC2, которые вы создаете в одной учетной записи и в одной зоне доступности, могут размещаться на одном физическом сервере, а могут — в разных дата-центрах. Во-вторых, два экземпляра EC2, созданных в разных учетных записях, могут быть, а могут и не быть в одном и том же дата-центре, даже если зоны доступности будут разными.

Например, если в учетных записях 1 и 3 из табл. 21.3 будет создано по экземпляру us-east-1d, они оба окажутся в дата-центре 3. Это важно понимать по одной простой причине: *нельзя считать независимыми с точки зрения обеспечения доступности два экземпляра EC2 только потому, что они находятся в двух разных учетных записях и в двух разных зонах доступности*.

Как уже обсуждалось в частях I и II, обеспечение независимости репликационных компонентов необходимо для обеспечения доступности и управления рисками. Однако при использовании нескольких учетных записей AWS модель зон доступности может не позволить достичь этого. На модель зон доступности можно полагаться только в том случае, если вы используете лишь одну учетную запись AWS.

Почему вообще может понадобиться использовать более чем одну учетную запись AWS? На самом деле это довольно распространенная потребность. Многие компании создают несколько учетных записей AWS для использования различными группами внутри компании. Причинами могут быть управление способами оплаты, организация прав пользователей и многое другое.

Любопытно, почему они это делают?

Вы никогда не задумывались о том, почему, извещая о перебоях в работе, AWS говорит, что будут недоступны несколько зон доступности в данном регионе, но никогда не указывает, какие именно?

Причина в структуре системы. Если проблема произошла, скажем, в дата-центре 4, то с вашей точки зрения это может означать us-east-1a,

а с точки зрения другого пользователя — us-east-1c. Они не могут указать название конкретной зоны доступности, потому что эти названия свои для каждой учетной записи.

Почему же AWS использует такую странную схему соответствия? Одна из основных причин — балансировка нагрузки. Когда люди запускают экземпляры EC2, разумеется, это происходит неравномерно по всем зонам доступности. Фактически зону us-east-1a пользователи выбирают для запуска экземпляров EC2 гораздо чаще, чем us-east-1e. Это свойство человеческой натуры. Если бы в AWS не было искусственного перенаправления, зоны доступности, чьи названия находятся *в начале* алфавита, были бы постоянно перегружены, в то время как зоны *в конце* алфавита — недогружены. С помощью искусственного перенаправления компания может распределять нагрузку более эффективно.

Поддержка распределения локаций для обеспечения доступности

Можете ли вы быть уверены в том, что ресурсы AWS, которые вы запускаете, имеют достаточно резервных компонентов, гарантированно расположенных в различных дата-центрах, что обеспечит достаточную защищенность от рисков?

Есть несколько средств, которые можно применить. В первую очередь удостоверьтесь, что вы обеспечили резервные компоненты в отдельных зонах доступности внутри каждой учетной записи. Если есть резервные компоненты в нескольких учетных записях, убедитесь, что обеспечили резервы в нескольких зонах доступности внутри каждой учетной записи отдельно. Не сравнивайте зоны доступности в разных учетных записях.

Неуправляемый ресурс

Неуправляемый облачный ресурс предоставляет пользователю только базовые возможности и базовое управление. Примером неуправляемого облачного ресурса является Amazon EC2, который предоставляет основные возможности серверов в управляемой манере.

Облако обеспечивает управление базовым серверным слоем виртуализации, а также создание экземпляров сервера и их начальной файловой системы. Но после того, как экземпляр готов и запущен в работу, провайдер не имеет доступа к управлению сервером.

Облачный провайдер управляет данными, которыми обменивается экземпляр (то есть сетью), а также центральным процессором и его загруженностью. Но в то же время провайдер ничего не знает о том, что именно запущено на сервере, и не следит за тем, что там происходит.

Amazon и подобные компании намеренно так поступают. Происходящее на сервере — ваше дело, и Amazon не желает нести ответственность за любые аспекты программного обеспечения, запущенного на нем. *Зона ответственности* Amazon заканчивается на входе на виртуальный сервер.

Это влияет на самые разные аспекты. Например, рассмотрим метрики экземпляров EC2, которые Amazon собирает и предоставляет вам через Cloud Watch:

- сколько сетевого трафика получает экземпляр;
- сколько сетевого трафика генерирует экземпляр;
- сколько данных считывается с дисков;
- сколько данных записывается на диски;
- сколько ресурсов центрального процессора потребляется.

Но в этом списке отсутствуют другие полезные метрики, которые Amazon не отслеживает:

- количество свободной памяти (ему неизвестно, как используется память);

Copyrighted material

- количество свободного дискового пространства (ему неизвестны структуры данных на дисках);
- количество работающих процессов (ему неизвестны процессы, работающие на сервере);
- память, потребляемая приложениями (ему неизвестно, какое именно приложение потребляет память);
- подкачка страниц памяти (ему неизвестен механизм распределения или управления памятью);
- какие процессы потребляют больше всего ресурсов (ему неизвестно, какие именно процессы работают).

Рассмотрим еще один пример — контроль доступа к экземпляру сервера. Как правило, на сервере могут работать следующие виды контроля доступа:

- *контроль списка сетей, имеющих доступ* (Access Control List, ACL) — контроль доступа, осуществляющийся на уровне сети (брандмауэра);
- *идентификация пользователя* — контроль доступа путем идентификации пользователей, которые могут, авторизовавшись, получить доступ к возможностям сервера.

Amazon управляет сетевым контролем доступа на всем протяжении существования экземпляра сервера (в Amazon это называется security groups, то есть «группы безопасного доступа»). Вы можете изменить группу безопасного доступа ACL в любое время. Если вы блокируете доступ порта, блокировка происходит немедленно. Если позволяете доступ по порту или IP-адресу, этот порт или адрес немедленно получает доступ. Все это происходит на уровне сети: прежде чем трафик достигнет сервера в случае внутреннего ACL и после того, как трафик покидает сервер, — в случае внешнего. Доступа к программному обеспечению на самом сервере не требуется.

В случае пользовательской идентификации все происходит иначе. Прежде чем можно будет первый раз развернуть сервер, вы должны указать ключевую пару — публичный/приватный ключи доступа, которые будут установлены на экземпляр. Эта ключевая пара позволяет

Copyrighted material

платформой, включая операционную систему и программное обеспечение базы данных. Это хорошо видно по метрикам, которые CloudWatch предоставляет для экземпляров RDS. Помимо обычной информации об экземпляре EC2, вы получаете результаты отслеживания работы самой базы данных, например такие, как:

- количество выполненных подключений к базе данных;
- пространство файловой системы, которое потребляет база данных;
- количество запросов, запущенных в базе данных;
- задержка репликации.

Это метрики, доступные только из операционной системы или самого ПО, управляющего базой данных.

Другой способ оценить это отличие — рассмотреть типы конфигурации, которые вы можете выполнить. Вам доступна конфигурация не только базовых параметров сервера (сетевые соединения и подключенные диски), но и параметров самой базы данных, например максимального количества соединений, кэширования информации и других конфигурационных или настроечных параметров.

И наконец, управление обновлением ПО теперь находится на уровне облачного слоя. Если ПО, управляющему базой данных, необходимо обновление, его выполняет для вас сама Amazon.

Управляемые несерверные ресурсы

Управляемый несерверный ресурс — это ресурс, который предоставляет вам какую-то конкретную возможность, но не открывает серверную инфраструктуру, с помощью которой эта возможность работает.

