

Майкл Нейгард

Release it!

Проектирование и дизайн ПО для тех, кому не всё равно

- ⌋ Стабильность
- ⌋ Мощность
- ⌋ Бесперебойность
- ⌋ Работоспособность



Michael T. Nygard

Release it!

Design and Deploy Production-Ready Software





БИБЛИОТЕКА ПРОГРАММИСТА

Майкл Нейгард

Release it!

Проектирование
и дизайн ПО для тех,
кому не всё равно



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

2016

ББК 32.973.2-018-02
УДК 004.45
Н46

М. Нейгард

- Н46 Release it! Проектирование и дизайн ПО для тех, кому не всё равно. — СПб.: Питер, 2016. — 320 с.: ил. — (Серия «Библиотека программиста»).
- ISBN 978-5-496-01611-7

Неважно, каким инструментом вы пользуетесь для программной разработки — Java, .NET, или Ruby on Rails. Написание кода — это еще только полдела. Готовы ли вы к внезапному наплыву ботов на ваш сайт? Предусмотрена ли в вашем ПО «защита от дурака»? Правильно ли вы понимаете юзабилити? Майкл Нейгард утверждает, что большинство проблем в программных продуктах были заложены в них еще на стадии дизайна и проектирования. Вы можете двигаться к идеалу сами — методом проб и ошибок, а можете использовать опыт автора. В этой книге вы найдете множество шаблонов проектирования, помогающих избежать критических ситуаций, и не меньшее количество антишаблонов, иллюстрирующих неправильные подходы с подробным анализом возможных последствий. Любой разработчик, имеющий опыт многопоточного программирования, легко разберется в приведенных примерах на Java, которые подробно поясняются и комментируются.

Стабильность, безопасность и дружелюбный интерфейс — вот три важнейших слагаемых успеха вашего программного продукта. Если в ваши планы не входит в течение последующих лет отвечать на недовольные письма пользователей, выслушивать критику заказчиков и постоянно латать дыры, устраняя возникающие баги, то прежде чем выпустить финальный релиз, прочтите эту книгу.

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

ББК 32.973.2-018-02
УДК 004.45

Права на издание получены по соглашению с Pragmatic Bookshelf. 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 978-0978739218 англ.

© Pragmatic Bookshelf

ISBN 978-5-496-01611-7

© Перевод на русский язык ООО Издательство «Питер», 2016

© Издание на русском языке, оформление ООО Издательство «Питер», 2016

© Серия «Библиотека программиста», 2016

Краткое содержание

Предисловие	12
-------------------	----

1. Введение	16
-------------------	----

Часть I. Стабильность

2. Исключение, помешавшее работе авиакомпании	24
---	----

3. Понятие стабильности.....	38
------------------------------	----

4. Антипаттерны стабильности	46
------------------------------------	----

5. Паттерны стабильности	106
--------------------------------	-----

6. Заключение по теме стабильности	135
--	-----

Часть II. Вычислительная мощность

7. Затоптаны клиентами	138
------------------------------	-----

8. Понятие вычислительной мощности.....	151
---	-----

9. Антипаттерны вычислительной мощности.....	165
--	-----

10. Паттерны вычислительной мощности	191
--	-----

Часть III. Общие вопросы проектирования

11. Организация сети	204
12. Безопасность	211
13. Доступность.....	214
14. Администрирование.....	225
15. Заключение по теме проектирования	233

Часть IV. Эксплуатация

16. Феноменальная мощь и маленькое жизненное пространство	236
17. Прозрачность	248
18. Адаптация.....	292
Список литературы.....	317

Содержание

Предисловие	12
Для кого предназначена эта книга?.....	13
Структура книги	14
Анализ примеров	15
Благодарности	15
 1. Введение	 16
1.1. Правильный выбор цели	17
1.2. Важность интуиции	18
1.3. Качество жизни	18
1.4. Охват проблемы	19
1.5. Миллионом больше, миллионом меньше.....	20
1.6. Прагматичная архитектура	21
 Часть I. Стабильность	
 2. Исключение, помешавшее работе авиакомпании	 24
2.1. Авария.....	25
2.2. Последствия.....	30

2.3. Анализ причин сбоя	31
2.4. Бесспорное доказательство	34
2.5. Легче предупредить, чем лечить?	36
3. Понятие стабильности	38
3.1. Определение стабильности	39
3.2. Режимы отказов	41
3.3. Распространение трещин	42
3.4. Цепочка отказов	43
3.5. Паттерны и антипаттерны	44
4. Антипаттерны стабильности	46
4.1. Точки интеграции	48
4.2. Цепные реакции	61
4.3. Каскадные отказы	64
4.4. Пользователи.....	67
4.5. Блокированные программные потоки.....	79
4.6. Самостоятельно спровоцированный отказ в обслуживании	85
4.7. Эффекты масштабирования	88
4.8. Несбалансированные мощности.....	93
4.9. Медленная реакция.....	97
4.10. SLA-инверсия	98
4.11. Огромные наборы результатов	102
5. Паттерны стабильности	106
5.1. Задавайте таймауты	106
5.2. Предохранители	110
5.3. Переборки.....	113
5.4. Стабильное состояние	117
5.5. Быстрый отказ	123
5.6. Квитирование.....	126
5.7. Тестовая программа	128
5.8. Разделение связующего ПО	132
6. Заключение по теме стабильности	135

Часть II. Вычислительная мощность

7. Затоптаны клиентами	138
7.1. Обратный отсчет и запуск	138
7.2. Цель — пройти тест качества	139
7.3. Нагрузочное тестирование	142
7.4. Растоптаны толпой.....	145
7.5. Недочеты тестирования.....	147
7.6. Последствия.....	148
8. Понятие вычислительной мощности.....	151
8.1. Определение вычислительной мощности.....	151
8.2. Ограничители.....	153
8.3. Взаимное влияние	155
8.4. Масштабируемость.....	155
8.5. Мифы о вычислительной мощности.....	157
8.6. Заключение	163
9. Антипаттерны вычислительной мощности.....	165
9.1. Конкуренция в пуле ресурсов	165
9.2. Избыточные JSP-фрагменты.....	169
9.3. Перебор с AJAX	170
9.4. Затянувшиеся сеансы	173
9.5. Напрасный расход пространства HTML-страницами.....	175
9.6. Кнопка обновления страницы	179
9.7. Кустарный SQL-код.....	180
9.8. Эвтрофикация базы данных.....	183
9.9. Задержка в точках интеграции.....	186
9.10. Cookie-монстры.....	187
9.11. Заключение	189
10. Паттерны вычислительной мощности	191
10.1. Организация пула соединений	192
10.2. Будьте осторожны с кэшированием.....	194

10.3. Предварительное вычисление контента.....	196
10.4. Настройка сборщика мусора	199
10.5. Заключение.....	202

Часть III. Общие вопросы проектирования

11. Организация сети	204
11.1. Многоинтерфейсные серверы.....	204
11.2. Маршрутизация	207
11.3. Виртуальные IP-адреса	208
12. Безопасность	211
12.1. Принцип минимальных привилегий	211
12.2. Настроенные пароли	212
13. Доступность.....	214
13.1. Сбор требований доступности	214
13.2. Документирование требований доступности.....	215
13.3. Балансировка нагрузки	217
13.4. Кластеризация	223
14. Администрирование.....	225
14.1. «Совпадают ли условия тестирования с условиями эксплуатации?»	226
14.2. Конфигурационные файлы.....	228
14.3. Начало и завершение работы	231
14.4. Административные интерфейсы.....	232
15. Заключение по теме проектирования	233

Часть IV. Эксплуатация

16. Феноменальная мощь и маленькое жизненное пространство	236
16.1. Время максимальной нагрузки.....	236
16.2. Первое в жизни Рождество	237
16.3. Рука на пульсе.....	238

16.4. День благодарения	240
16.5. Черная пятница	240
16.6. Жизненно важные функции	242
16.7. Диагностические тесты.....	243
16.8. Обращение к специалисту.....	244
16.9. Сравнение вариантов лечения	245
16.10. Есть ли реакция на лекарство?.....	246
16.11. Выводы.....	247
17. Прозрачность	248
17.1. Точки зрения	250
17.2. Проектирование с учетом прозрачности.....	257
17.3. Применение технологий	258
17.4. Протоколирование	259
17.5. Системы мониторинга	266
17.6. Стандарты де юре и де факто.....	271
17.7. База данных функционирования системы	280
17.8. Вспомогательные процессы.....	286
17.9. Заключение	290
18. Адаптация.....	292
18.1. Адаптация со временем	293
18.2. Проектирование адаптируемого ПО	294
18.3. Адаптируемая архитектура предприятия	301
18.4. Безболезненный переход к новой версии	308
18.5. Заключение	316
Список литературы.....	317

Предисловие

Вы больше года работали над сложным проектом. Наконец, кажется, все программные компоненты готовы, и даже модульное тестирование выполнено. Можно вздохнуть с облегчением. Вы закончили.

Или?

Означает ли «готовность программного компонента», что он готов к работе? Подготовлена ли ваша система к развертыванию? Смогут ли с ней справиться без вашей помощи обслуживающий персонал и толпы реальных пользователей? Нет ли у вас нехорошего предчувствия грядущих ночных звонков с паническими просьбами о помощи? Ведь разработка — это не только ввод всех необходимых программных компонентов.

Слишком часто работающая над проектом группа ставит своей целью не долгую бесппроблемную эксплуатацию, а прохождение тестов контроля качества. То есть изрядная часть работы концентрируется вокруг грядущего тестирования. Однако тестирование, даже гибкое, практичное и автоматизированное, не является гарантией готовности программы к функционированию в реальных условиях. Возникающие в реальности нагрузки, обусловленные сумасшедшими пользователями, трафиком и хакерами из стран, о которых вы даже никогда не слышали, выходят за рамки условий, которые можно смоделировать в пределах теста.

Чтобы проследить за готовностью программы к реальным условиям, требуется соответствующая подготовка. Я хочу помочь вам выявить источник проблем и подсказать, как с ними бороться. Но прежде чем мы приступим к делу, хотелось бы осветить некоторые популярные заблуждения.

Во-первых, следует примириться с тем, что даже самое лучшее планирование не позволяет избежать некоторых нехороших вещей. Разумеется, их всегда по возможности следует предотвращать. Но мысль о том, что вы в состоянии предсказать

и устранить все возможные проблемы, является роковой ошибкой. Вместо этого следует предпринять некие действия, сводящие на нет возможные риски, постаравшись при этом гарантировать работоспособность системы даже в случае непредвиденных тяжелых повреждений.

Во-вторых, следует понять, что «версия 1.0» — это не конец процесса разработки, а начало самостоятельного существования системы. Ситуация напоминает выросшего ребенка, который в первое время вынужден жить с родителями. Вряд ли вам захочется, чтобы ваш сын навсегда поселился в вашей квартире, особенно вместе со своей супругой, четырьмя детьми, двумя собаками и какаду.

Аналогичным образом все принятые на этапе разработки конструкторские решения окажут огромное влияние на вашу жизнь после выхода версии 1.0. Если вы не сумеете сконструировать систему, ориентированную на реальную производственную среду, ваша жизнь после ее выпуска будет полна волнений. И зачастую неприятных. В этой книге вы познакомитесь с важными компромиссами и узнаете, как их добиваться.

И наконец, несмотря на всеобщую любовь к технологиям, привлекательным новым методам и крутым системам, вам придется признать, что ничего из этого по большому счету не имеет значения. В мире бизнеса — а именно он заказывает музыку — все сводится к деньгам. Системы стоят денег. Чтобы компенсировать затраты на свое создание, они должны приносить деньги в виде прямого дохода или экономии средств. Дополнительная работа, как и простои, оборачивается дополнительными расходами. Неэффективный код *дорого* обходится, так как требует капитальных вложений и эксплуатационных затрат. Чтобы понять действующую систему, нужно думать в терминах выгоды. Чтобы остаться в деле, вы должны приносить деньги или хотя бы не терять их.

Я надеюсь, эта книга позволит изменить ситуацию и поможет вам и организации, в которой вы работаете, избежать огромных потерь и перерасхода средств — частой сопутствующей проблемы корпоративного программного обеспечения.

Для кого предназначена эта книга?

Я писал эту книгу для архитекторов, проектировщиков и разработчиков программного обеспечения корпоративного класса, в том числе сайтов, веб-служб и EAI-проектов. Для меня программное обеспечение *корпоративного класса* означает, что приложение должно работать, в противном случае компания потеряет деньги. В эту категорию попадают как коммерческие системы, напрямую связанные с получением дохода, например путем продаж, так и важные внутрикорпоративные системы, необходимые для выполнения рабочих обязанностей сотрудников. Если выход вашей программы из строя может на целый день оставить человека без работы, значит, эта книга предназначена для вас.

Структура книги

Книга разделена на четыре части, каждая из которых начинается с практического примера. В части I показано, как сохранить активность системы, обеспечив ее безотказную работу. Несмотря на обещанную надежность за счет избыточности, распределенные системы демонстрируют доступность, выражаемую скорее «двумя восьмерками», чем желанными «пятью девятками»¹.

Необходимой предпосылкой для рассмотрения любых других вопросов является *стабильность*. Если ваша система рухнет несколько раз в день, никто не будет рассматривать аспекты, связанные с отдаленным будущим. В такой среде доминируют краткосрочные исправления и, как следствие, краткосрочное мышление. Без стабильности не будет конкурентоспособного будущего, поэтому первым делом следует понять, каким образом вы можете гарантировать стабильность базовой системы, которая послужит основой для дальнейших работ.

Как только вы добьетесь стабильности, приходит время позаботиться о мощности системы. Этой теме посвящена часть II, в которой вы познакомитесь со способами измерения этого параметра, узнаете, что он на самом деле означает, и научитесь оптимизировать его в долгосрочной перспективе. Я покажу вам примеры паттернов и антипаттернов, иллюстрирующие хорошие и плохие проектные решения, и продемонстрирую потрясающее влияние, которое эти решения могут оказывать на вычислительную мощность (и, как следствие, на количество ночных телефонных звонков и сообщений).

В части III мы рассмотрим общие вопросы проектирования, которые архитектор должен учитывать при написании программ для центров обработки данных. За последнее десятилетие аппаратное обеспечение и проектирование инфраструктуры претерпели значительные изменения; к примеру, такая относительно редкая раньше практика, как виртуализация, сейчас распространена практически повсеместно. Сети стали куда сложнее — теперь они являются многоуровневыми и программируемыми. Обычным делом стали сети хранения данных. Разработка программного обеспечения должна учитывать и использовать эти новшества для обеспечения бесперебойной работы центров обработки данных.

В части IV существование системы рассматривается в рамках общей информационной экосистемы. Зачастую производственные системы напоминают кота Шрёдингера — они заперты в коробке без возможности наблюдать за их состоянием. Здоровью экосистемы это не способствует. Отсутствие информации делает невозможными целенаправленные улучшения². В главе 17 обсуждаются факторы, технологии и процессы, которые следует изучать у работающих систем (это единственная ситуация, когда *можно* изучать определенные вещи).

¹ То есть 88, а не 99,999 % времени безотказной работы.

² Случайные догадки порой могут приводить к улучшениям, но в подавляющем большинстве случаев хаос от них только возрастает.

Выяснив показатели работоспособности и производительности конкретной системы, вы сможете действовать на основе этих данных. Более того, подобный подход является обязательным — действия *нужно* предпринимать только в свете полученной информации. Иногда это проще сказать, чем сделать, и в главе 18 мы рассмотрим препятствия на пути к изменениям и способы их уменьшения и преодоления.

Анализ примеров

Для иллюстрации основных концепций книги я привел несколько развернутых примеров. Они взяты из реальной жизни и описывают системные отказы, которым я был свидетелем. Эти отказы оказались крайне дорогостоящими — и компрометирующими — для тех, кто имел к ним отношение. Поэтому я опустил информацию, которая позволила бы идентифицировать конкретные компании и людей. Также я поменял названия систем, классов и методов. Но изменениям подверглись только «несущественные» детали. Во всех случаях я указал отрасль, последовательность событий, режим отказа, путь распространения ошибки и результат. Цена всех этих отказов не преувеличена. Это реальные потери реальных фирм. Я упомянул эти цифры в тексте, чтобы подчеркнуть серьезность материала. Отказ системы ставит под удар реальные деньги.

Благодарности

Идея книги появилась после моего выступления перед группой Object Technology. Поэтому я должен поблагодарить Кайла Ларсона (Kyle Larson) и Клайда Каттинга (Clyde Cutting) — благодаря им я вызвался сделать доклад, который они благосклонно приняли. Неоценимую поддержку мне оказали Том (Tom Porpendieck) и Мэри Поппендик (Mary Porpendieck), авторы двух фантастических книг о «бережливой разработке»¹. Это они убедили меня, что я полностью готов к написанию собственной книги. Отдельное спасибо моему другу и коллеге Диону Стюарту (Dion Stewart), который постоянно оценивал написанные фрагменты.

Разумеется, было бы упущением не поблагодарить мою жену и дочерей. Первую половину своей жизни моя младшая дочь смотрела, как я работаю над книгой. Вы все так терпеливо ждали, когда я работал даже в выходные. Мари, Анна, Элизабет, Лаура и Сара, огромное вам спасибо.

¹ Книги *Lean Software Development* и *Implementing Lean Software Development*. Последняя была переведена на русский язык издательством «Вильямс» и вышла под названием «Бережливое производство программного обеспечения. От идеи до прибыли».

1 Введение

Современные учебные программы по разработке программного обеспечения являются до ужаса неполными. В их рамках рассматривается только то, как *должна* вести себя система. При этом не раскрывается обратный аспект — как системы *не должны* себя вести. А они не должны рушиться, зависать, терять данные, предоставлять несанкционированный доступ к информации, приводить к потере денег, разрушать вашу компанию или лишать вас заказчиков.

В этой книге мы рассмотрим способы планирования, проектирования и создания программного обеспечения — особенно распределенных систем — с учетом всех неприятных аспектов реального мира. Мы подготовимся к армии нелогичных пользователей, делающих сумасшедшие, непредсказуемые вещи. Атака на вашу программу начинается с момента ее выпуска. Программа должна быть устойчива к любому флэшмобу, слэшдот-эффекту или ссылкам на сайтах Fark и Digg. Мы непредвзято рассмотрим программы, не прошедшие тестирование, и найдем способы гарантировать их выживание при контакте с реальным миром.

Современная разработка программного обеспечения своей оторванностью от реальности напоминает ситуацию с разработкой автомобилей в начале 90-х годов. Модели автомобилей, придуманные исключительно в прохладном комфорте лабораторий, великолепно выглядели на чертежах и в системах автоматизированного проектирования. Идеальные изгибы машин блесли перед гигантскими вентиляторами, создающими ламинарный поток. Проектировщики, населявшие эти спокойные

производственные помещения, выдавали конструкции, которые были элегантными, изящными, хорошо продуманными, но хрупкими, не соответствующими критериям прочности и в итоге имели весьма скромный ресурс. Построение и проектирование программного обеспечения по большей части происходит в таких же отдаленных от жизненных реалий стерильных условиях.

Нам же нужны автомобили, спроектированные для реального мира. Мы хотим, чтобы этим занимались люди, понимающие, что смена масла *всегда* происходит на 3000 миль позже, чем нужно; что шины на последней одной шестнадцатой дюйма протектора должны функционировать так же хорошо, как и на первой; что водитель рано или поздно может ударить по тормозам, держа в одной руке сэндвич с яйцом, а в другой сотовый телефон.

1.1. Правильный выбор цели

Большинство программ разрабатывается для исследовательской лаборатории или тестеров из отдела контроля качества. Они проектируются и строятся, чтобы проходить такие тесты, как, к примеру, «клиенту требуется ввести имя и фамилию и по желанию — отчество». То есть усилия направляются на выживание в вымышленном царстве отдела контроля качества, а не в реальном мире.

Можно ли быть уверенным, что прошедшая контроль качества система готова к использованию? Простое прохождение тестов ничего не говорит о стабильности системы в следующие три, а то и десять лет ее жизни. В результате может родиться Toyota Camry от программирования, способная на тысячи часов безотказной работы. А может получится Chevy Vega, за несколько лет прогнивающий до дыр, или Ford Pinto, норовящий взорваться при каждом ударе в задний бампер. За несколько дней или недель тестирования невозможно понять, что принесут следующие несколько лет эксплуатации.

Проектировщики промышленных изделий долгое время придерживались политики подгонки «конструкции под технологичность изготовления». При таком подходе к разработке продуктов высокое качество должна сопровождать минимальная стоимость производства.

А еще раньше проектировщики и производители товаров вообще обитали в разных мирах. Нежелание проектировщиков участвовать в дальнейшем процессе приводило к появлению винтов в труднодоступных местах, деталей, которые легко было перепутать, и самопальных компонентов там, где могли бы применяться готовые комплектующие. Из всего этого неизбежно вытекало низкое качество и высокая стоимость производства.

Звучит знакомо, не так ли? Мы сейчас находимся в аналогичном положении. Мы постоянно опаздываем с выпуском новых систем, потому что время отнимают

постоянные звонки с требованиями сопровождения недоделанных проектов, уже выпущенных нами в свет. Нашим аналогом подгонки «конструкции под технологичность изготовления» является подгонка «конструкции под эксплуатацию». Результаты своих трудов мы передаем не изготовителю, а тем, кто будет с ними работать. Мы должны проектировать индивидуальные программные системы и целый комплекс независимых систем, обеспечивая низкую стоимость и высокое качество своей продукции.

1.2. Важность интуиции

Решения, принятые на ранней стадии, оказывают сильное влияние на конечный вид системы. Чем раньше принимается решение, тем сложнее потом от него отказаться. От того, как вы определите границы системы и как разобьете ее на подсистемы, зависит структура рабочей группы, объем финансирования, структура сопровождения программного продукта и даже хронометраж работ. Распределение обязанностей внутри группы является первым наброском архитектуры (см. закон Конвея в разделе 7.2). Ирония состоит в том, что на ранней стадии решения принимаются в условиях минимальной информированности. Группа еще понятия не имеет о конечной структуре будущего программного продукта, но уже должна принимать необратимые решения.

Даже в случае «гибких» проектов¹ для принятия решений лучше обладать даром предвидения. То есть проектировщик должен «пользоваться интуицией», чтобы заглянуть в будущее и выбрать наиболее надежный вариант. При одинаковых затратах на реализацию разные варианты могут иметь совершенно разную стоимость жизненного цикла, поэтому важно учесть, какой эффект каждое из решений окажет на доступность, вычислительную мощность и гибкость конечного продукта. Я продемонстрирую вам результаты десятков вариантов конструкции с примерами выгодных и вредных подходов. Все эти примеры я наблюдал в системах, с которыми мне довелось работать. И многое из этого стоило мне бессонных ночей.

1.3. Качество жизни

Версия 1.0 — это начало жизни вашей программы, но не конец проекта. И качество вашей жизни после предоставления версии 1.0 заказчикам зависит от решений, которые вы приняли в далеком прошлом.

¹ Честно признаюсь, что являюсь убежденным сторонником гибких методов. Акцент на раннем предоставлении и последовательных усовершенствованиях продукта способствует быстрому вводу программы в эксплуатацию. А так как только в процессе эксплуатации можно видеть реакцию программы на реальные воздействия, я выступаю за любые методы, позволяющие максимально быстро приступить к изучению ее поведения в реальных условиях.

И чем бы ни занимался ваш заказчик, он должен знать, что получает надежный, протестированный в условиях бездорожья и несокрушимый автомобиль, который повезет его бизнес вперед, а не хрупкую игрушку из стекла, которой суждено проехать в магазине гораздо больше времени, чем на реальной дороге.

1.4. Охват проблемы

«Кризис программного обеспечения» начался более тридцати лет назад. *Золотовладельцы* считают, что программное обеспечение по-прежнему стоит слишком дорого. (Этой теме посвящена книга Тома де Марко *Why Does Software Cost So Much?*) А по мнению *целевых доноров*, его разработка занимает слишком много времени, несмотря на то что теперь она измеряется месяцами, а не годами. Судя по всему, достигнутая за последние тридцать лет продуктивность иллюзорна.

Эти термины появились в сообществе гибкого проектирования. Золотовладельцем (gold owner) называют того, кто платит за программное обеспечение, а целевым донором (goal donor) — того, чьи запросы это программное обеспечение призвано удовлетворять. Как правило, это два разных человека.

В то же время, возможно, реальный рост продуктивности выражается в том, что теперь мы можем решать более масштабные проблемы, а не в удешевлении и ускорении производства уже существующих программ. За последние десять лет сфера охвата наших программ выросла на несколько порядков.

В спокойные времена клиент-серверных систем информационная база могла охватывать сотни и тысячи пользователей, при этом одновременно с системой работали в лучшем случае несколько дюжин из них. Современные заказчики без смущения называют такие цифры, как «25 тысяч параллельно работающих пользователей» и «4 миллиона уникальных посетителей в день».

Возросли и требования к сроку эксплуатации. Если раньше «пять девяток» (99,999 %) требовались только центральным серверам и обслуживающему их персоналу, то сейчас даже заурядным коммерческим сайтам нужна доступность 24 часа в сутки и 365 дней в году¹. Очевидно, мы резко шагнули вперед в масштабах создаваемого программного обеспечения, но одновременно с этим появились новые варианты отказов, обстановка стала более неблагоприятной, а терпимость к дефектам снизилась.

¹ Такая формулировка мне никогда не нравилась. Как инженер, я бы больше оценил выражение «24 на 365» или «24 на 7 и на 52».

Расширение границ задачи — быстро создавать программы, которые нравились бы пользователям и были простыми в обслуживании, — требует постоянного улучшения методик построения архитектуры и проектирования. При попытке применить вещи, подходящие для небольших сайтов, к системам с тысячами пользователей, к системам обработки транзакций, к распределенным системам, происходят самые разные отказы, часть из которых мы будем рассматривать подробно.

1.5. Миллионом больше, миллионом меньше

На карту ставится многое: успех вашего проекта, ваши права на акции или участие в прибылях, выживание вашей компании и даже ваше трудоустройство. Системы, ориентированные на прохождение тестов качества, часто так дороги в условиях эксплуатации, простоя и сопровождения, что никогда не становятся рентабельными, не говоря уже о выгоде, которая начинается после того, как прибыль от эксплуатации системы покрывает расходы на ее создание. Подобные системы демонстрируют низкий уровень эксплуатационной готовности, что приводит к прямым потерям в виде упущенной выгоды, а порой и косвенным потерям из-за ущерба, нанесенного торговой марке. Для многих моих клиентов цена простоя программного обеспечения превышает 100 000 долларов в час.

За год разница между 98 % и 99,99 % времени безотказной работы может дойти до 17 миллионов долларов¹. Представьте себе эти 17 миллионов, которые к итоговой сумме можно было бы добавить, всего лишь улучшив проект!

В лихорадке разработки можно легко принять решение, которое оптимизирует затраты на проектирование за счет эксплуатационных расходов. С точки зрения команды разработчиков, которой выделен фиксированный бюджет и указана фиксированная дата сдачи проекта, это имеет смысл. А вот для заказавшей программное обеспечение организации это плохой выбор. Эксплуатируются системы куда дольше, чем разрабатываются, — по крайней мере, те, которыми не прекратили пользоваться. Значит, бессмысленно избегать разовых затрат, вводя в уравнение постоянные расходы на эксплуатацию. Более того, с финансовой точки зрения куда лучше противоположный подход. Потратив 5000 долларов на автоматизированную систему сборки и выпуска, позволяющую избежать простоя при переходе к новой версии, можно сэкономить 200 000 долларов². Я думаю,

¹ Средняя стоимость 100 000 долларов за час простоя имеет место у поставщиков первого порядка.

² Предполагается 10 000 долларов на версию (оплата работы плюс стоимость запланированных простоев), четыре версии в год в течение пяти лет. Большинство компаний предпочитают более пяти версий в год, но я консерватор.

что большинство финансовых директоров без колебаний санкционируют расходы, способные принести 4000 % прибыли на инвестированный капитал.

Не избегайте разовых затрат на разработку, стараясь добиться снижения расходов на эксплуатацию.

Выбор структуры и архитектуры программы относится к финансовому решению. И его следует принимать с прицелом на стоимость внедрения и всех последующих затрат. Объединение технической и финансовой точек зрения относится к одной из наиболее важных тем, которые то и дело будут повторяться в этой книге.