Отличный пример управляемого несерверного ресурса — Amazon S3. Этот сервис предоставляет возможности хранения и передачи файлов в облаке. Если вы храните файл в S3, то взаимодействуете напрямую с сервисом S3. В данном случае нет какого-либо отдельного сервера или серверов, выделенных под ваши нужды; напротив, один

или несколько серверов[1] незаметно для вас выполняют запрошенные вами действия.

Вся операция является управляемой, но вы видите только интерфейс ПО, предоставляемый сервисом (в случае S3 это загрузка, выгрузка, удаление файлов и пр.), и имеете возможность взаимодействовать с ним. Вы не можете ни видеть операционную систему или запущенные в ней сервисы, ни управлять ими. Эти серверы распределены между всеми пользователями сервиса, поэтому ими управляет Amazon и вам нет необходимости вмешиваться.

Особенности использования управляемых ресурсов

Если вы используете управляемый сервис или его часть, это дает вам как пользователю сервиса некоторые преимущества.

- Вам не нужно устанавливать или обновлять ПО управляемой системы.
- Вам не нужно настраивать или оптимизировать систему (но в некоторых случаях вы можете сделать это, используя возможности облачного провайдера).
- Вам не нужно отслеживать и проверять работу ПО, убеждаясь, что оно функционирует как надо.
- Если нужно, облачный провайдер может представить вам результаты отслеживания, при этом нет необходимости подключать дополнительные возможности или дополнительное ПО.
- Облачный провайдер может позволить выполнить репликацию или восстановление сервиса.

У управляемых компонентов наряду с преимуществами существуют и недостатки.

- Как правило, у вас нет возможности внести существенные изменения в то, как программное обеспечение выполняет ваши операции.

[1] В случае Amazon S3 — очень много серверов.

- У вас нет возможности контролировать, когда и как обновляется программное обеспечение, а также какая именно версия ПО запущена в данный момент[1].
- При мониторинге и конфигурировании сервиса вы ограничены возможностями, предоставленными облачным провайдером.

Особенности использования неуправляемых ресурсов

Если сервис или часть сервиса, которым вы пользуетесь, не является управляемым, вы как пользователь также получаете некоторые преимущества. Вот некоторые из них.

- Можно контролировать, какое ПО запущено на сервисе, какая версия работает и как ПО настроено.
- Вы выбираете, как и когда выполнять обновление и выполнять ли вообще.
- Вы можете отслеживать и контролировать работу ПО как вам угодно, используя любые подходящие механизмы.

При этом использование неуправляемых компонентов обладает следующими недостатками.

- За все надо платить: вы несете полную ответственность за все управление и обслуживание, необходимые системе.
- Вы должны следить за своевременным резервным копированием и репликацией данных.
- Вы должны отслеживать работу своего программного обеспечения, чтобы убедиться, что оно функционирует верно, — никто иной не оповестит вас в случае сбоя.

---

[1] Тем не менее облачный провайдер может предоставить вам некоторые возможности. Например, RDS предоставляет ряд версий баз данных, которые он поддерживает, хотя не все они доступны. А в управляемых системах наподобие S3 вы вообще не можете контролировать процесс обновления.

Copyrighted material

- Если программное обеспечение вышло из строя, вы и только вы ответственны за исправление. Облачный провайдер ничем вам не поможет.

CloudWatch и мониторинг

Меня часто спрашивают, достаточно ли CloudWatch для мониторинга облачных ресурсов, и я снова и снова отвечаю, что нет.

Почему? Дело в том, что CloudWatch предоставляет возможности мониторинга только для тех компонентов, которыми управляет Amazon. Это значит, что он обеспечивает довольно хорошее покрытие мониторинга для DynamoDB, RDS, ElastiCache и подобных им управляемых серверных ресурсов.

А вот для серверов EC2 Cloud Watch предоставляет только базовую информацию мониторинга на низком уровне сервера и виртуализации. Выше уровня серверной виртуализации никаких мониторинга или оповещений не предоставляется. Поэтому для полноценного отслеживания серверов EC2 вам нужны два дополнительных средства.

- Отслеживание сервера или операционной системы, которое наблюдает за работой системы и сервера изнутри. Благодаря ему вы можете получать информацию о распределении памяти, подкачке и использовании файловой системы.
- Система отслеживания производительности приложения, которая наблюдает за работой приложения изнутри. Таким образом вы будете знать, как функционирует приложение, как с ним работают пользователи и какие ошибки и проблемы возникают внутри.

Только с этими тремя средствами (CloudWatch и два приведенных ранее) вы можете считать отслеживание сервера EC2 полноценным.

- Если ваше приложение потребляет меньше указанного, остаток выделенного ресурса просто не используется.
- Если вашему приложению потребуется больше, чем выделено, оно будет испытывать недостаток ресурсов.
- Тщательные расчеты возможностей очень важны для точного определения необходимого вам количества этого ресурса.

Классический пример облачных ресурсов с выделением мощности — экземпляры Amazon EC2. Вы указываете, сколько серверов вам нужно и какого размера должен быть каждый из них, и облако выделяет именно столько. Кроме того, управление инфраструктурными компонентами, такими как облачные базы данных[1], используют модели выделенной мощности. В любом случае вы указываете количество необходимых вам единиц и их размер, а облачный провайдер выделяет их для вашего использования.

Но есть и другие примеры облачных ресурсов с выделением мощности, работающих несколько по-другому, например Amazon DynamoDB. Используя эту опцию, вы можете указать, какая емкость необходима для работы с таблицами в DynamoDB. В этом случае мощность рассчитывается не в *серверах*, а в *единицах потребляемой емкости*. Вы указываете, какая емкость требуется для ваших таблиц, и получаете именно столько для работы. Если вашему приложению не потребуется такая емкость, часть ее останется неиспользованной. Когда потребуется больше, приложение будет испытывать недостаток емкости, если только вы не приобретете дополнительные ресурсы. Таким образом, эти *единицы емкости* выделяются и потребляются аналогично *серверам*, притом что на первый взгляд это совершенно разные вещи.

Изменение выделенной мощности

Как правило, мощность выделяется дискретно (пользование сервером стоит вам определенной суммы денег в час; единица емкости DynamoDB также стоит какой-то суммы за час). Вы можете изменить

---

[1] Примерами могут служить Amazon RDS, Amazon Aurora и ElastiCache.

количество серверов, выделенных для вашего приложения, или количество единиц емкости, выделенных для таблиц в DynamoDB, но только ступенчато (то есть указав целое число серверов или единиц емкости DynamoDB). Хотя эти единицы могут быть разных размеров, так как доступны, например, серверы разного размера, всякий раз вам приходится выбирать целое число единиц.

Поэтому ответственность за наличие достаточного количества ресурсов лежит на вас. Можно проделать тщательные расчеты, аналогичные тем, что выполняются для серверов в традиционных дата-центрах. Вы можете также тщательно распределить мощность, основываясь на величине ожидаемой нагрузки, и руководствоваться полученным числом, пока не придет время пересмотреть положение вещей и определить, что ваши требования к мощности изменились. Это типичный подход для необлачного распределения серверов.