1.6. Прагматичная архитектура

Термин *архитектура* применяется к двум разным видам деятельности. Один из них касается более высоких уровней абстракции, способствующих упрощению перехода с одной платформы на другую и большей независимости от аппаратного обеспечения и сетевого окружения. В крайних формах это превращается в «башню из слоновой кости» — населенную надменными гуру чистую комнату в стиле Кубрика, все стены которой украшены ящиками и стрелками. Оттуда исходят указания непрерывно работающим кодерам: «Используйте контейнерно-управляемое сохранение EJB-состояния!», «Для конструирования всех пользовательских интерфейсов используйте JSF!», «Все, что вам нужно, было нужно и когда-либо будет нужно, есть в Oracle!». Если в процессе написания кода по «корпоративным стандартам» вы когда-либо заикнетесь, что другая технология позволяет достичь нужного результата более простым путем, то станете жертвой архитектора, оторванного от реальности. И могу руку дать на отсечение, что архитектор, не слушающий кодеров из собственной рабочей группы, не станет прислушиваться и к пользователям. Думаю, результаты такого подхода вам доводилось видеть — это пользователи, которые бурно радуются падению системы, так как это позволяет им некоторое время от нее отдохнуть.

Другая порода архитекторов не только водит дружбу с кодерами, но может даже принадлежать к их числу. Такие архитекторы без колебаний понижают уровень абстракции и даже отказываются от него, если он не вписывается в общую картину. Эти прагматичные ребята, скорее всего, обсудят с вами такие вопросы, как использование памяти, требования процессора и пропускной способности, а также выгоды и недостатки гиперпоточности и объединения процессоров.

Архитектору из башни больше всего нравится представлять конечный результат в виде идеальных позванивающих кристаллов, в то время как прагматичный

архитектор непрерывно учитывает динамику изменений. «Как выполнить развертывание, не заставляя всех перезагружаться?», «Какие параметры нам следует собрать, как они будут анализироваться?», «Какая часть системы больше всего нуждается в усовершенствованиях?». Выстроенная архитектором из башни система не допускает доработки; все ее части строго пригнаны друг к другу и адаптированы к своим задачам. У прагматичного архитектора все компоненты системы хорошо справляются с текущими нагрузками, и при этом он знает, что нужно поменять, если со временем нагрузка распределится по-иному.

Если вы относитесь к прагматичным архитекторам, у меня есть для вас целая книга полезной информации. Если же вы пока сидите в башне из слоновой кости, но не прекратили чтения, возможно, я смогу уговорить вас спуститься на несколько уровней абстракции — туда, где пересекаются аппаратная часть, программное обеспечение и пользователи: в мир эксплуатации. В результате ваши пользователи и фирма, на которую вы работаете (да и вы сами), станут намного счастливее, когда подойдет время выхода новой версии!

Часть I

Стабильность

2 Исключение, помешавшее работе авиакомпанияи

Вы когда-нибудь замечали, что большие проблемы часто начинаются с мелочей? Крошечная ошибка программиста превращается в катящийся с горы по склону снежный ком. И по мере ускорения его движения растет масштаб проблемы. Именно такой случай произошел в одной крупной авиакомпании. Тысячам пассажиров пришлось поменять свои планы, а компании это обошлось в сотни тысяч долларов. Вот как это случилось.

Все началось с запланированного перевода кластеров баз данных программы Core Facilities (CF) на резервную систему¹. Авиакомпания переходила на сервис-ориентированную архитектуру, чтобы, как обычно бывает в таких случаях, повысить степень многократного использования, уменьшить время разработки и снизить производственные расходы. На тот момент применялась первая версия программы. Рабочая группа планировала поэтапное развертывание, обусловленное добавлением новых программных компонентов. План был вполне разумным, и, возможно, вы уже сталкивались с подобными вещами — в настоящий момент такое практикуется большинством крупных компаний.

Программа CF выполняла поиск рейсов — этот сервис часто встречается в приложениях для авиакомпаний. По дате, времени, коду аэропорта, номеру рейса или любой комбинации этих данных приложение ищет и возвращает данные с деталями рейса. Инцидент произошел, когда киоски самостоятельной регистрации, IVR-система и приложения «партнера по продажам» обновлялись для получения

¹ Все названия, места и даты изменены.

доступа к программе CF. Приложения партнера по продажам генерировали веб-каналы передачи данных для крупных сайтов бронирования. Голосовые сообщения и киоски регистрации отвечали за выделение пассажирам мест на борту. Были запланированы новые версии приложений для сопровождающих пассажиров и информационно-справочной службы, в результате которых поиск рейсов перекладывался на программу CF, но развертывание выполнить не успели, что, как вскоре выяснилось, было к лучшему.

Система предварительно записанных голосовых сообщений обозначается аббревиатурой IVR (Interactive Voice Response — интерактивный автоответчик).

Архитекторы программы CF прекрасно понимали ее важность. Они обеспечили высокую надежность. Программа работала на кластере серверов приложений J2EE с резервированием в базе данных Oracle 9i. Все данные хранились на большом внешнем RAID-массиве, дважды в день дублировались на внешний ленточный накопитель с копированием данных во второй массив хранения как минимум раз в пять минут.

В каждый момент времени сервер базы данных Oracle должен был работать на одном узле кластера, контролируемый продуктом Cluster Server на основе технологии Veritas, присваивающим виртуальные IP-адреса, и осуществляющий монтирование и размонтирование файловых систем в RAID-массиве. Пара резервных устройств распределителей нагрузки заблаговременно направляла входящий трафик на один из серверов приложений. Вызывающие приложения, например для киосков самостоятельной регистрации и IVR-системы, могли получать виртуальные IP-адреса через входной интерфейс. Ничто не предвещало проблем.

Если вы когда-либо имели дело с веб-сайтами или веб-службами, вам, скорее всего, будет знакомо то, что изображено на рис. 1. Это крайне распространенная архитектура с высоким уровнем надежности. У программы CF не было проблем, связанных с конкретным уязвимым звеном. Все аппаратное обеспечение дублировалось: процессоры, вентиляторы, накопители, сетевые карты, источники питания и сетевые коммутаторы. Каждый сервер стоял в собственной стойке на случай, если какая-нибудь из стоек получит повреждение. По сути, в случае пожара, наводнения, бомбардировки или падения метеорита за работу принимался второй комплект оборудования, расположенный в пятидесяти километрах от первого.

2.1. Авария

Как и в случае с большинством моих крупных клиентов, местная группа инженеров имела навыки работы с инфраструктурой аэропорта. Более того, большая часть работ выполнялась этой группой более трех лет. И вот однажды ночью местные

инженеры вручную осуществляли переход от базы данных 1 приложения CF к базе данных 2. Перенос рабочей базы данных с одного хостинга на другой выполнялся при помощи технологии Veritas. Это требовалось для плановых работ на первом сервере. Совершенно обычных работ. В прошлом подобная процедура проводилась десятки раз.

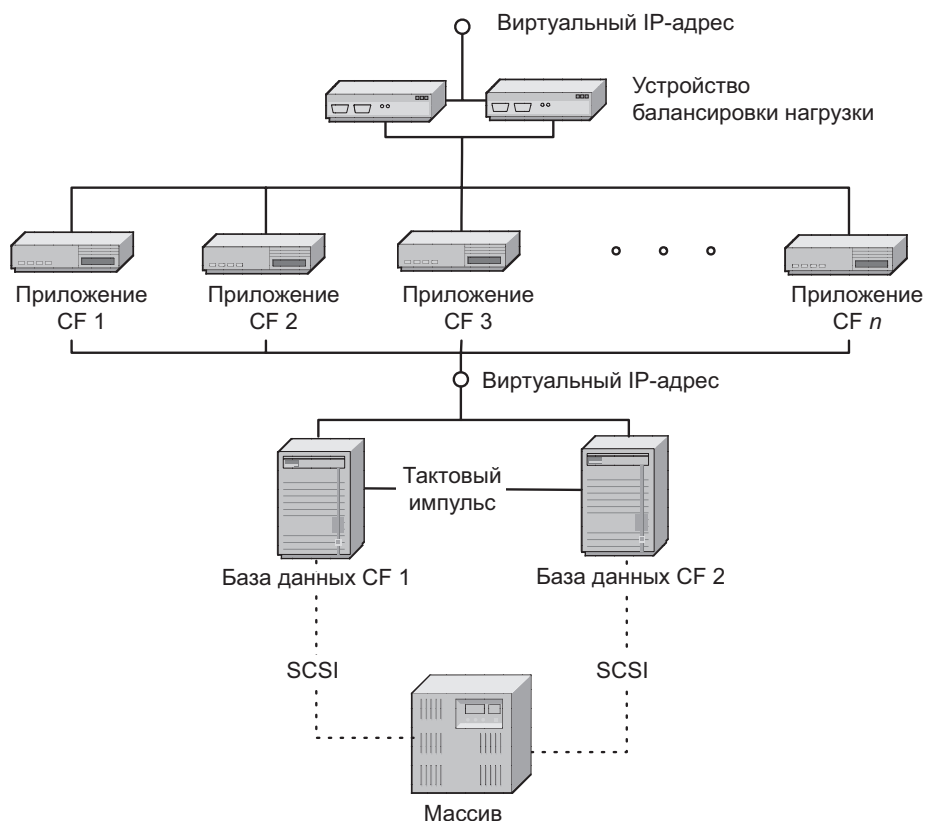


Рис. 1. Архитектура развертывания приложения CF

Перевод на другой ресурс осуществляло приложение Veritas Cluster Server. В течение минуты оно может остановить работу сервера Oracle с базой данных 1, размонтировать файловые системы с RAID-массива, повторно подключить их к базе данных 2, запустить там Oracle и назначить этой базе виртуальный IP-адрес. С точки зрения сервера приложений при этом вообще ничего не меняется, так как он настроен для соединения только с виртуальным IP-адресом.

Это конкретное изменение было запланировано на клиенте в четверг вечером, примерно в 11 часов по тихоокеанскому времени. Процессом руководил один из местных

инженеров через центр управления. Все шло по плану. Рабочей была сделана база данных 2, после чего выполнили обновление базы данных 1. Дважды проверив корректность обновления, базу данных 1 снова сделали рабочей, а аналогичные изменения были внесены в базу данных 2. При этом регулярный мониторинг сайта показывал, что приложение все время оставалось доступным. В данном случае простой не планировался, поэтому его и не было. В половине первого ночи команда поставила на изменение метку «завершено успешно» и вышла из системы. Инженер пошел спать после 22-часовой смены. В конце концов, он и так уже держался только за счет кофе.

Следующие два часа прошли в обычном режиме.

Примерно в половине третьего утра все киоски самостоятельной регистрации на консоли мониторинга засветились красным — они прекратили свою работу одновременно по всей стране. Через несколько минут то же самое произошло с IVR-серверами. Ситуация была близка к катастрофе, ведь 2:30 по тихоокеанскому времени — это 5:30 по восточному времени, когда начинается основной поток регистраций на рейсы, летящие к Восточному побережью. Центр оперативного управления немедленно забил тревогу и собрал группу поддержки на телефонную конференцию.

При любых нештатных ситуациях я первым делом стараюсь восстановить обслуживание. Это куда важнее расследования причин. Возможность собрать данные для последующего анализа первопричины проблемы бесценна, но только в случае, если это не увеличивает время простоя системы. В экстренной ситуации полагаться на импровизацию не стоит. К счастью, давным-давно существуют сценарии, создающие дампы потоков и снимки баз данных. Подобный автоматизированный сбор данных идеально подходит к таким ситуациям. Исчезает необходимость действовать экспромтом, время простоя не увеличивается, но при этом накапливается информация, которую потом можно будет проанализировать. По инструкции центр оперативного управления немедленно запустил эти сценарии, одновременно попытавшись запустить один из серверов с приложением для киосков самостоятельной регистрации.

ПОЛУЧЕНИЕ ДАМПОВ ПОТОКОВ

Любое Java-приложение генерирует дампы состояния любого потока в JVM при отправке сигнала 3 (SIGQUIT) в UNIX или нажатии комбинации клавиш Ctrl+Break в Windows.

При этом в операционной системе Windows вы должны пользоваться консолью с Java-приложением, запущенным в окне командной строки. Очевидно, что для входа с удаленного доступа вам потребуется система VNC или протокол Remote Desktop.

В операционной системе UNIX для отправки сигнала можно воспользоваться командой `kill`:

```
kill -3 18835
```

К сожалению, дампы потоков всегда отправляются на стандартное устройство вывода. Многие готовые запускающие сценарии не распознают стандартное устройство вывода или отправляют дампы в `/dev/null`. (Например, JBoss от Gentoo Linux по умолчанию отправляет отладочную информацию `JBoss_CONSOLE` в `/dev/null`.) Файлы журналов, полученные при помощи библиотеки `Log4J` или пакета `java.util.logging`, дампы потоков не отображают. Поэкспериментируйте с запускающими сценариями вашего сервера приложений, чтобы понять, каким образом создаются дампы потоков.

Вот небольшой фрагмент такого дампа.

```
"http-0.0.0.0-8080-Processor25" daemon prio=1 tid=0x08a593f0 \
  nid=0x57ac runnable [a88f1000..a88f1ccc]
    at java.net.PlainSocketImpl.socketAccept(Native Method)
    at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:353)
    - locked <0xac5d3640> (a java.net.PlainSocketImpl)
    at java.net.ServerSocket.implAccept(ServerSocket.java:448)
    at java.net.ServerSocket.accept(ServerSocket.java:419)
    at org.apache.tomcat.util.net.DefaultServerSocketFactory.\
acceptSocket(DefaultServerSocketFactory.java:60)
    at org.apache.tomcat.util.net.PoolTcpEndpoint.\
acceptSocket(PoolTcpEndpoint.java:368)
    at org.apache.tomcat.util.net.TcpWorkerThread.\
runIt(PoolTcpEndpoint.java:549)
    at org.apache.tomcat.util.threads.
ThreadPool$ControlRunnable.\
run(ThreadPool.java:683)
    at java.lang.Thread.run(Thread.java:534)

"http-0.0.0.0-8080-Processor24" daemon prio=1 tid=0x08a57c30 \
  nid=0x57ab in Object.wait() [a8972000..a8972ccc]
    at java.lang.Object.wait(Native Method)
    - waiting on <0xacede700> (a org.apache.tomcat.util.threads.\
ThreadPool$ControlRunnable)
    at java.lang.Object.wait(Object.java:429)
    at org.apache.tomcat.util.threads.
ThreadPool$ControlRunnable.\
run(ThreadPool.java:655)
    - locked <0xacede700> (a org.apache.tomcat.util.threads.\
ThreadPool$ControlRunnable)
    at java.lang.Thread.run(Thread.java:534)
```

Дампы бывают крайне подробными.