Но все-таки в облаке проще изменить распределение ресурсов, чем в традиционном дата-центре. Поэтому для расчетов можно использовать другие алгоритмы. Например, поскольку изменения выделенной мощности могут быть сделаны практически мгновенно, вы можете подождать, пока не выработаете почти всю имеющуюся мощность, прежде чем примете решение о расширении.

Доступен также вариант мощности, меняющейся по определенному расписанию, основанному на предполагаемой потребности. Например, число серверов может увеличиваться в дневные часы, когда приложение активно используется, и уменьшаться ночью, когда посещаемость падает.

Еще один вариант — динамически и автоматически изменять выделенную мощность, базируясь на актуальных шаблонах использования. Вы можете, скажем, отслеживать потребление мощности процессора на своих серверах, и как только оно достигнет определенного значения, автоматически ввести в работу дополнительные серверы[1]. Можно встроить автоматические механизмы в приложение

---

[1] А также, наоборот, остановить работу серверов, как только потребление упадет ниже определенной отметки.

или инфраструктуру сервиса, а также воспользоваться преимуществами облачных сервисов, таких как AWS AutoScaling, чтобы автоматически менять выделенные вам ресурсы на основе параметров текущего использования, которые вы укажете.

Какой бы механизм определения и изменения мощности вы ни выбрали, важно отметить, что в любой момент времени располагаете лишь той мощностью, которая выделена вам именно сейчас. Вы всегда можете столкнуться с тем, что часть выделенной (и оплаченной) мощности простаивает без дела или, что еще хуже, необходимых ресурсов не хватает, так как мощности недостаточно. Даже если вы используете автоматическую схему выделения, такую как AutoScale, которая в случае необходимости немедленно обеспечивает вашему приложению дополнительную мощность, это не значит, что используемый этим сервисом алгоритм обнаружит недостаток ресурсов прежде, чем в вашем приложении начнутся проблемы из-за, скажем, резкого возрастания потребления какого-то определенного ресурса.

Проблемы выделения мощности

Рассмотрим Amazon's Elastic Load Balancer (ELB) — гибкий балансировщик нагрузки.

Этот сервис обеспечивает балансировку нагрузки для вашего приложения, чтобы оно автоматически подстраивалось и справлялось с любым количеством трафика, который был ему направлен. Если вы получаете очень мало трафика, ELB уменьшит количество серверов, которое вы задействуете, и их мощность. Если получаете много трафика, ELB снова сбалансирует приложение, введя в работу больше серверов большей мощности. Все это автоматизировано и абсолютно прозрачно для вас как владельца приложения. Таким образом, предоставляя балансировщик нагрузки по сравнительно низкой начальной цене, ELB позволяет автоматически справляться с огромными количествами трафика (с соответствующим увеличением стоимости). Так вы экономите деньги, когда трафик мал, и в то же время можете подстраиваться под более высокие уровни трафика, если возникает такая необходимость.

Резервированная мощность

Как правило, вы можете менять выделенную мощность с любой нужной вам частотой[1], понижая или повышая ее в соответствии со своими потребностями. В этом заключается одно из преимуществ облака. Если вам нужны 500 серверов в один час и только 200 — в следующий, вы и заплатите за 500 серверов в первый час и за 200 — во второй. Это ясно и логично. Однако за эту почти идеальную гибкость количества выделенной мощности, которую вы потребляете, вам придется платить премиум-тариф.

Но что, если ваши нужды более стабильны? Что, если вам *всегда* достаточно 200 серверов? Зачем же тогда платить за возможность ежечасного гибкого изменения количества серверов?

Настало время поговорить о *резервированной мощности*. Резервированная мощность означает, что вы обязуетесь потребить определенное количество мощности у вашего провайдера за определенный период времени (например, от одного до трех лет), а за это получаете выгодный тариф.

Пример 23.1. Использование резервированной мощности

Резервированная мощность не ограничивает вашу гибкость в использовании выделенных ресурсов, она лишь гарантирует провайдеру, что вы потребите определенное количество ресурсов.

Допустим, у вас есть приложение, которое обычно требует работы 200 серверов. Но иногда трафик резко возрастает. В эти короткие периоды требуются 500 серверов. Вы можете применять AutoScale, чтобы динамически увеличивать количество серверов, в результате чего использование серверов будет варьироваться от 200 (минимум) до 500 (максимум).

Поскольку вы всегда будете задействовать как минимум 200 серверов, то можете оплатить их резервированную мощность. Допустим,

[1] Иногда вы можете столкнуться с ограничениями: например, в DynamoDB допустимая частота изменений выделенной мощности не может превышать определенного значения.

вы оплачиваете работу 200 серверов на один год. Вы будете платить за 200 серверов постоянно, хотя и по более низкому тарифу. Это нормально, так как эти серверы будут работать постоянно. За дополнительные 300 серверов (500 – 200) вы станете платить по обычному, то есть более высокому, почасовому тарифу, но лишь за то время, когда будете пользоваться этими серверами.

Резервированная мощность позволяет вам получать скидку в обмен на обязательство пользоваться данными ресурсами[1].

Ресурсы с расчетом задействования

Ресурсы с расчетом задействования — это облачные ресурсы, которые не выделяются заранее, а просто потребляются в том количестве, которого потребует ваше приложение. Вы оплачиваете ровно столько ресурсов, сколько задействовали. Никакого выделения не требуется.

Вы можете узнать ресурсы с расчетом задействования по следующим признакам.

- Этап выделения мощности отсутствует, и, следовательно, никаких предварительных расчетов необходимой мощности не требуется.
- Если приложению нужно меньше ресурсов, вы используете меньше ресурсов и, соответственно, меньше платите.
- Если приложению нужно больше ресурсов, вы используете больше ресурсов и, соответственно, больше платите.
- Вы можете перейти от очень маленького к очень большому количеству потребляемого ресурса без каких-либо действий

---

[1] Использование резервированной мощности также гарантирует, что определенный тип ресурса будет доступен в конкретной необходимой вам зоне, когда это потребуется. Не имея резервированной мощности, вы можете столкнуться с ситуацией, когда AWS не сможет выполнить ваш запрос на определенный тип ресурса в определенной зоне доступности.

Copyrighted material

по масштабированию своего приложения или облачного ресурса. Если, конечно оно потребляет его в разумных пределах.

- Разумные пределы из предыдущего пункта определяются исключительно облачным провайдером и его возможностями.

Как правило, вы не можете узнать, как именно выделяются или масштабируются эти ресурсы, этот процесс целиком скрыт от вас.

Классический пример ресурсов с расчетом взаимодействия — Amazon S3. Работая с ним, вы оплачиваете только то количество данных, которое храните или пересылаете с помощью этого сервиса. Вы не должны заранее планировать количество данных, которое будете хранить, или необходимую пропускную способность. Вам в любое время будет доступно необходимое количество ресурсов (с учетом возможностей системы), и вы будете оплачивать только то, что использовали фактически.

Магия ресурсов с расчетом задействования

Управлять и масштабировать сервисы с расчетом задействования очень легко, так как не требуется никакого предварительного планирования. Это возможно благодаря мультитенантной природе облачных сервисов.

У таких сервисов, как Amazon S3, имеется огромное количество дискового пространства и большое число серверов, которые вводятся в работу по мере поступления индивидуальных запросов от отдельных пользователей. При появлении в вашем приложении пиковых нагрузок соответствующие ресурсы будут выделены из общей *совокупности доступных ресурсов* сервиса.