Этот фрагмент демонстрирует два потока, имена которых выглядят примерно так: `http-0.0.0.0-8080-ProcessorN`. Номер 25 — запущенный поток, в то время как номер 24 заблокирован в методе `Object.wait()`. Этот дампы ясно показывает, что вы имеете дело с членами пула потоков, и ключом к решению могут оказаться некоторые классы в стеке с именами `ThreadPool$ControlRunnable()`.

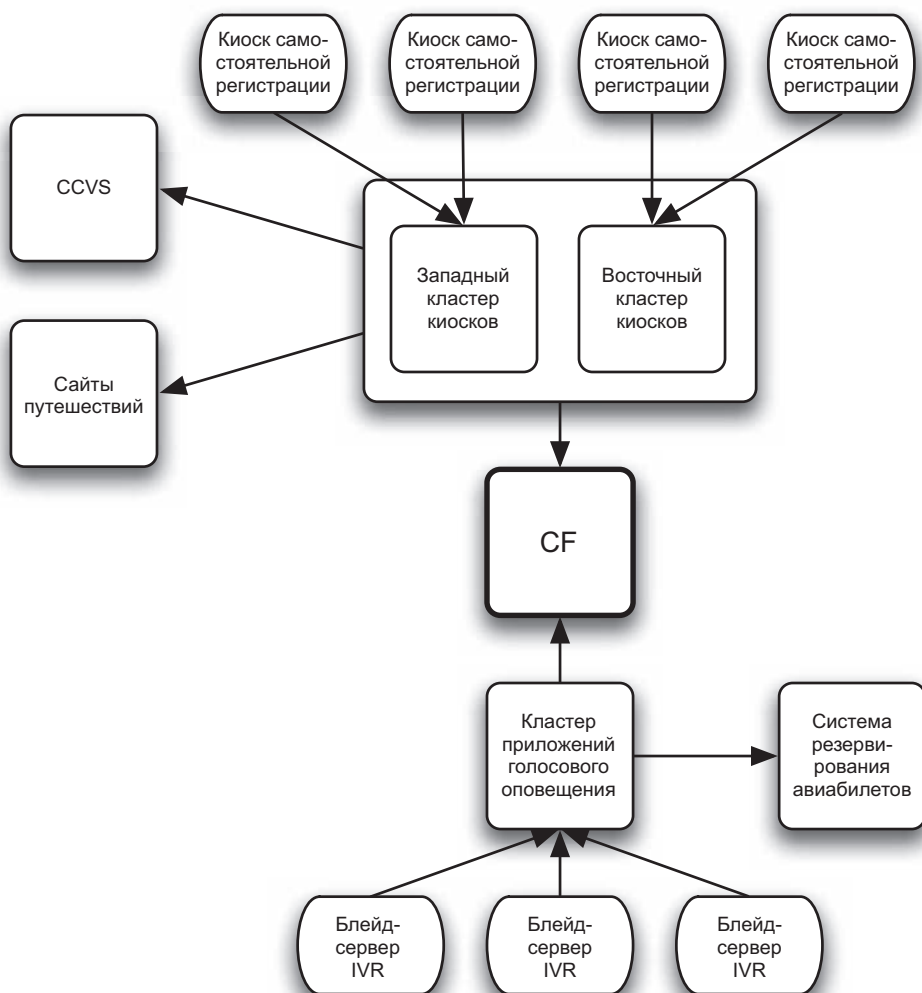
Для восстановления обслуживания важно правильно выбрать направление работ. Разумеется, всегда можно «перезагрузить мир», один за другим заново запуская все серверы. Эта стратегия почти всегда эффективна, но занимает *много* времени. В большинстве случаев причину проблем найти можно. В некотором смысле это как медицинская диагностика. Вы можете лечить пациента от всех существующих заболеваний, но это болезненно, дорого и долго. Вместо этого имеет смысл изучить симптомы и понять, что именно нужно лечить. Проблема в том, что отдельные симптомы недостаточно конкретны. Разумеется, порой симптом четко указывает на основную проблему, но это редкий случай. Чаще всего симптомы, например высокая температура, сами по себе ни о чем не говорят. Сотни заболеваний сопровождаются повышением температуры. И чтобы отличить одно от другого, нужны анализы или наблюдения.

В данном случае команда столкнулась с двумя наборами отказавших приложений. Они зависли практически одновременно, временной промежуток между этими событиями был настолько ничтожен, что вполне мог оказаться задержкой срабатывания инструментов слежения за киосками и IVR-приложениями. В подобном случае первым делом приходит на ум, что оба набора приложений зависят от некой третьей сущности, в работе которой произошел сбой. Но из рис. 2 следует, что единственным связующим звеном между киосками и IVR-системой была программа CF. Состоявшееся за три часа до инцидента переключение баз данных еще больше усиливало подозрения. Но мониторинг программы не выявил никаких проблем. Не дали результатов ни исследование журнала регистрации, ни проверка URL-адреса. При этом оказалось, что осуществлявшее мониторинг приложение попадало только на страницу состояния, поэтому информации о реальном состоянии серверов приложения CF было немного. Мы поместили, что позднее, когда появится действующий канал, эту ошибку нужно будет исправить.

Напоминаю, что главное в подобных ситуациях — восстановить работоспособность служб. В данном случае простой длился уже почти час, что в соответствии с соглашением об уровне обслуживания является максимально допустимым временем, поэтому рабочая группа решила перезагрузить все серверы приложения CF. Перезагрузка первого же сервера инициировала восстановление систем IVR. После загрузки остальных серверов информационно-справочная служба начала полностью функционировать, но киоски самостоятельной регистрации по-прежнему были выделены красным. По наитию главный инженер решил перезагрузить серверы их собственных приложений. Это помогло.

Всего на решение проблемы было потрачено чуть больше трех часов.

Соглашением об уровне обслуживания (Service Level Agreement, SLA) называется контракт между клиентом и поставщиком услуги, нарушение которого обычно ведет к значительным штрафам.

**Рис. 2.** Общие зависимости

2.2. Последствия

Может показаться, что три часа — это немного, особенно если вспомнить некоторые легендарные простои. (Например, в голову приходит отключение сайта eBay на целые сутки, происшедшее в 1999 году.) Но на работу авиакомпании этот инцидент влиял куда дольше. Сотрудников авиакомпании не хватало для обеспечения

нормального функционирования в отсутствие приложений. Выход из строя киосков самостоятельной регистрации заставил руководство вызвать работников, у которых в этот день был выходной. Некоторые из них уже отработали свои 40 часов в неделю, то есть по трудовому соглашению это рассматривалось как сверхурочная работа. Но даже дополнительные сотрудники были всего лишь людьми. И к моменту, когда они оказались на рабочих местах, им пришлось иметь дело с задержкой. Ликвидировать ее последствия получилось только к трем часам дня.

Именно столько времени заняла регистрация на утренние рейсы. Осуществляющие эти рейсы самолеты все еще стояли у своих выходов на посадку. Они были заполнены наполовину. Многие путешественники в этот день не смогли вовремя улететь или прилететь. Оказалось, что четверг сопровождается массовыми перелетами компьютерщиков: консультанты возвращаются в родные города. А так как выходы на посадку были заняты неулетевшими самолетами, прилетающие самолеты перенаправлялись к незанятым выходам. То есть неудобства испытали даже те, кто успел пройти регистрацию. Им пришлось в спешном порядке переходить к новому выходу на посадку.

Материал об этом происшествии был показан в телепрограмме *Доброе утро, Америка* (дополненный видеосюжетом про трогательно беспомощных одиноких мам с детьми) и в информации для туристов телеканала Weather Channel.

Федеральное управление гражданской авиации США требует включения информации о своевременности прилетов и вылетов в годовую отчетность авиакомпаний. Сюда же включаются присылаемые в управление потребительские жалобы.

И именно от этой отчетности частично зависит зарплата генерального директора. Думаю, вы понимаете, что день, когда в центр управления врывается генеральный директор, чтобы узнать, по чьей вине он не может себе позволить провести отпуск на Виргинских островах, не предвещает ничего хорошего.

2.3. Анализ причин сбоя

В 10:30 утра по тихоокеанскому времени, через восемь часов после начала инцидента наш сотрудник по связям с клиентами Том¹ пригласил меня для анализа причин сбоя. Так как отказ произошел вскоре после переключения баз и технического обслуживания, эти действия естественным образом попали под подозрение. В подобных случаях обычно исходят из посыла «после этого, значит, из-за этого»². Это далеко не всегда так, но, по крайней мере, дает отправную точку для размышлений.

¹ Имя изменено.

² Стандартная ошибка в логических выводах, заключающаяся в неверном представлении о том, что если событие Y следует сразу же за событием X, то X является причиной Y. Иногда это формулируется как «ты трогал это последним».

Вообще говоря, когда Том мне позвонил, он попросил меня приехать, чтобы выяснить, почему переключение базы данных привело к сбою.

Оказавшись в воздухе, я достал свой ноутбук и принялся изучать сообщение о проблеме и предварительный отчет о происшествии.

Повестка дня была проста: произвести анализ причин сбоя и ответить на следующие вопросы:

- ☐ Привело ли к сбою переключение базы данных? Если нет, что было его причиной?
- ☐ Корректной ли была конфигурация кластера?
- ☐ Корректно ли рабочая группа провела техническое обслуживание?
- ☐ Как можно было выявить отказ до того, как он привел к отключению?
- ☐ И, что самое важное, как мы можем гарантировать, что подобное никогда не повторится?

Разумеется, мое присутствие требовалось еще и затем, чтобы продемонстрировать клиенту, насколько серьезно мы относимся к происшедшему. Кроме того, мое расследование должно было снять опасения, что местная рабочая группа попытается замаять инцидент. Разумеется, они не стали бы делать ничего подобного, но создать правильное впечатление после крупного сбоя порой так же важно, как и разобраться с самим сбоем.

Создать правильное впечатление после крупного сбоя так же важно, как и разобраться с самим сбоем.

Искать причины сбоя постфактум — это все равно что расследовать убийство. У вас есть ряд улики. Некоторые из них вполне надежны: например, журналы регистрации событий сервера, скопированные с момента сбоя. К некоторым стоит относиться скептически: например, к утверждениям свидетелей. Ведь люди склонны приписывать к результатам наблюдений собственные догадки и представлять гипотезы как факты. Вообще говоря, анализировать причины сбоя тяжелее, чем расследовать убийство. У вас нет тела для вскрытия, так как серверы возвращены в состояние, предшествующее сбою, и работают. Состояния, ставшего причиной отказа, больше не существует. Следы отказа могут обнаружиться в журналах регистрации или среди собранных в этот момент данных мониторинга, а могут и не обнаружиться. Найти улики бывает крайне тяжело.

Работая с уже имеющейся информацией, я наметил, какие данные следует собрать. С серверов приложений мне требовались журналы регистрации, дампы потоков и конфигурационные файлы. С серверов баз данных требовались конфигурационные файлы баз данных и сервера кластера. Я также пометил, что нужно сравнить

текущие конфигурационные файлы с аналогичными файлами, полученными после ночного резервного копирования. Резервная копия была сделана до сбоя, а значит, я мог понять, были ли к моменту моего расследования внесены какие-либо изменения в конфигурацию. Другими словами, я узнал бы, не пытается ли кто-то скрыть ошибку.

К моменту прибытия в гостиницу мой организм говорил, что время уже позднее. Мне хотелось принять душ и завалиться в постель. Вместо этого меня поджидал менеджер с информацией об изменениях, происшедших, пока я был в воздухе. Мой рабочий день закончился только к часу ночи.

Утром, влив в себя изрядную порцию кофе, я закопался в конфигурации кластера базы данных и RAID-массива. Я искал следы наиболее распространенных проблем: недостаточное количество контрольных сигналов, прохождение контрольных сигналов через коммутаторы, по которым идет рабочий трафик, серверы настроены на работу с реальными IP-адресами вместо виртуальных, плохие зависимости между управляемыми пакетами и т. п. Контрольным списком я в тот раз не пользовался; просто припоминал проблемы, с которыми мне доводилось сталкиваться или о которых я слышал. Никаких неисправностей я не обнаружил. Группа инженеров отлично поработала с кластером базы данных. Надежный результат, хоть в учебник вставляй. Более того, казалось, что некоторые сценарии взяты непосредственно из учебных материалов фирмы Veritas.

Пришло время заняться конфигурацией серверов приложений. Инженеры скопировали все журналы регистрации серверов, обеспечивающих работу киосков, на время сбоя. Заодно я получил журналы регистрации серверов приложения CF. И все они содержали информацию, начиная с момента сбоя, ведь прошли всего одни сутки. И, что еще лучше, вместе с двумя наборами журналов в мои руки попали дампы потоков. Как человек, долгое время программирующий на языке Java, я обожаю пользоваться дампами потоков при отладке зависающих приложений.

Если вы умеете читать дампы, приложение становится открытой книгой. Вы можете многое узнать о приложении, кода которого никогда не видели. Вы узнаете, какими сторонними библиотеками оно пользуется, увидите его пулы потоков и количество потоков в каждом из них, выполняемые им фоновые задания. Обнаруженные в процессе трассировки стека каждого потока классы и методы скажут вам даже о том, какими протоколами пользуется приложение.

Выявление проблемы в приложении CF не заняло много времени. Дампы потоков серверов, обеспечивающих работу приложений для киосков самостоятельной регистрации, показали именно то, что я ожидал увидеть по описанию поведения системы во время инцидента. Все сорок потоков, выделенных для обработки запросов от отдельных киосков, были заблокированы внутри метода `SocketInputStream.socketRead0()` — нативного метода библиотеки сокетов Java. Все они пытались прочесть ответ, которого не было.

Эти дампы потоков также позволили мне узнать точное имя класса и метода, который пытались вызывать потоки: `FlightSearch.lookupByCity()`. Чуть выше в стеке я с изумлением обнаружил ссылки на RMI- и EJB-методы. Программу CF всегда называли «веб-службой». Конечно, в наши дни определение «веб-служба» трактуется достаточно широко, но в случае сеансового объекта без сохранения состояния оно выглядит большой натяжкой. Интерфейс вызова удаленных методов (Remote Method Invocation, RMI) предоставляет технологии EJB способ вызова удаленной процедуры. EJB-вызовы передаются по одному из двух механизмов: CORBA или RMI. При всей моей любви к программной модели RMI пользоваться ею небезопасно из-за невозможности добавить к вызовам таймаут. А значит, вызывающая функция слабо защищена от проблем на удаленном сервере.

2.4. Бесспорное доказательство

На данном этапе результат моего анализа вполне соответствует симптомам сбоя: именно программа CF вызвала зависание как IVR-системы, так и киосков самостоятельной регистрации. Но остался самый главный вопрос: что случилось с программой CF?

Картина прояснилась, когда я исследовал дампы ее потоков. Сервер приложения CF для обработки EJB-вызовов и HTTP-запросов пользовался отдельными пулами потоков. Именно поэтому приложение CF могло отвечать приложению мониторинга даже во время сбоя. Практически все HTTP-потоки были бездействующими, что вполне согласуется с поведением EJB-сервера. Но при этом EJB-потоки были отданы под обработку вызовов метода `FlightSearch.lookupByCity()`. По сути, все потоки всех серверов приложений блокировались в одной и той же строке кода: при попытке проверить связь с базой данных из пула ресурсов.

Конечно, это было косвенное доказательство, но с учетом переключения базы данных перед сбоем создавалось впечатление, что я на верном пути.

Дальше начиналась неопределенность. Мне нужно было увидеть код, но центр управления не имел доступа к системе управления исходным кодом. В рабочей среде были развернуты только двоичные файлы.

Как правило, это хорошая мера предосторожности, но в данный момент она доставила мне некоторые неудобства. Спросив администратора, как мне получить доступ к исходному коду, я столкнулся с противодействием. Учитывая масштаб простоя, можно вообразить количество обвинений, носившихся в воздухе в поисках подходящей головы. Отношения между центром управления и разработчиками, никогда не отличавшиеся особой теплотой, были натянутыми сильнее, чем обычно. Каждый был настороже, опасаясь нацеленного в его сторону обвиняющего перста.

Так и не получив доступа к исходному коду, я сделал единственное, что мог: взял двоичные файлы из рабочего окружения и произвел их декомпиляцию¹. Увидев код подозреваемого мной EJB-сервера, я понял, что нашел бесспорное доказательство. Вот этот сеансовый объект оказался единственным программным компонентом, который на данный момент реализовывала программа CF:

```
package com.example.cf.flightsearch;
...
public class FlightSearch implements SessionBean {

    private MonitoredDataSource connectionPool;

    public List lookupByCity(. . .) throws SQLException, RemoteException {
        Connection conn = null;
        Statement stmt = null;

        try {
            conn = connectionPool.getConnection();
            stmt = conn.createStatement();

            // Реализуем логику поиска
            // возвращаем список результатов
        } finally {
            if (stmt != null) {
                stmt.close();
            }

            if (conn != null) {
                conn.close();
            }
        }
    }
}
```

На первый взгляд метод сконструирован хорошо. Наличие блока `try..finally` указывает на желание автора вернуть системе ресурсы. Более того, такой блок фигурировал в некоторых учебниках по языку Java. К сожалению, у него был существенный недостаток.

Оказалось, что метод `java.sql.Statement.close()` может порождать исключение `SQLException`. Этого практически никогда не происходит. Драйвер Oracle поступает так, только обнаружив, что `IOException` пытается закрыть соединение, например при переходе базы данных на другой ресурс.

Предположим, JDBC-соединение создано до переключения. Используемый для создания соединения IP-адрес будет перенесен с одного хоста на другой, а вот

¹ Моим любимым инструментом для декомпиляции Java-кода остается JAD. Он работает быстро и точно, хотя и с трудом разбирается с кодом, написанным на Java 5.

текущее состояние TCP-соединений на хост второй базы данных переброшено не будет. При любой записи в сокет в конечном счете выбрасывается исключение `IOException` (после того, как операционная система и сетевой драйвер решают, что TCP-соединение отсутствует). Это означает, что любое JDBC-соединение в пуле ресурсов представляет собой мину замедленного действия.

Что поразительно, JDBC-соединение по-прежнему жаждет создавать запросы. Для этого объект соединения драйвера проверяет только собственное внутреннее состояние¹. Если JDBC-соединение решает, что подключение никуда не делось, оно создает запрос. Выполнение этого запроса при сетевом вводе-выводе приводит к исключению `SQLException`. Но закрытие этого запроса дает аналогичный результат, так как драйвер попытается заставить сервер базы данных освободить связанные с запросом ресурсы.

Коротко говоря, драйвер хочет создать объект `Statement`, который не может быть использован. Это можно считать программной ошибкой. Разработчики из авиакомпании явно упирали именно на это. Но вы должны извлечь из этого главный урок. Если спецификация JDBC позволяет методу `java.sql.Statement.close()` выбрасывать исключение `SQLException`, ваш код должен его обрабатывать.

В показанном неверном коде исключение, выбрасываемое закрывающим запросом, приводило к тому, что соединение не закрывалось, а значит, возникала утечка ресурсов. После сорока подобных вызовов пул ресурсов истощался, и все последующие вызовы блокировались в методе `connectionPool.getConnection()`. Именно это я увидел в дампах потоков программы CF.

Работа авиакомпании, обслуживающей весь земной шар, с сотнями самолетов и десятками тысяч сотрудников была парализована совершенно школьной ошибкой: единственным необработанным исключением `SQLException`.

2.5. Легче предупредить, чем лечить?

Когда маленькая ошибка приводит к таким огромным убыткам, естественной реакцией становится фраза «Это никогда не должно повториться». Но как предотвратить подобные вещи? Экспертизой кода? Только при условии, что кто-то из экспертов знаком с особенностями JDBC-драйвера в Oracle или вся экспертная группа часами проверяет каждый метод. Дополнительным тестированием? Возможно. После обнаружения данной проблемы группа выполнила тестирование в условиях повышенной нагрузки и получила ту же самую ошибку. Но при обычном тестировании

¹ Это может быть особенностью, присущей только JDBC-драйверам в Oracle. Я проводил декомпиляцию только базы на Oracle.

нагрузка на некорректный метод была недостаточной для возникновения проблемы. Другими словами, если заранее знать, где искать, то создать тест, обнаруживающий ошибку, очень просто.

Однако в конечном счете ожидания, что все до единой ошибки будут обнаружены и устранены на стадии тестирования, относятся скорее к мечтам. Неполадки будут возникать. А раз от них невозможно избавиться, нужно научиться минимизировать потери от них.

В данном случае хуже всего то, что ошибка в одной системе может распространиться на другие связанные с ней системы. Поэтому лучше задать другой вопрос: как сделать так, чтобы ошибки в одной системе не влияли ни на что другое? На каждом предприятии в наши дни имеется целая сеть взаимосвязанных и взаимозависимых систем. Они не могут — и не должны — допускать появления цепочек отказов. Поэтому мы рассмотрим паттерны проектирования, предотвращающие распространение подобных проблем.

3 Понятие стабильности

Новое программное обеспечение, как и свежий выпускник университета, появляется на свет полным оптимизма и внезапно сталкивается с суровыми жизненными реалиями. В реальном мире происходят вещи, которых обычно не бывает в лабораториях, и, как правило, вещи плохие. Лабораторные тесты придумываются людьми, знающими, какой ответ следует получить. Реальные тесты порой вообще не имеют ответов. Иногда они просто вызывают сбой вашего программного обеспечения.

Программное обеспечение предприятий должно быть скептически настроенным. Оно должно ожидать проблемы и никогда им не удивляться. Такое программное обеспечение не доверяет даже самому себе и ставит внутренние барьеры для защиты от сбоев. Оно отказывается вступать в слишком близкий контакт с другими системами, так как это может привести к повреждениям.

Программа Core Facilities, о которой шла речь в предыдущей главе, была недостаточно скептической. Как это часто бывает, рабочая группа слишком восхищалась новыми технологиями и усовершенствованной архитектурой. Сотрудники могли бы сказать много восторженных слов об использовании заемных средств и суммирующем воздействии нескольких факторов. Ослепленные денежными знаками, они не увидели, что пора остановиться, и свернули не в ту сторону.

Слабая стабильность приводит к значительным убыткам. К убыткам относится и упущенная выгода. Поставщики, которых я упоминал в главе 1, теряют за час просто 100 000 долларов, причем даже не в разгар сезона. Торговые системы могут потерять такую сумму из-за одной не прошедшей транзакции!

Эмпирическое правило гласит, что привлечение одного клиента обходится интернет-магазину в 25–50 долларов. Представьте, что он теряет 10 % из 5000 уникальных посетителей в час. Это означает, что на привлечение клиентов было впустую потрачено 12 500–25 000 долларов.

Потеря репутации менее осязаема, но не становится от этого менее болезненной. Ущерб от появившегося на бренде пятна проявляется не так быстро, как урон от потери клиентов, но попробуйте опубликовать в журнале BusinessWeek отчет о проблемах при работе с вашей фирмой в горячий сезон. Потраченные на рекламу образа миллионы долларов — рекламу ваших услуг в Интернете — могут за несколько часов превратиться в пыль из-за нескольких некачественных жестких дисков.

Нужная вам стабильность вовсе не обязательно должна стоить дорого. В процессе построения архитектуры, дизайна и даже низкоуровневой реализации системы существует масса моментов принятия решения, от которых будет зависеть окончательная стабильность системы. В такие моменты порой оказывается, что функциональным требованиям удовлетворяют два варианта (с упором на прохождение тестов контроля качества). Но при этом один приведет к ежегодным часам простоя системы, а второй нет. И что самое интересное, реализация стабильного проекта стоит столько же, сколько реализация нестабильного.

Реализация стабильного проекта стоит столько же, сколько реализация нестабильного.

3.1. Определение стабильности

Прежде всего я хотел бы дать определение некоторым терминам. *Транзакцией* (transaction) называется выполненная системой абстрактная часть работы. Это не имеет отношения к транзакциям в базах данных. Одна часть работы может включать в себя несколько транзакций в базе данных. Например, в интернет-магазинах распространен тип транзакции «заказ, сделанный клиентом». Такая транзакция занимает несколько страниц, зачастую включая в себя интеграцию с внешними сервисами, такими как проверка кредитных карт. Системы создаются именно для проведения транзакций. Если система обрабатывает транзакции только одного типа, она называется выделенной. *Смешанной нагрузкой* (mixed workload) называется комбинация различных типов транзакций, обрабатываемых системой.

Слово *система* я использую для обозначения полного, независимого набора аппаратного обеспечения, приложений и служб, необходимого, чтобы обработать транзакции для пользователей. Система может быть совсем небольшой, например состоящей из единственного приложения. А может быть огромной, многоуровневой сетью из приложений и серверов. Еще я обозначаю этим термином набор хостов, приложений, сетевых сегментов, источников питания и пр., обрабатывающий транзакцию от одного узла до другого.

Отказоустойчивая система продолжает обрабатывать транзакции, даже когда импульсные помехи, постоянная нагрузка или неисправность отдельных узлов нарушают нормальный процесс обработки. Именно такое поведение большинство подразумевает под словом *стабильность* (stability). Серверы и приложения не только продолжают функционировать, но и дают пользователю возможность закончить работу.

Термины *импульс* (impulse) и *нагрузка* (stress) пришли из машиностроения. Импульс подразумевает резкую встряску системы. Как будто по ней ударили молотком. Нагрузка же, наоборот, представляет собой силу, прикладываемую к системе длительное время.

Огромное число посещений страницы с информацией об игровой приставке Xbox 360, случившееся после слухов о распродаже, вызвало импульс. Десять тысяч новых сеансов в течение одной минуты выдержать трудно. Мощный всплеск посещаемости сайта является импульсом. Выгрузка в очередь 12 миллионов сообщений ровно в полночь 21 ноября — это тоже импульс. Подобные вещи ломают систему в мгновение ока.

В то же время медленный ответ от процессинговой системы кредитных карт, производительности которой не хватает на работу со всеми клиентами, означает нагрузку на систему. В механической системе материал под нагрузкой меняет свою форму. Это изменение формы называется *деформацией* (strain). Напряжение вызывает деформацию. То же самое происходит с компьютерными системами. Напряжение, возникающее из-за процессинга кредитных карт, приводит к тому, что деформация распространяется на другие части системы, что может приводить к странным последствиям, например увеличению расхода оперативной памяти на веб-серверах или превышению допустимой частоты ввода-вывода на сервере базы данных.

ПРОДЛЕНИЕ СРОКА СЛУЖБЫ

Основные опасности для долговечности вашей системы — это утечки памяти и рост объемов данных. Обе эти вещи убивают систему в процессе эксплуатации. И обе практически не обнаруживаются при тестировании.

Тестирование выявляет проблемы, что дает вам возможность их решить (вот почему я всегда благодарю тестеров моих программ, когда они обнаруживают очередную ошибку). Но по закону Мерфи случаются именно те вещи, на которые программа не тестировалась. То есть если вы не проверили, не падает ли программа сразу после полуночи или не возникает ли ошибка из-за нехватки памяти на сорок девятый час безотказной работы, именно тут вас подстерегает опасность. Если вы не проверяли наличие утечек памяти, проявляющихся только на седьмой день, через семь дней вас ждет утечка памяти.

Проблема в том, что в среде разработки приложения никогда не выполняются так долго, чтобы можно было обнаружить ошибки, проявляющиеся со временем. Сколько времени сервер приложений обычно функционирует у вас в среде разработки? Могу поклясться, что среднее время не превосходит продолжительности ситкома¹. При тестировании качества он может поработать подольше, но, скорее всего, будет перезагружаться по крайней мере раз в день, если не чаще. Но даже запущенное и работающее приложение не испытывает непрерывной нагрузки. В подобных средах продолжительные тесты — например, работа сервера в течение месяца с ежедневным трафиком — невозможны.

Впрочем, ошибки такого сорта во время нагрузочного тестирования, как правило, все равно не выявляются. Нагрузочный тест проводится некоторое время, а затем

¹ Если пропустить рекламу, а также начальные и конечные титры, оно составит примерно 21 минуту.