Доступные ресурсы сервиса являются общими для всех пользователей, поэтому их общее количество очень велико. В то время как в вашем приложении потребность в ресурсе падает, в чьем-то еще может случиться пиковый рост нагрузки и ненужные вам ресурсы будут выделены ему. Этот процесс абсолютно прозрачен.

Поскольку количество доступных ресурсов очень велико, сервис может обработать *все* запросы и все пиковые нагрузки у всех пользователей и ни один из них не столкнется с нехваткой ресурсов. Чем

Copyrighted material

крупнее сервис, то есть чем больше у него пользователей, тем больше способность провайдера покрыть все пиковые нагрузки и запасти достаточно мощности для всевозможных пользовательских нужд.

Эта модель работает до тех пор, пока какой-либо отдельный пользователь не станет потреблять значительную долю общих ресурсов, доступных у облачного провайдера. Если появляется крупный клиент, которому требуется довольно много ресурсов провайдера, он может столкнуться с нехваткой ресурсов во время пиковых нагрузок, а также повлиять на количество мощности, доступной другим клиентам. Впрочем, масштаб таких сервисов, как Amazon S3, настолько велик[1], что ни один крупный клиент не способен потребить значительную долю его ресурсов, поэтому выделение ресурсов в S3 остается *магическим*.

И все же пределы возможностей Amazon S3 существуют. Если вы запустите приложение, которое пересылает или хранит на сервисе значительное количество данных, то можете столкнуться с ограничениями, которые устанавливает S3 для того, чтобы предотвратить нехватку ресурсов у других пользователей. В результате крупный потребитель S3 может достичь этих искусственных пределов и сам почувствовать нехватку ресурсов, но это происходит только тогда, когда речь идет о масштабах хранения и пересылки данных, измеряемых петабайтами.

Но даже если вы потребляете ресурсы S3 в таких огромных количествах, есть способы их перемещения, которые позволяют снизить влияние ограничений. Кроме того, вы можете связаться с Amazon и попросить о расширении пределов. Компания может раздвинуть рамки в конкретных необходимых вам областях, а затем учитывать эти увеличившиеся ограничения в своих планах по обеспечению мощностей и их распределению, чтобы нужды всех пользователей своевременно удовлетворялись.

[1] Согласно последним данным, опубликованным Amazon, которые мне удалось найти, в 2013 г. в S3 хранилось 2 трлн объектов. Это по пять объектов на каждую звезду Млечного Пути. См.: Amazon S3 — Two Trillion Objects, 1.1 Million Requests / Second, AWS Official Blog, 18 апреля 2013 г. https://aws.amazon.com/blogs/aws/amazon-s3-two-trillion-objects-11-million-requests-second/.

Copyrighted material

Преимущества и недостатки технологий распределения ресурсов

Как видно из табл. 23.1, все технологии, которые мы обсудили, имеют свои преимущества и недостатки.

Таблица 23.1. Сравнение технологий распределения облачных ресурсов

Параметр	Выделенная емкость	На основе использования
Примеры сервисов (Amazon AWS)	EC2, ELB, DynamoDB	S3, Lambda, SES, SQS, SNS
Требует планирования потребления	Да	Нет
Основа расчета оплаты	Выделенная мощность	Потребленная мощность
Недостаточная загрузка	Остаток мощности простаивает	—
Чрезмерная загрузка	Приложение испытывает нехватку ресурсов	—
Возможность резервирования мощности с целью экономии	Да	Нет
Возможность масштабирования мощности	Выделение изменяется вручную или с помощью скриптов, возможны задержки	Автоматически и немедленно
Обработка пиковых нагрузок	Возможна нехватка ресурсов при пиковых нагрузках или резком возрастании потребности в ресурсах	Прозрачное регулирование
Дополнительная мощность	Закреплена за вами для последующего использования	Общая совокупность ресурсов доступна для использования всеми пользователями

Copyrighted material

24 Другие средства масштабирования

Настройка сервисов, их конфигурирование, управление ими и последующее развертывание — это лишь один из способов доставки крупномасштабных приложений. Есть и другие, каждый из которых имеет свои преимущества и недостатки. В этой главе мы обсудим различные альтернативные варианты от облачных серверов до AWS Lambda, рассмотрим их сходство и различия, а также выбор самого подходящего из них для ваших нужд.

В этой главе мы сравним различные облачные[1] среды исполнения приложений, в частности:

- *облачные серверы* — базовая серверная технология, примером которой является Amazon EC2;
- *вычислительные слайсы* — традиционные программные приложения, запущенные в вычислительных средах, независимых от серверов. В качестве примера можно привести Heruku Dynos и Google App Engine;
- *динамические контейнеры* — обеспечивают полноценные серверные возможности в сочетании со способностью быстрого

[1] Некоторые из этих технологий применимы и в необлачных средах доставки, но поскольку мы обсуждаем масштабирование, то будем говорить только об облачных технологиях.

Copyrighted material

запуска, остановки и миграции на новые системы. Лучший пример — Docker;

- *микровычисления* — небольшие событийно-управляемые среды выполнения с большим потенциалом масштабирования. Лучший пример — AWS Lambda.

Прежде чем начать сравнивать эти методы, рассмотрим каждый из них в отдельности.

Облачные серверы

Облачные серверы — простейший способ перейти к масштабируемым вычислениям, который лучше всего соответствует традиционному программированию и архитектурным моделям систем. Облачные серверы доступны где угодно, и экземпляры Amazon EC2 — самый известный и популярный пример этой технологии.

Основное преимущество использования облачных серверов — это возможность быстро вводить их в работу онлайн и выводить из нее, что особенно важно в контексте масштабирования. Однако само приложение при этом должно быть сконструировано так, чтобы добавление дополнительных серверов в принципе могло улучшить масштабируемость. Для большинства приложений в большинстве сред это так и есть, но все же не всегда.

Преимущества

- Наиболее дешевый вариант за цикл.
- Очень мало функциональных ограничений — доступно большинство возможностей серверов.
- Непрерывная работа.

Недостатки

- Выделение мощности. Масштабирование осуществляется путем ввода в эксплуатацию нового сервера и развертывания приложения на нем.

- Вы платите за выделенную, а не за использованную мощность.
- Архитектура приложения должна поддерживать масштабирование путем добавления дополнительных серверов онлайн.
- Многие возможности физических серверов не могут использоваться эффективно в масштабированной среде (например, возможность локального хранилища).
- Владелец приложения должен управлять работой сервера и обслуживать его, в том числе обновлять программное обеспечение и средства безопасности.

Примеры оптимального использования

Облачные серверы — подходящее решение в широком спектре задач. Они могут использоваться для масштабирования в большинстве случаев.

Вычислительные слайсы

Вычислительные слайсы — это альтернативная модель выполнения, предусматривающая выполнение приложений в отсутствие знания того, на каком сервере они запускаются. Используя эту модель, необходимо развернуть программное обеспечение приложения на инфраструктуре «платформа как сервис» (Platform as a Service, PaaS), которая будет организованно исполнять стек. Этот процесс протекает без предоставления сведений о сервере, на котором работает программа.

Существует несколько примеров вычислительных движков, основанных на вычислительных слайсах, самый известный из которых — Heroku Dynos.

Преимущества

- Выделенная мощность легко меняется с относительно небольшой дискретностью.
- Провайдеры вычислительных слайсов следуют стратегии постепенного предоставления слайсов. Таким образом удается снизить

цену за отдельный слайс, что особенно важно для приложений с низким трафиком.

- Не требуется никакого управления сервером.

Недостатки

- Вычислительный цикл более дорогой, чем на облачных серверах.
- Выделяемая мощность. Масштабирование состоит в предоставлении новых слайсов для запуска вашего приложения.
- Вы платите за выделенную вам, а не за использованную мощность.
- Вы никак не можете контролировать сервер и компоненты инфраструктуры, которые используете.

Примеры оптимального использования

Традиционно эта технология подходит для приложений с низким использованием трафика. Наиболее эффективна она в том случае, если необходимо запустить традиционное приложение или сервис, но его владелец не хочет брать на себя заботу о серверах и других компонентах инфраструктуры.

Для приложений, число обращений к которым невелико и которые могут работать на небольшом количестве слайсов, слайсы могут оказаться довольно выгодны по цене. Но чем больше приложение, тем дороже они будут по сравнению с обычными серверами.

Следует отметить, что Google App Engine — еще один пример компьютерного программирования, основанного на слайсах, имеющего аналогичные преимущества и недостатки, за некоторыми исключениями: оплату они рассчитывают иначе, а выделение рассчитывается более динамично. Важно понять, что большинство преимуществ и недостатков слайс-ориентированной концепции свойственны реализации конкретного сервиса, а не общей концепции.

Динамические контейнеры

Динамические контейнеры — частный случай контейнерной технологии, которая подразумевает динамическое выделение и перемещение контейнеров между серверами для создания хорошо управляемого и способного к масштабированию вычислительной среды. Контейнеры могут использоваться как технология развертывания в любой из упомянутых вычислительных сред, но сейчас мы обсуждаем динамическую модель быстрого планирования и перемещения контейнеров для оптимизации системных ресурсов и упрощения добавления новых контейнеров.

В динамической контейнерной среде приложение, заключенное в контейнер, разрабатывается так, чтобы его можно было легко и быстро запустить, остановить или переместить. В сочетании с архитектурой, основанной на микросервисах, контейнеры могут обеспечить высокодинамичную и гибкую работу приложения, а также возможность быстрого и эффективного масштабирования. Существующие вычислительные ресурсы могут быть более эффективно оптимизированы по сравнению с традиционными вычислительными или более статическими контейнерными стратегиями развертывания.

Преимущества

- Невысокая цена, притом что можно просто и эффективно оптимизировать существующие серверные возможности.
- Несложное масштабирование, обеспечиваемое возможностью динамического развертывания контейнеров везде, где доступны вычислительные ресурсы.
- Ограничения для приложений незначительны, за исключением разве что долгого запуска приложения, что может снизить выгоду, которую дает контейнерное планирование.
- Контейнеры могут как работать непрерывно, так и вводиться в работу по необходимости, консервируя ресурсы.
- Контейнеры несложно разворачивать, для использования требуется минимальная конфигурация серверов.

Copyrighted material

в том, что вычислительные слайсы выделяются и планируются дискретно. Вы можете менять количество работающих единиц дискретно по мере обработки входящих запросов. Но в конечном итоге возможности масштабирования в соответствии с количеством входящих запросов ограничены масштабированием дискретных вычислительных слайсов. Микровычисления же масштабируются с полным динамическим соответствием тому количеству экземпляров, которое запускает конкретную функцию, необходимую для обработки текущего количества запросов.

Для обеспечения этого вычислительные функции, работающие внутри Lambda, должны быть небольшими, подвижными и практически не требовать инициализации или деструкции для эффективного выполнения.

Преимущества

- Автоматическое и практически безграничное масштабирование по мере надобности[1].
- Не требуется ни управление сервером, ни обновление системы.
- Оплата основана на фактическом количестве обработанных событий, накладные расходы ограниченны.
- Поддерживаются быстрота и легкость изменений необходимого масштабирования (изменения в сторону как увеличения, так и уменьшения прозрачны).

Недостатки

- Значительно ограничены функциональность и количество поддерживаемых языков.
- Архитектура приложения должна быть ориентирована на использование микровычислительных технологий.

[1] Разумеется, определенные пределы масштабирования существуют, но на практике масштабирование для нужд подавляющего большинства приложений можно считать безграничным.

- Данная среда является самой дорогой в расчете на один вычислительный цикл, но различные модели ценообразования и гибкие тарифы могут это компенсировать.
- Существующие реализации (AWS Lambda) не слишком хорошо обеспечивают возможности развертывания и отката или A/B-тестирования, затрудняя эти задачи в высокодоступных приложениях без необходимости изменения архитектуры или привлечения дополнительных внешних инструментов.

Примеры оптимального использования

Микровычисления — лучший вариант для массивной потоковой передачи данных и обработки событий. Полезны они также для валидации, трансформирования и фильтрации данных. А на границах сети — для валидации и регулирования входящих данных.

Наиболее эффективно их использование, когда необходимы небольшие операции, работающие на некрупных базах кода, для исполнения конкретных событийно-ориентированных программ.

Дальнейшие действия

При доступности широкого ряда различных возможностей для масштабирования почти для всех операционных нужд приложений и информационных технологий можно найти один или несколько подходящих вариантов. Но как понять, какой из них подойдет лучше всего? Большинство этих техник доступны только у определенных облачных провайдеров. А значит, выбирая облачного провайдера, придется принимать во внимание намерение или необходимость использования определенных вычислительных возможностей.

Тем не менее почти все облачные провайдеры поддерживают большинство этих опций, что в наши дни считается преимуществом. Не бойтесь использовать несколько опций внутри одного приложения, так как различные части вашего приложения, скорее всего, будут иметь различные нужды. Как всегда, учитывайте требования доступности и масштабируемости приложения и выбирайте то, что лучше всего соответствует потребностям приложения и команды, которая над ним работает.

Copyrighted material

25 AWS Lambda

AWS Lambda — новая среда выполнения программного обеспечения, созданная AWS и предназначенная для предоставления возможностей событийно-ориентированных вычислений без необходимости покупать, настраивать, конфигурировать и обслуживать серверы. Lambda обеспечивает практически безграничное масштабирование при тарификации в долях секунды.

Lambda — замечательная технология для фоновой обработки событийно-управляемых действий. Вот несколько типичных примеров использования.

- Преобразование новых загружаемых изображений.
- Обработка данных различных метрик в реальном времени.
- Валидация, фильтрация и преобразование потоковых данных.

Lambda наиболее хорошо подходит для следующих типов обработки:

- выполнения операций в результате наступления какого-либо события в приложении или его среде;
- фильтрации или преобразования потока данных;
- пограничной валидации или регулирования входящих данных.

Но основное преимущество AWS Lambda состоит в решении всех проблем с масштабированием. С Lambda вы можете масштабироваться практически до любого необходимого размера, не предпринимая никаких дополнительных действий.

Copyrighted material

Использование Lambda

В этот момент, возможно, вы думаете: «Звучит круто, но какие типы архитектуры допускают использование Lambda?» Некоторые из них рассматриваются в следующих разделах.

Обработка событий

Вообразите себе предложение для хранения и организации файлов изображений. Пользователи могут загружать файлы в облако, после чего они хранятся в S3. Приложение отображает уменьшенные копии этих изображений и позволяет пользователям изменять связанные с ними атрибуты, такие как имя, локация, имена людей на фотографии и т. п.