программа прекращает работу. Час нагрузочного тестирования стоит изрядную сумму, поэтому никто не просит провести недельную проверку. Команда разработчиков обычно пользуется общей корпоративной сетью, поэтому вы не можете каждый раз лишать всех сотрудников фирмы доступа к таким жизненно необходимым вещам, как электронная почта и Интернет.

Каким же образом выяснить ошибки данного типа? Единственное, что вы можете сделать до того, как они причинят вам проблемы при эксплуатации, — это собственноручно провести испытания на долговечность. По возможности выделите для разработчика отдельную машину. Запустите на ней JMeter, Marathon или другой инструмент нагрузочного тестирования. Не делайте ничего экстремального, просто все время выполняйте запросы. Также обязательно сделайте несколько часов временем бездействия сценариев, имитируя отсутствие нагрузки ночью. Это позволит протестировать таймауты пула соединений и фаервола.

Иногда создать полноценную среду тестирования не удастся по экономическим причинам. В этом случае попытайтесь проверить хотя бы самые важные части. Это все равно лучше, чем ничего.

В крайнем случае среда для тестирования долговечности программы сама собой возникнет в процессе ее эксплуатации. Ошибки в программе обязательно проявятся, но это не тот рецепт, который обеспечит вам счастливую жизнь.

Долговечная система способна обрабатывать транзакции *долгое время*. Какой период попадает под это определение? Зависит от конкретной ситуации. В качестве точки отсчета имеет смысл взять время между развертываниями кода. Если каждую неделю в производство вводится новый код, не имеет значения, сможет ли система проработать два года без перезагрузки. В то же время центры сбора данных в западной части штата Монтана не нуждаются в том, чтобы их вручную перезагружали раз в неделю.

Проведите тестирование срока службы. Это единственный способ обнаружить ошибки, проявляющиеся со временем.

3.2. Режимы отказов

К катастрофическому отказу могут привести как внезапные импульсы, так и чрезмерная нагрузка. Но в любом случае какой-то компонент системы начинает отказывать раньше, чем все остальное. В своей книге *Inviting Disaster* Джеймс Р. Чайлз называет это трещинами (cracks) системы. Он сравнивает сложную систему, находящуюся на грани отказа, со стальной пластинкой с микроскопической трещиной. Под нагрузкой эта трещина может начать расти быстрее и быстрее. В конечном счете скорость распространения дефекта превысит скорость звука, и металл с резким треском сломается. Фактор, послуживший триггером, а также способ распространения дефекта в системе вместе с результатом повреждения называют *режимом отказа* (failure mode).

В ваших системах в любом случае будут присутствовать различные режимы отказов. Отрицая неизбежность сбоев, вы лишаете себя возможности контролировать и сдерживать их. А признание неизбежности сбоев позволяет спроектировать реакцию системы в случае их возникновения. И как проектирующие автомобили инженеры создают *зоны деформации* — области, спроектированные специально, чтобы защитить пассажиров, поглотив в процессе своего разрушения силу удара, — вы можете реализовать безопасные режимы отказов, обеспечивающие защиту остальной части системы. Самозащита такого вида определяет надежность системы в целом.

Чайлз называет эти защитные меры *ограничителями трещин* (crackstoppers). Подобно тому как встраивание в автомобиль зон деформации поглощает импульс и сохраняет жизнь пассажирам, вы можете решить, какие программные компоненты системы не должны отключаться, и реализовать такие режимы отказов, которые защищают эти компоненты от выхода из строя. В противном случае однажды можно столкнуться с непредсказуемыми и, как правило, опасными вещами.

3.3. Распространение трещин

Применим вышесказанное к случаю в авиакомпании, о котором шла речь в предыдущей главе. В проекте Core Facilities режимы отказов отсутствовали. Проблема началась с необработанного исключения `SQLException`, но работа системы могла остановиться и по ряду других причин. Рассмотрим несколько примеров от низкоуровневых деталей до высокоуровневой архитектуры.

Так как пул был настроен таким образом, что блокировал потоки с запросами в отсутствие ресурсов, в конечном счете он заблокировал все обрабатывающие запросы потоки. (Это произошло независимо друг от друга на всех экземплярах серверов приложений.) Можно было настроить пул на открытие новых соединений в случае его истощения или после проверки всех соединений на блокировку вызывающих потоков на ограниченное время, а не навсегда. Любой из этих вариантов остановил бы распространение трещины. На следующем уровне проблема с единственным вызовом в программе CF привела к сбою вызывающих приложений на остальных хостах. Так как службы для CF построены на технологии Enterprise JavaBeans (EJB), для них используется RMI. А по умолчанию вызовы этого интерфейса лишены таймаута. Другими словами, вызывающие потоки блокировались в ожидании, пока будут прочитаны их ответы от EJB-компонентов программы CF. Первые двадцать потоков для каждого экземпляра получили исключения, точнее `SQLException`, в оболочке исключения `InvocationTargetException`, которое в свою очередь располагалось в оболочке исключения `RemoteException`. И после этого вызовы начали блокироваться.

Можно было написать клиент с таймаутами на RMI-сокетах¹. В определенный момент времени можно было принять решение строить веб-службы для CF на основе HTTP,

¹ Например, установив фабрику сокетов, вызывающую для всех создаваемых ею сокетов метод `Socket.setSoTimeout()`.

а не EJB. После этого осталось бы задать на клиенте таймаут на HTTP-запросы¹. Еще вызовы клиентов можно было настроить таким образом, чтобы блокированные потоки отбрасывались, вместо того чтобы заставлять обрабатывающий запрос поток делать внешний интеграционный вызов. Ничего из этого не было сделано, и сбой в программе CF распространился на все использующие эту программу системы.

В еще большем масштабе серверы программы CF можно было разбить на несколько групп обслуживания. В этом случае проблема с одной из групп не прекращала бы работу всех пользователей программы. (В рассматриваемой ситуации все группы обслуживания вышли бы из строя одним и тем же способом, но такое происходит далеко не всегда.) Это еще один способ остановить распространение сбоя на все предприятие.

Если взглянуть на вопросы архитектуры еще более масштабно, то программу CF можно было построить с применением очередей сообщений запрос-ответ. При этом вызывающая сторона в курсе, что ответа может и не быть. И ее действия в этом случае являются частью обработки самого протокола. Можно поступить еще более радикально, заставив вызывающую сторону искать рейсы, отбирая в пространстве кортежей записи, совпадающие с критерием поиска. А программа CF заполняла бы пространство кортежей записями с информацией о рейсах. Чем сильнее связана архитектура, тем выше шанс, что ошибка в коде распространится по системе. И наоборот, менее связанная архитектура действует как амортизатор, уменьшая влияние ошибки, а не увеличивая его.

Любой из перечисленных подходов мог прервать распространение проблемы с исключением `SQLException` на остальную часть системы аэропорта. К сожалению, проектировщики при создании общих служб не учли возможность появления «трещин».

3.4. Цепочка отказов

В основе любого сбоя в работе системы лежит такая же цепочка событий. Одна небольшая вещь ведет к другой, которая в свою очередь вызывает нечто третье. При рассмотрении такой цепочки отказов постфактум отказ кажется неизбежным. Но если попытаться оценить вероятность последовательного возникновения именно таких событий, ситуация покажется потрясающе неправдоподобной. Однако неправдоподобие возникает лишь потому, что мы рассматриваем все события независимо друг от друга. Монетка не обладает памятью; результат каждого ее броска выпадает с той же самой вероятностью, что и любого из предыдущих бросков. Комбинация же событий, приведших к отказу, независимой не является. Отказ в одной точке или слое повышает вероятность остальных отказов. Если база данных начинает работать медленно, *повышается* вероятность того, что серверам приложений не

¹ При условии, что там не используются компоненты `java.net.URL` и `java.net.URLConnection`. До появления Java 5 задать таймаут на HTTP-вызовы, осуществляемые средствами стандартной библиотеки Java, было невозможно.

хватит памяти. Так как все слои связаны друг с другом, происходящие на них события не являются независимыми.

На каждом шаге цепочки отказов формирование трещины можно ускорить, замедлить или остановить. Высокий уровень сложности дает дополнительные направления распространению трещины.

Сильная взаимозависимость ускоряет распространение трещин. К примеру, сильная взаимозависимость EJB-вызовов привела к тому, что проблема истощения ресурсов в SF-программе создала еще большие проблемы в вызывающих эту программу компонентах. В рамках подобных систем объединение потоков, занимающихся обработкой запросов, с вызовами внешних, интегрированных служб приводит к проблемам с удаленным доступом, которые, в свою очередь, приводят к простоям.

Одним из способов подготовки ко всем возможным отказам является учет всех внешних вызовов, всех операций ввода-вывода, всех использований ресурсов и всех ожидаемых результатов в поиске ответа на вопрос, какими способами может возникнуть сбой? Вспомните все возможные типы импульсов и напряжений:

- ☐ Что, если не удастся установить начальное соединение?
- ☐ Что, если установка соединения займет десять минут?
- ☐ Что, если после установки соединения произойдет разрыв связи?
- ☐ Что, если, установив соединение, я все равно не получу ответа?
- ☐ Что, если ответ на мой запрос займет две минуты?
- ☐ Что, если одновременно придут 10 000 запросов?
- ☐ Что, если мой диск переполнится в момент, когда я пытаюсь записать сообщение об ошибке `SQLException`, возникшей потому, что работа сети парализована вирусной программой?

Я уже устал писать вопросы, а ведь это только небольшая часть возможных вариантов. Следовательно, изнурительный перебор нецелесообразен, если только речь не идет о жизненно важных системах. А как быть, если программа должна быть готова еще в этом десятилетии? Нужно искать готовые паттерны, которые позволят вам создать амортизаторы, смягчающие все эти нагрузки.

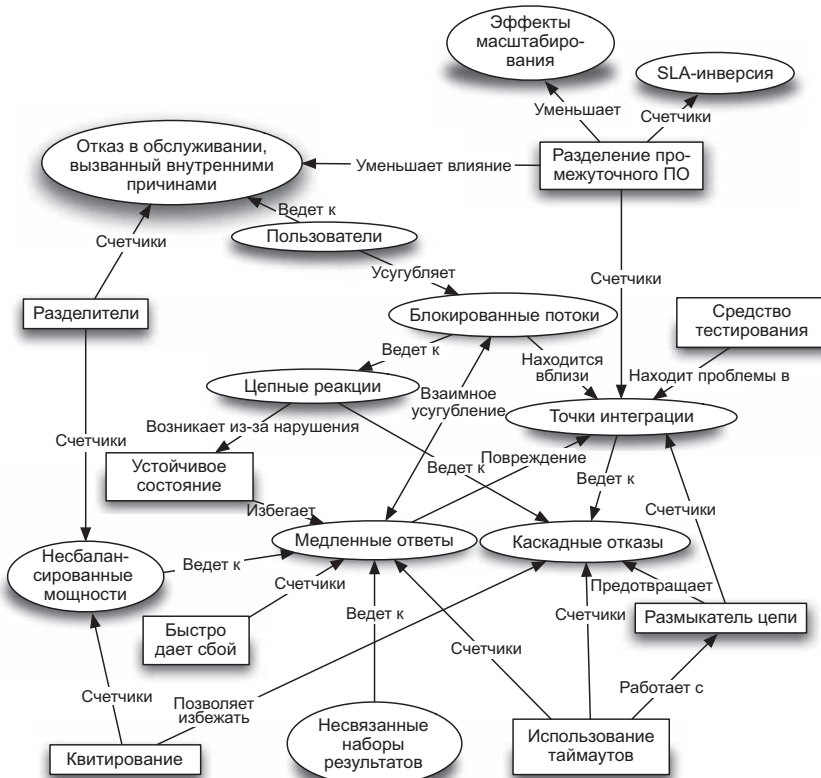
3.5. Паттерны и антипаттерны

Мне пришлось иметь дело с сотнями эксплуатационных отказов. Каждый из них был по-своему уникален. (В конце концов, я ведь старался, чтобы однажды возникший отказ больше никогда не повторялся!) Я не могу вспомнить ни одной пары инцидентов с одинаковыми цепочками отказов: с одними и теми же триггерами, одними и теми же повреждениями и одним и тем же распространением. Но со временем я заметил наличие неких паттернов. Определенная уязвимость,

тенденция *определенной* проблемы разрастаться *определенным* образом. С точки зрения стабильности эти паттерны отказов являются антипаттернами. Паттерны отказов рассматриваются в главе 4.

Наличие систематических закономерностей отказов заставляет предположить наличие распространенных решений. Предположив это, вы будете совершенно правы. Паттернам проектирования и архитектуры, позволяющим бороться с антипаттернами, посвящена глава 5. Разумеется, они не могут предотвратить трещины в системе. Подобного средства пока попросту не существует. Всегда будут некие условия, приводящие к сбою. Но эти паттерны останавливают распространение сбоя. Они помогают сдерживать разрушения и сохранить частичную функциональность.

Вполне логично, что эти паттерны и антипаттерны взаимодействуют. Последние имеют тенденцию усиливать действие друг друга. И подобно набору из чеснока, серебра и огня для борьбы с соответствующими киношными монстрами¹, каждый из паттернов облегчает последствия возникновения определенных проблем. Следующий рисунок демонстрирует наиболее важные из этих взаимодействий. А мы перейдем к рассмотрению антипаттернов — распространенного источника отказов.



¹ Это вампиры, оборотни и Франкенштейн.

4 Антипаттерны стабильности

Раньше сбой приложения был одним из самых распространенных типов ошибок, а второе место занимали сбои операционной системы. Я мог бы ехидно заметить, что к настоящему моменту практически ничего не изменилось, но это было бы нечестно. Сбои приложений в наши дни происходят относительно редко благодаря широкому внедрению Java, PHP, Ruby и других интерпретируемых языков. Тяжелая работа множества программистов сделала операционные системы более стабильными и надежными. Раньше крупной называлась система, которой одновременно пользовались сотни человек, сейчас число пользователей крупных систем измеряется тысячами. Непрерывная безотказная работа приложений измеряется теперь не часами, а месяцами. Резко увеличилась широта охвата, ведь сначала системы интегрировались в рамках отдельных предприятий, а потом и между предприятиями.

Разумеется, это означает большие проблемы. В мире интеграции тесно связанные системы становятся скорее правилом, чем исключением. Крупные системы обслуживают большее количество пользователей, управляя большим числом ресурсов; но во многих режимах отказов они быстрее выходят из строя, чем небольшие системы. Масштаб и сложность этих систем выдвигают нас на передовые рубежи технологии, где комбинация чрезвычайной сложности взаимодействий и сильной взаимозависимости норовит превратить быстро распространяющиеся трещины в полномасштабные отказы.