С помощью AWS Lambda это простое приложение может обрабатывать изображения перед загрузкой на S3. Как только новое изображение загружено, автоматически вызывается функция Lambda, которая создает уменьшенную копию изображения и отправляет ее в S3. Кроме того, другая функция Lambda считывает различные параметры изображения (размер, разрешение и т. п.), а затем сохраняет эти метаданные в базе данных. Затем приложение для хранения фотографий может предоставить инструменты для управления данными в базе. Эта архитектура показана на рис. 25.1.

Таким образом, само приложение для хранения изображений может не участвовать в процессе загрузки. Можно положиться на стандартные возможности загрузки S3 и две функции Lambda, которые сделают все необходимое для загрузки файлов. Поэтому приложение может сосредоточиться на собственных функциях — на управлении метаданными в базе данных с уже загруженными изображениями.

Внутренняя часть мобильного приложения

Рассмотрим мобильную игру, где пользовательский прогресс, трофеи и результаты хранятся в облаке, в результате чего эти данные доступны игровому сообществу, а также самому пользователю на любом устройстве. У этого приложения есть серия различных API,

Copyrighted material

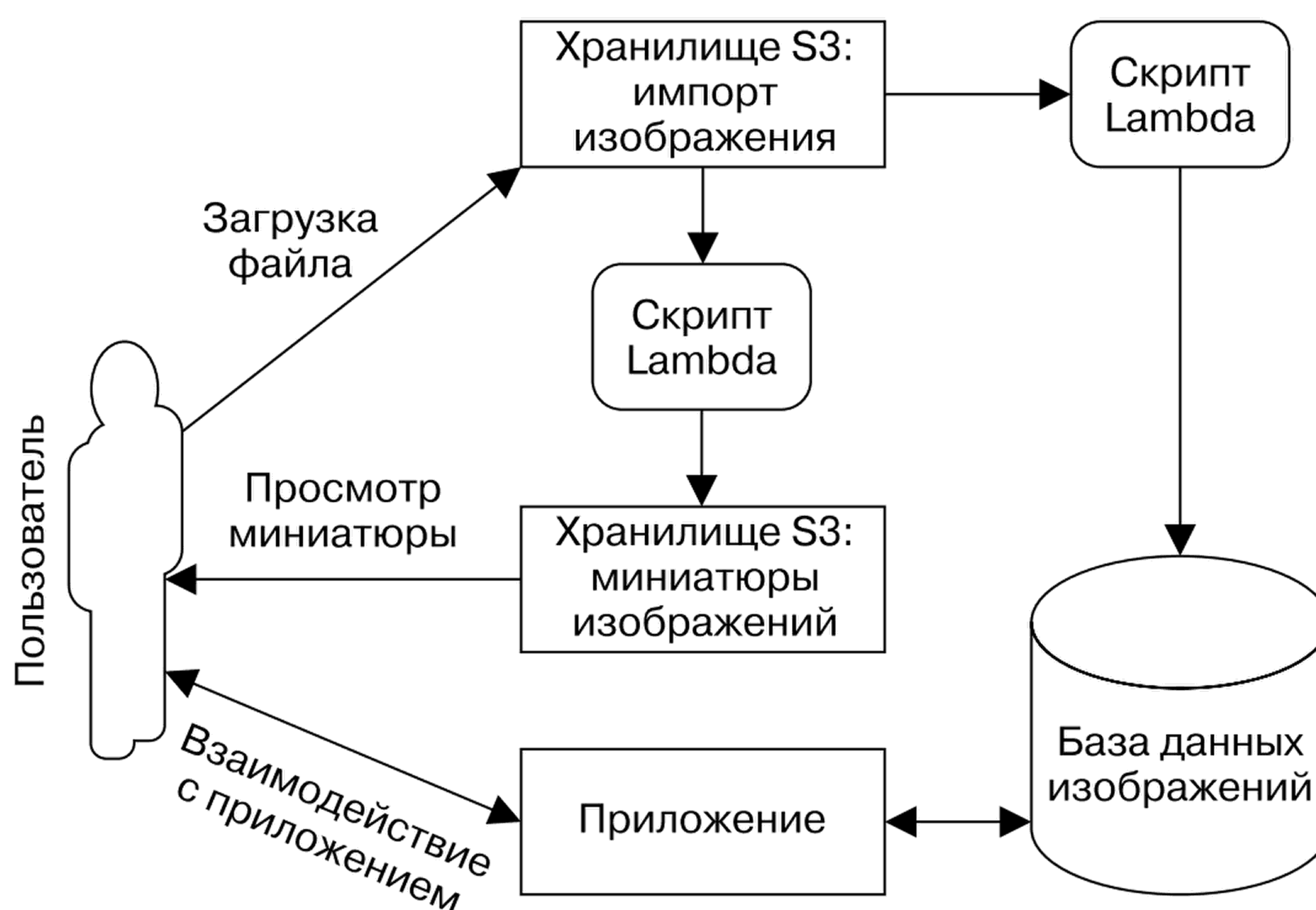


Рис. 25.1. Использование Lambda при загрузке файлов

благодаря которым приложение может сохранять данные в облако, получать оттуда пользовательскую информацию, а также обеспечивать взаимодействие в сообществе.

Необходимые API созданы с помощью API Gateway[1], которое связывается с рядом функций Lambda. Скрипты выполняют необходимые операции в связке с какой-либо базой данных, чтобы обработать облачную внутреннюю часть мобильной игры. Эта архитектура показана на рис. 25.2.

В этой модели на внутренней части не требуется никаких серверов, а любое масштабирование происходит автоматически.

Сбор данных в Интернете вещей

Представьте себе приложение, собирающее данные с огромного количества датчиков, установленных по всему миру. Данные с этих датчиков поступают регулярно. Таким образом, на серверной стороне