Чрезвычайная сложность взаимодействий возникает при наличии в системе достаточного количества движущихся частей и скрытых внутренних зависимостей, представление о которых у человека-оператора является неполным или откровенно неверным. В своей книге *Design of Everyday Things* («Дизайн привычных вещей») (

Дональд Норман описывает разрыв между ментальной моделью пользователей и моделью реализации, возникающий, когда принцип реализации непонятен, а внешний вид неочевиден. Норман рассказал про регуляторы в холодильнике, которые, как ему показалось, давали возможность напрямую задавать температуру в холодильной и морозильной камерах. Руководствуясь возникшей у него ментальной моделью, он получил замороженное молоко и растаявшее мясо, так как на самом деле механизм регулировал долю охлажденного воздуха, посылаемую в каждую из камер. В проявляющих чрезвычайно сложные взаимодействия системах интуитивные действия оператора в лучшем случае будут неэффективными, а в худшем окажут пагубное влияние. Из самых благих побуждений оператор может предпринять основанные на его представлениях о функционировании системы действия, которые запустят неожиданную цепочку связанных друг с другом событий. Подобные цепочки вносят вклад в *нарастание проблемы* (problem inflation), превращая небольшую неисправность в серьезный инцидент. Именно такая цепочка в системах охлаждения, наблюдения и управления частично стала причиной повреждения активной зоны реактора на атомной станции Три-Майл-Айленд¹. Зачастую эти скрытые взаимосвязи выглядят совершенно очевидными при анализе последствий аварии, но на самом деле предусмотреть их практически невозможно.

Сильная взаимозависимость позволяет неисправностям из одной части системы распространиться по всей системе. На физическом уровне можно представить монтажную платформу, подвешенную к потолку на четырех врезанных в металлическую пластину болтах. Очевидно, что платформа, гайки, болты, пластина и потолок тесно связаны. (При этом вся система держится на болтах!) Отказ одного болта резко увеличивает нагрузку на остальные болты, потолок и платформу. И эта возросшая нагрузка повышает вероятность отказа следующего компонента — возможно, самой платформы. В компьютерных системах сильная взаимозависимость может иметь место в коде приложения, в межсистемных вызовах и в любом ресурсе, потребляемом множеством других ресурсов.

В этой главе вы прочитаете про одиннадцать антипаттернов стабильности, которые я наблюдал своими глазами. Эти антипаттерны — распространенные факторы отказа множества систем. Некоторые из них напоминают человека, который приходит к врачу и говорит: «Доктор, когда я так делаю, мне больно», после чего бьет себя по голове молотком. Доктор может только сказать: «Не делайте так!». Каждый из них порождает, увеличивает и умножает трещины в системе. А значит, их следует всячески избегать.

Антипаттерны создают, увеличивают и множат трещины в системе.

Но во всех случаях главное — помнить, что системы имеют свойство ломаться. Не обольщайтесь, что сможете устранить все возможные источники отказов,

¹ https://ru.wikipedia.org/wiki/Авария_на_АЭС_Три-Майл-Айленд. — *Примеч. пер.*

так как форс-мажорные обстоятельства или человеческий фактор предугадать невозможно. Поэтому лучше сразу готовиться к худшему, ведь от отказов никуда не деться.

4.1. Точки интеграции



С проектом, состоящим исключительно из веб-сайта, я не сталкивался с 1996 года. Если вы занимаетесь такими же проектами, как и я, то, скорее всего, это проекты, интегрированные на уровне предприятия с входным интерфейсом на основе HTML. На самом деле, что бы там ни говорилось, взгляд на интеграцию поменялся, только когда возникла необходимость перехода к динамическим сайтам. Именно это стало импульсом, заставившим многие компании заняться интеграцией плохо совместимых друг с другом систем. Если посмотреть на контекстную диаграмму системы любого подобного проекта, обнаружится расположенный в центре сайт, от которого отходят многочисленные стрелочки. Веб-каналы формируются инвентарными ведомостями, расчетом цен, управлением содержимым, системами CRM, ERP, MRP, SAP, WAP, BAP, BPO, R2D2 и C3P0. Данные «на лету» извлекаются для передачи их в CRM, заполнения форм, бронирования, авторизации, проверки легитимности, нормализации адресов, планирования, отгрузки и т. п. Генерируются отчеты, демонстрирующие бизнес-статистику бизнесменам, техническую статистику — техническим специалистам, статистику предприятия — управляющему персоналу.

Главными убийцами систем являются точки интеграции. Каждый из упомянутых веб-каналов ставит под угрозу стабильность системы. Любой сокет, процесс, канал или удаленный вызов процедуры может зависнуть — и обязательно это сделает. Зависшим может оказаться даже обращение к базе данных, причем добиться этого можно как очевидными, так и неочевидными способами. Любой веб-канал с системой может подвесить ее, вызвать ее сбой или сгенерировать другую проблему в наименее подходящий для этого момент времени. Поэтому рассмотрим, какими способами эти точки интеграции способны причинить нам неприятности и что мы можем этому противопоставить.

ЧИСЛО ВЕБ-КАНАЛОВ

Я помогал запустить проект по переходу на другую платформу/к другой архитектуре для крупного предприятия розничной торговли. Пришло время определить все правила фаервола и разрешить санкционированный доступ к рабочей системе. Мы уже рассмотрели обычные варианты соединения: веб-серверов с сервером приложений, сервера приложений с сервером баз данных, диспетчера кластера с узлами кластеров и т. п.

Когда пришло время добавлять правила для входящих и исходящих веб-каналов в рабочей среде, нам указали на руководителя проекта, отвечающего за интеграцию приложений предприятия. Да, именно так, по проекту перестройки сайта работал

человек, отвечающий за интеграцию. Это еще раз показало нам нетривиальность задачи. (Первым намеком был тот факт, что никто, кроме этого человека, не смог назвать общее количество веб-каналов.) Руководитель проекта точно понял, что нам требуется. Он вошел в базу данных интеграции и запустил процедуру формирования отчета с данными о деталях соединений.

С одной стороны, я был впечатлен наличием базы слежения за различными веб-каналами (синхронный/асинхронный, пакетный или поточный, система-источник, частота, объем, номера перекрестных ссылок, заинтересованная сторона и т. д.), с другой — меня ужаснуло то, что для этой цели потребовалась целая база.

Неудивительно, что загруженный сайт постоянно имел проблемы со стабильностью. Ситуация напоминала присутствие в доме новорожденного ребенка; меня то и дело будили в 3 часа утра сообщениями об очередном сбое. Мы записывали, где именно возникла проблема с приложением, и передавали систему в группу технического обслуживания. Статистики подобных случаев у меня нет, но я уверен, что каждая точка интеграции вызывает по крайней мере один сбой.

Сокет-ориентированные протоколы

Многие высокоуровневые протоколы интеграции реализуются через сокеты. На самом деле, сокеты являются основой всего, кроме именованных каналов и взаимодействия процессов в общей памяти. Протоколы более высокого уровня обладают собственными режимами отказов, но подвержены и отказам на уровне сокетов.

Простейший режим отказа возникает, когда удаленная система не хочет устанавливать соединение. Вызывающей системе нужно что-то сделать с отказом соединения. Обычно это не составляет проблемы, ведь во всех языках от C до Java и Ruby присутствует простое средство обозначения такого отказа — возвращаемое значение —1 в C и исключения в Java, C# и Ruby. Так как API четко дает понять, что соединение возникает не всегда, программисты умеют бороться с этой проблемой.

Но следует обратить внимание на еще один аспект. Чтобы обнаружить невозможность соединения, иногда требуется *долгое* время. Не поленитесь как следует ознакомиться с особенностями функционирования сетей TCP/IP.

Любая архитектурная схема рисуется при помощи прямоугольников и стрелок, как показано на рис. 3. Подобно множеству других вещей, с которыми нам приходится иметь дело, каждая такая стрелка является абстракцией для сетевого соединения. При этом, по сути, это абстракция абстракции. Сетевое «соединение» само по себе является логической структурой — абстракцией. Все, что мы можем увидеть в сети, — это пакеты¹. Это работа маршрутизируемого протокола сетевого уровня (IP), входящего в стек TCP/IP. Протокол управления передачей (Transmission Control

¹ Разумеется, «пакет» — тоже абстракция. Это просто электроны, бегущие по проводам. Между электронами и TCP-соединением лежит несколько уровней абстракции. К счастью, в любой момент времени мы можем выбрать любой удобный для нас уровень абстракции.

Protocol, TCP) представляет собой соглашение о способе создания некой сущности, которая выглядит как непрерывное соединение, состоящее из дискретных пакетов. Рисунок 4 демонстрирует «трехстороннее квитирование», которое определяет протокол TCP, чтобы открыть соединение.

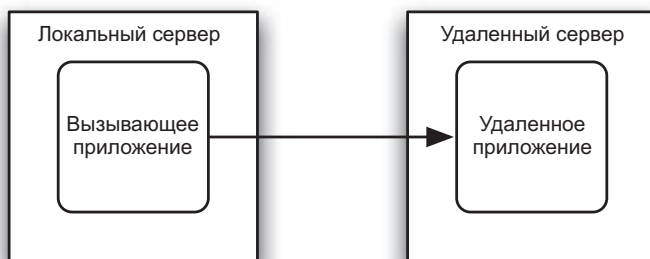


Рис. 3. Простейшая топология: прямое соединение

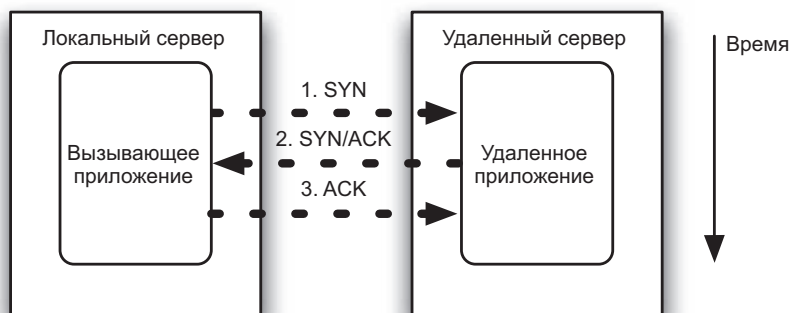


Рис. 4. Трехстороннее квитирование

Соединение начинает устанавливаться, когда вызывающая программа (в рассматриваемом сценарии это клиент, хотя сам по себе он является сервером для других приложений) отправляет на порт на удаленном сервере пакет с флагом SYN. Если на этом порту никто не слушает, удаленный сервер немедленно отправляет обратно пакет TCP «сброса». Вызывающее приложение получает исключение или неверное возвращаемое значение. Если обе машины подсоединены к одному коммутатору, все это происходит очень быстро, менее чем за десять миллисекунд.

Если же в порту назначения присутствует слушающее приложение, удаленный сервер отправляет назад пакет с флагом SYN/ACK, обозначая свою готовность к установке соединения. Получив этот пакет, вызывающее приложение

отправляет собственный пакет с флагом ACK. Эти три пакета устанавливают «соединение», что дает приложениям возможность пересылать данные в обоих направлениях¹.

Предположим, удаленное приложение слушает порт, но количество запросов на соединение столь велико, что оно попросту не может их обслужить. При этом порт обладает *очередью прослушивания* (listen queue), которая определяет, сколько ожидающих соединений (запрос с флагом SYN послан, а обратного запроса с флагом SYN/ACK нет) допускает сетевой стек. Как только эта очередь переполняется, все дальнейшие попытки соединений отклоняются. Это самое слабое место. Когда сокет находится в частично сформированном состоянии, все вызвавшие метод `open()` потоки блокируются в ядре операционной системы, пока удаленное приложение не соизволит, наконец, принять соединение или пока не наступит блокировка по превышению лимита времени. Время ожидания подключения зависит от операционной системы, но, как правило, этот параметр измеряется в *минутах*! Поток вызывающего приложения может быть заблокирован в ожидании ответа от удаленного сервера на десять минут!

Примерно то же самое происходит, когда вызывающая сторона может создать соединение и отправить запрос, но серверу требуется долгое время на чтение этого запроса и отправку ответа. Вызов метода `read()` блокируется до получения ответа от сервера. В языке Java по умолчанию выполняет бессрочная блокировка. Для прерывания заблокированного вызова нужно воспользоваться методом `Socket.setSoTimeout()`. В этом случае будьте готовы к исключению `IOException`.

Сетевые сбои могут влиять на вас двумя способами: быстро или медленно. В первом случае в вызывающем коде сразу же появляется исключение. Отказ типа «connection refused» («отказ в соединении») приходит через несколько миллисекунд. Медленные отказы, например сброшенный флаг ACK, приводят к тому, что до появления исключения блокировка потока длится несколько минут. Заблокированный поток не может обрабатывать другие транзакции, значит, падает общая производительность системы. При блокировке всех потоков сервер прекращает выполнять полезную работу. Так что медленный ответ куда хуже отсутствия ответа.

Проблема пяти часов утра

Один из сайтов, в запуске которых я принимал участие, породил крайне неприятный паттерн, приводивший каждый день в 5 часов утра к полному зависанию. Сайт был запущен примерно на тридцати серверах приложений, и нечто заставляло

¹ TCP-протокол также допускает «одновременное открытие», при котором обе машины перед тем, как обменяться пакетами с флагами SYN/ACK, посылают друг другу пакеты с флагом SYN. В клиент-серверных системах подобное встречается относительно редко.

их все зависать на пять минут (именно с такой частотой работала наша программа, проверяющая, предоставляется ли определенный URL-адрес). Проблема решалась перезагрузкой серверов приложений, так что приостановка работы сайта была временной. К сожалению, она происходила именно в тот момент, когда начинал расти трафик. От полуночи до 5 утра за час совершалось только около 100 транзакций, но как только в сети появлялись пользователи с Восточного побережья, это число резко увеличивалось. Перезагрузку всех серверов приложений в момент, когда на сайте появляется серьезный поток посетителей, сложно отнести к оптимальным вариантам.