---

1 Amazon API Gateway — сервис для создания API, который спроектирован для работы с AWS Lambda.

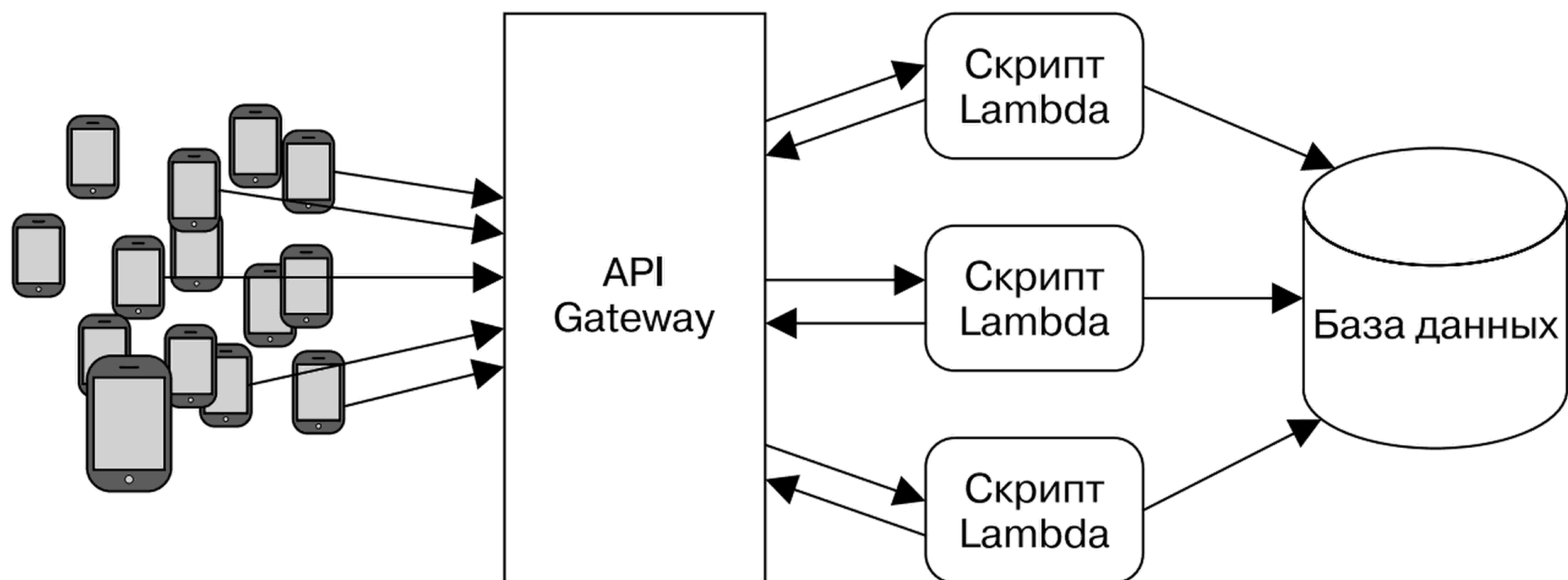


Рис. 25.2. Использование Lambda для внутренней части мобильного приложения

скапливается огромное количество данных, которые регулярно обрабатывает какое-то приложение для хранения в хранилище данных определенного типа. При сборе данные валидируются, возможно, проходят небольшую обработку, после чего полученные в результате данные отправляются в хранилище.

Это очень простое приложение, выполняющее только базовую валидацию и верификацию данных, а затем сохраняющее последние во внутреннем хранилище для последующей обработки. Однако, несмотря на его простоту, масштаб сбора данных довольно велик — может достигать миллионов или миллиардов событий в минуту. Точное количество зависит от числа датчиков.

Эта архитектура задействует канал[1] сбора данных, который отправляет их функции AWS Lambda, выполняющей все необходимые действия по фильтрации или обработке данных, прежде чем отправить их в хранилище. Архитектура изображена на рис. 25.3.

Lambda отлично подходит для обработки больших объемов данных, которые поступают непрерывно и с большой скоростью.

---

[1] Amazon Kinesis — канал сбора потоковых данных в реальном времени, специально разработанный для сбора большого количества потоковых данных.

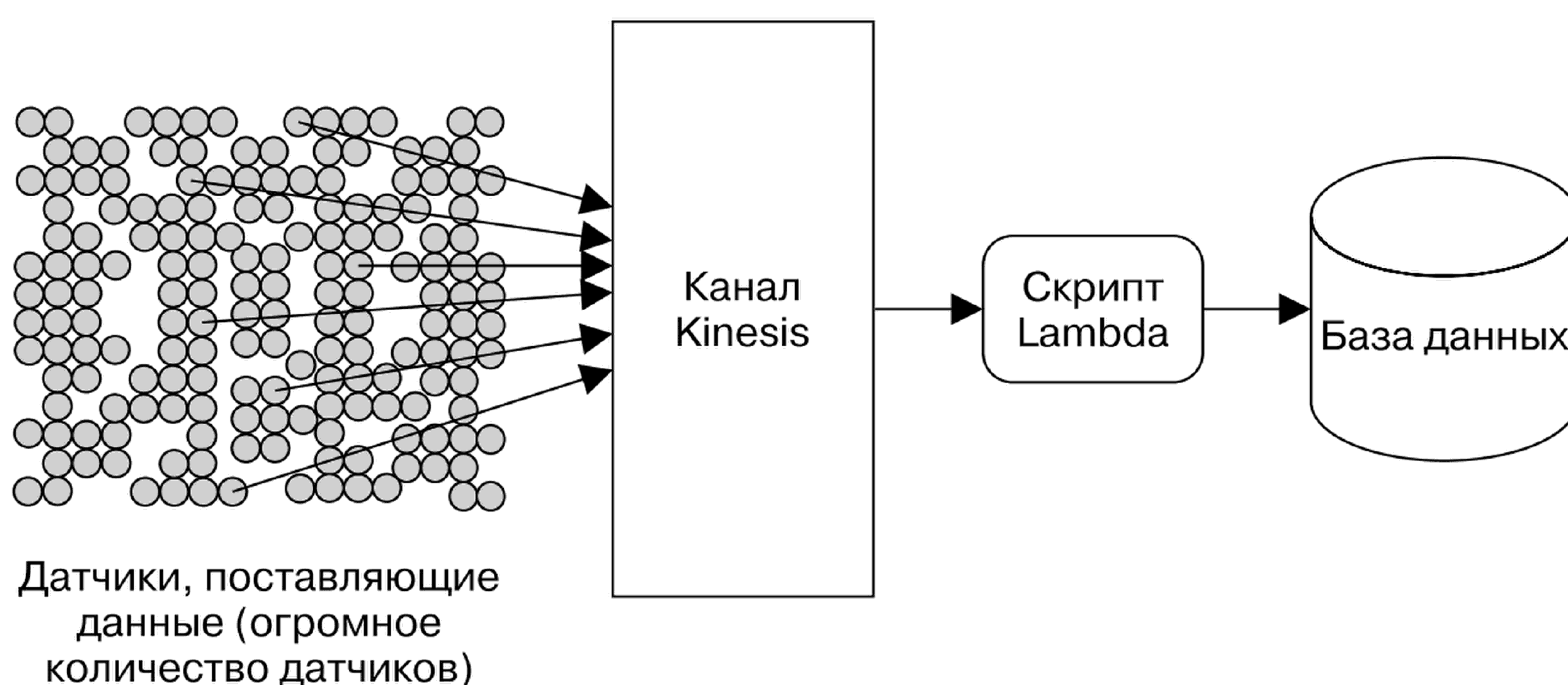


Рис. 25.3. Сбор данных с датчиков в Интернете вещей

Преимущества и недостатки Lambda

У Lambda есть одно неоспоримое преимущество — масштаб. Для обработки нагрузок наибольшего масштаба без необходимости расширять инфраструктуру, выделенную вашему приложению, ничего лучше не найти. Это обеспечивается за счет обязательных требований к коду: предельной простоты, возможности быстро и эффективно запустить его на нескольких серверах и в нескольких стеках. Это и есть ключ к успеху Lambda: небольшие фрагменты кода со скромными требованиями к ресурсам выполняются в многократно увеличенном масштабе.

В каких же случаях стоит использовать Lambda? Чтобы ответить на этот вопрос, давайте рассмотрим недостатки этого сервиса.

- Внутренние требования к коду — простота, управляемость событиями, быстрая обработка.
- Сложные установка и конфигурация.
- Отсутствуют собственные среды для стейджинга или тестирования.
- Отсутствуют собственные возможности для развертывания или отката.

- Отсутствуют собственные возможности для A/B-тестирования.
- Отсутствует среда разработки для создания и тестирования функций Lambda.

Короче говоря, Lambda — отличное решение для масштабирования небольших скриптов, но для всего прочего, необходимого при развертывании крупномасштабного приложения, она плохо подходит.

При эффективном использовании технология Lambda может быть очень полезна для обеспечения самых экстремальных уровней масштабирования. Однако старайтесь использовать ее только для тех задач, в которых она хороша. Для удовлетворения других нужд вашего приложения, которые не могут быть в полной мере обеспечены Lambda, лучше применять другие среды выполнения или развертывания.

Copyrighted material

Часть VI
Заключение

Масштабирование приложений не ограничивается обеспечением работы с большим количеством пользователей.

Copyrighted material

26 Общий обзор всех аспектов масштабирования

В этой книге мы рассмотрели очень много материала по значительному количеству тем. Помочь вам успешно масштабировать свои приложения могут все эти темы, взятые вместе:

- доступность и обеспечение доступности;
- риски и управление рисками;
- создание приложений с использованием сервисов и микросервисов;
- масштабирование приложения и команды разработки, ответственной за приложение;
- использование облачных технологий для облегчения масштабирования приложений.

Доступность

Доступность — способность приложения выполнять те задачи, для которых оно и предназначено. Доступность не является синонимом надежности, которая означает способность приложения

Copyrighted material

работать безошибочно. Система, которая, складывая 2 и 3, получает в результате 6, обладает плохой надежностью, а система, которая вообще не возвращает результат этого действия, — плохой доступностью.

Плохая доступность может объясняться несколькими факторами, в частности:

- недостатком ресурсов;
- незапланированным изменением нагрузки;
- возросшим количеством перемещаемых фрагментов кода;
- наличием внешних зависимостей;
- наличием технического долга.

Доступность приложения, как правило, становится первой проблемой, с которой сталкивается приложение, превысившее имеющиеся возможности для роста. Подробно тема доступности раскрывается в части I, где в числе прочего освещаются аспекты ее измерения, а также поддержания высокой доступности приложения. Даже при постоянно увеличивающихся потребностях в масштабировании доступность приложения можно поддерживать на необходимом уровне.

Управление рисками

Невозможно управлять рисками в системе, если эти риски неизвестны. Это самый важный урок из части II. Осознание ваших рисков — первый и самый важный шаг к созданию высокодоступного крупномасштабного приложения.

После того как вам стали известны все ваши риски, вы можете управлять ими. Конечно, лучше всего было бы устранить риск вовсе, однако зачастую это не окупается как с точки зрения реальной стоимости, так и с позиции оценки возможностей приложения. Скорее всего, найдутся более важные как для вас, так и для ваших клиентов задачи, работа над которыми будет выгоднее, чем устранение уже известных рисков.

Вместо того чтобы устранять риски, ими нужно управлять. Этот процесс включает в себя оценку двух параметров каждого риска — *вероятности* и *критичности*. Значение этих двух величин так высоко, что им посвящена целиком глава 6. Кратко вспомним: критичность риска определяется тяжестью его последствий для системы в случае реализации, а вероятность — степенью возможности того, что риск произойдет.

Риск, который может вызвать в приложении серьезные проблемы, но в реальности маловероятен, вряд ли представляет серьезную опасность. Аналогично риск, реализация которого весьма вероятна, но последствия этого незначительны, устранять также было бы нерационально. А вот риск, наступление которого весьма возможно и при этом последствия могут оказаться тяжелыми, достаточно важен для того, чтобы устранение его стало приоритетной задачей.

Мы рассмотрели инструмент, называемый *матрицей рисков*, который может быть очень эффективным для управления рисками в вашем приложении и в определении того, какие риски должны быть удалены или смягчены. Мы обсудили также техники смягчения рисков, валидации планов устранения последствий рисков, а также создания приложений с пониженными рисками.

Сервисы

Сервис — отдельная закрытая система, обеспечивающая бизнес-возможность, которая может быть использована при создании одного или нескольких более крупных продуктов. Сервисы предоставляют шаблон архитектуры приложения, использование которого позволяет создавать системы с обеспечением возможности масштабирования как самой системы, так и команды разработки.

При создании крупномасштабных приложений сервисы предоставляют возможность принимать прогрессивные решения о масштабировании, улучшают степень концентрации и контроля командой своей работы, снижают сложность на локальном уровне, повышают возможности для тестирования и развертывания.

Copyrighted material

Обсуждая сервисы, мы рассмотрели также инструменты и рекомендации, с помощью которых можно обеспечить высокую доступность приложения на сервисном уровне, а также снизить негативное влияние на пользователей в случае выхода сервисов из строя.

Масштабирование

Мы уже обсудили, как учесть управление системой и обеспечение доступности при планировании масштабирования системы. Поговорили о том, как закладывать запас на две ошибки, чтобы избежать аварийных циклов и каскадных зависимостей.

Мы рассмотрели также парадигму отдельной команды, владеющей сервисной архитектурой. Используя ее, можно обеспечить масштабирование вашей организации по мере роста приложения, а также эффективно организовать работу большого числа инженеров над одним приложением. Это предусматривает четкое определение обязанностей владельцев сервисов и организацию вашего приложения в соответствии с этими принципами.

Мы также обсудили использование инструментов для управления зависимостями сервисов для поддержания качества приложения даже в период быстрого роста, включая внутренние SLA и уровни сервисов.

Облачные технологии

Последней темой нашего обсуждения стали облачные технологии и возможности, которые они предоставляют для масштабирования приложений.

Мы рассмотрели, как облачные технологии изменили наши представления о программировании и способ мышления в области создании приложений. Мы обсудили организацию топографической распределенности вашего приложения в облаке с географической и сетевой точек зрения. Кроме того, рассмотрели, в каких случаях можно ошибочно считать, что приложение является распределенным и географически, и по сети, в то время как это не

так и возникают внутренние зависимости, увеличивающие риск возникновения проблем.

Мы обсудили управление инфраструктурой и конкретные особенности, связанные с крупным масштабом приложений. Затронули вопрос выделения облачных ресурсов и вашу роль в обеспечении приложения всеми необходимыми для функционирования ресурсами.

Мы рассмотрели также доступные вам при использовании облака вычислительные опции. Обсудили AWS Lambda и революционное будущее масштабируемой разработки, связанное с этим сервисом.

Архитектура под масштабирование

Архитектура под масштабирование не ограничивается обеспечением работы с большим количеством пользователей в каждый момент времени. В обеспечении масштабируемости существует множество аспектов:

- способность обеспечить одновременную работу большого и постоянно увеличивающегося количества пользователей;
- способность обеспечить обработку большого и постоянно увеличивающегося количества данных, необходимых вашим пользователям;
- способность справиться с ростом сложности запросов клиентов, которые они хотят удовлетворить с помощью вашего приложения;
- способность по мере надобности увеличить количество инженеров, работающих над приложением, без потери качества, скорости и эффективности разработки;
- способность поддерживать работоспособность и корректное функционирование вашего приложения даже во время осуществления всех упомянутых ранее изменений и улучшений.

Добиться всего этого — нелегкая задача, но техники, описанные в этой книге, могут помочь в решении этих и многих других проблем, связанных с масштабированием вашего приложения.

Copyrighted material

Ли Атчисон

Масштабирование приложений. Выращивание сложных систем

Перевели с английского *А. Ананич, О. Потапова, К. Русецкий*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художественный редактор	*С. Заматевская*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*А. Барцевич*

Изготовлено в России. Изготовитель: ООО «Питер Пресс».
Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург, улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 09.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Подписано в печать 24.08.17. Формат 60×90/16. Бумага офсетная. Усл. п. л. 16,000.
Тираж 1000. 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

Copyrighted material