На третий день я взял дампы потоков одного из затронутых серверов приложений. Сам сервер был давно поднят и благополучно работал, но все обрабатывающие запросы потоки оказались заблокированными в библиотеке Oracle JDBC, точнее внутри OCI-вызовов. (Мы пользовались драйвером толстого клиента, характеризующимся отличной отказоустойчивостью.) На самом деле как только я отключил потоки, заблокированные при попытке входа в синхронизированный метод, создалось впечатление, что все активные потоки были заняты низкоуровневыми сокетными вызовами чтения или записи.

Затем пришло время программ `tcpdump` и `ethereal`¹. Как ни странно, они показали крайне мало. Серверы приложений отправили на серверы базы данных несколько пакетов, но ответов не получили. Никакой передачи информации от базы данных к серверам приложений не было. Но мониторинг показал, что база в полном порядке. Никаких блокировок, пустая очередь выполнения и обычная скорость ввода-вывода.

ЗАХВАТ ПАКЕТОВ

Абстракции позволяют выражаться лаконично. Намного проще и быстрее сказать, что мы загружаем документ по URL-адресу, не вдаваясь в скучные подробности, касающиеся настройки соединения, фрагментации пакетов, подтверждений приема, окон получения и т. п. Однако при работе с любой абстракцией порой наступает время, когда приходится углубляться в детали. Обычно это происходит в случае сбоев. Как для диагностики неисправностей, так и для повышения производительности программы захвата пакетов являются единственным средством понять, что происходит в сети.

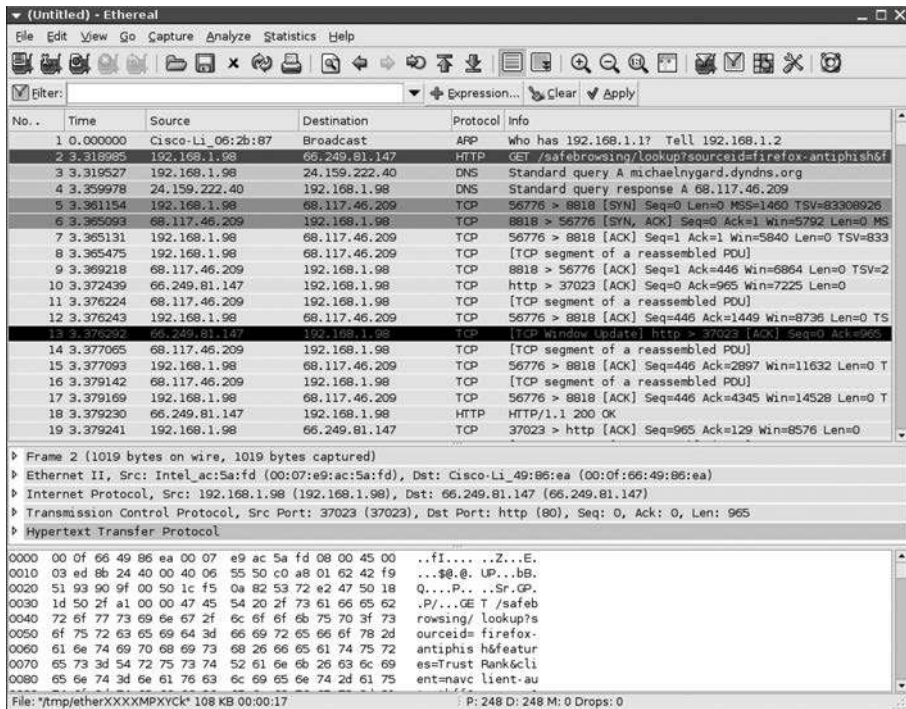
Программа `tcpdump` относится к UNIX-утилитам, позволяющим перехватывать и анализировать сетевой трафик. Ее запуск в «неразборчивом» режиме дает команду сетевой карте принимать все пакеты независимо от того, кому они адресованы. (В центрах обработки и хранения данных сетевая карта практически всегда соединяется с портом коммутатора, который относится к виртуальной локальной компьютерной сети (Virtual LAN, VLAN). Это гарантирует, что карта будет получать пакеты, привязанные к адресам только данной сети. Эта важная мера безопасности мешает злоумышленникам делать ровно то, чем мы занимались: сканировать сеть, вынюхивая «интересные» фрагменты информации.) Программа `Wireshark`² представляет собой комбинацию анализаторов

¹ Инструмент `Ethereal` теперь называется `Wireshark`.

² <http://www.wireshark.org>.

трафика и протоколов. Она может перехватывать и анализировать для вас пакеты, как и программа tcpdump. Но в отличие от последней она умеет распаковывать эти пакеты. За время своего существования программа Wireshark выявила множество брешей в системе безопасности, как небольших, так и серьезных. В определенный момент специально созданный пакет, посланный по сети (например, как фрагмент вредоносной программы на взломанном компьютере), мог вызвать переполнение буфера и запустить необходимый атакующему код. Так как для перевода сетевой карты в неразборчивый режим в программе Wireshark, как и в любой программе захвата пакетов, требуются права суперпользователя, этот эксплойт позволял атакующему получить соответствующие права на доступ к машине администратора сети.

Помимо проблем с безопасностью, программа Wireshark обладает большим и сложным GUI. В операционной системе UNIX для нее требуется множество X-библиотек (которые зачастую не устанавливаются в систему, работающую без монитора). Анализ и отображение пакетов на любой машине требует большого количества оперативной памяти и циклов процессора. На производственных серверах подобная нагрузка недопустима. По этим причинам пакеты лучше перехватывать неинтерактивно программой tcpdump, производя анализ полученного файла вне рабочей среды.



На показанном здесь снимке экрана программа Ethereal анализирует данные, полученные из моей домашней сети. Первый пакет демонстрирует запрос протокола определения адреса (Address Routing Protocol, ARP). Это запрос от канала беспроводной связи к кабельному модему. Следующий пакет оказался неожиданным: HTTP-запрос с набором параметров, отправленный в Google с URL-адресом /safebrowsing/lookup. Следующие два пакета соответствуют DNS-запросу к хосту michaelnygard.dyndns.org

и полученному оттуда ответу. Пакеты пять, шесть и семь относятся к трехстороннему квитированию для установления TCP-соединения. Можно проследить весь диалог между моим веб-браузером и сервером. Обратите внимание, что панель, расположенная под информацией о трассировке пакетов, показывает уровень инкапсуляции, созданный TCP/IP-стеком вокруг HTTP-запроса во втором пакете. Снаружи находится Ethernet-пакет, содержащий IP-пакет, внутри которого, в свою очередь, находится TCP-пакет. И наконец, в теле TCP-пакета — HTTP-запрос. Нижняя панель приложения показывает, сколько байтов содержит каждый пакет.

Настоятельно рекомендую для подобных исследований книгу «The TCP/IP Guide» Чарльза М. Козиеро или «TCP/IP Illustrated» Ричарда Стивенса.

Но на этот раз нам пришлось перезагрузить серверы приложений. Ведь первоочередной задачей было восстановление обслуживания. Мы выполняем сбор данных, где только можно, но не под угрозой нарушения соглашения об уровне обслуживания. С более детальным расследованием следовало подождать до следующего инцидента. Никто из нас не сомневался в том, что ситуация повторится.

Соглашение об уровне обслуживания (Service Level Agreement, SLA) — это договорное обязательство, содержащее описание услуги, права и обязанности сторон и, самое главное, согласованный уровень качества предоставления данной услуги. Нарушение SLA приводит к финансовым санкциям.

Разумеется, на следующее утро все произошло по тому же самому паттерну. Серверы приложений прекратили свою работу, а потоки оказались внутри JDBC-драйвера. На этот раз мне удалось посмотреть на трафик в сети баз данных. Ничего. Совсем ничего. Полное отсутствие трафика на этой стороне фаервола напомнило мне собаку, которая *не* лаяла ночью, из рассказов о Шерлоке Холмсе — главным ключом была именно нулевая активность. У меня появилась гипотеза. Быстрая декомпиляция класса, описывающего пул ресурсов сервера приложений, показала, что эта гипотеза вполне правдоподобна.

Как я уже отмечал, подключение через сокет является абстракцией. Такие соединения существуют только в виде объектов в памяти компьютеров, находящихся в конечных точках. И после установки TCP-соединение может существовать много дней без прохождения через него *хотя бы одного пакета* с какой-либо из сторон¹. Соединение существует, пока оба компьютера помнят состояние сокета. Могут меняться маршруты, разрываться и снова устанавливаться физические соединения. Все это не имеет значения; «соединение» существует, пока два конечных компьютера думают, что оно установлено.

Было время, когда все это прекрасно работало. Но в наши дни наличие кучи маленьких крепостей, страдающих чрезмерной подозрительностью, полностью изменило философию и реализацию Интернета в целом. Я имею в виду фаерволы.

¹ При условии, что вы задали в ядре приемлемо неверные таймауты.

Фаервол представляет собой всего лишь специализированный маршрутизатор. Он направляет пакеты от одного набора сетевых портов к другому. Внутри каждого фаервола набор списков контроля доступа задает правила, разрешающие определенные соединения. Например, правило может звучать так: «допустимы соединения с адресов от 192.0.2.0/24 до 192.168.1.199 с портом 80». При виде входящего пакета с флагом SYN фаервол проверяет его в базе правил. Пакет может быть пропущен (направлен в сеть назначения), отвергнут (TCP-пакет сброса отправляется в исходный пункт) или проигнорирован (оставлен без ответа). Если соединение разрешено, фаервол делает в своей внутренней таблице запись вида «192.0.2.98:32770 соединен с 192.168.1.199:80». После этого все пакеты в любом направлении, совпадающие с конечными точками соединения, будут направляться между сетями, обслуживаемыми фаерволом.

Какое отношение вся эта информация имеет к звонкам, каждый день будившим меня в пять утра?

Ключом является таблица соединений внутри фаервола. Она конечна. Поэтому она не допускает бесконечно длящихся соединений, хотя протокол TCP их разрешает. Фаервол сохраняет информацию не только о конечных точках соединения, но и о времени прохождения «последнего пакета». Если пакеты долгое время не передаются, фаервол предполагает, что конечные точки прекратили свою работу или были удалены. И выбрасывает соединение из таблицы, как показано на рис. 5. Но при проектировании протокола TCP не учитывалась возможность появления в середине соединения подобного интеллектуального устройства. Поэтому способа сообщить конечным точкам о разрыве соединения не существует. В результате конечные точки предполагают, что соединение действительно бесконечно долго, даже если по нему не было передано ни одного пакета.

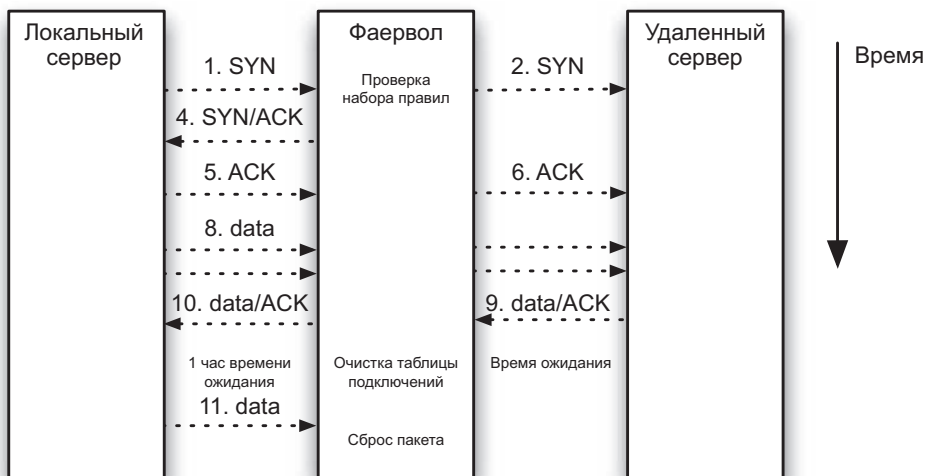


Рис. 5. Простаивающее соединение, сброшенное фаерволом

После этого любая попытка чтения из сокета или записи в сокет с любого конца не будет приводить ни к сбросу TCP, ни к ошибке, потому что сокет открыт только наполовину. Вместо этого стек TCP/IP отправит пакет, подождет ответа с флагом ACK, а не получив его, выполнит повторную передачу. Надежный стек снова и снова пытается восстановить контакт, а фаервол отфильтровывает запросы, не снисходя до сообщения «ICMP-объект назначения недоступен». (Это дает злоумышленникам возможность сканировать активные соединения, имитируя адреса отправителей.) В моей операционной системе Linux с ядром версии 2.6 переменная `tcp_retries2` имеет предлагаемое по умолчанию значение 15, что соответствует *двадцатиминутному* таймауту, после которого стек TCP/IP информирует библиотеку сокетов о разрыве соединения. Серверы HP-UX, которые мы использовали в то время, имели таймаут в тридцать минут. Попытка приложения сделать запись в сокет могла блокироваться на полчаса! Еще хуже обстояло дело с чтением из сокета. Оно блокировалось навсегда.

При декомпиляции класса пула ресурсов я обнаружил, что там использовалась стратегия «пришел последним, обслужен первым». По ночам объем трафика был настолько невелик, что одно соединение с базой данных захватывалось из пула, использовалось и возвращалось обратно. Следующий запрос мог задействовать то же самое соединение, оставив остальные тридцать девять ожидать возрастания трафика. В итоге было превышено часовое время ожидания соединения, заданное при настройке фаервола.

Как только трафик начинал возрастать, эти тридцать девять соединений с сервером приложений немедленно блокировались. Даже если для обслуживания страниц до сих пор использовалось одно соединение, рано или поздно его захватывал поток, в конце концов блокировавшийся из какого-нибудь другого пула. После этого единственное работающее соединение удерживалось заблокированным потоком. Сайт зависал.

После того как мы идентифицировали все звенья в этой цепочке отказов, следовало найти решение. Пул ресурсов умеет проверять допустимость JDBC-соединений перед тем, как захватить их. Это делается при помощи SQL-запроса, такого как `SELECT SYSDATE FROM DUAL`. Но это в любом случае приводит к зависанию потока, обрабатывающего запросы. Еще можно заставить пул отслеживать время ожидания JDBC-соединений и отбрасывать все с продолжительностью более одного часа. К сожалению, для этого нужно отправить на сервер базы данных пакет, сообщающий о конце сеанса. Снова зависание.

Мы начали обдумывать сложные варианты. Например, создать поток «жнец», который будет искать устаревающие соединения и закрывать их до истечения таймаута. К счастью, умный администратор базы данных напомнил одну вещь. В Oracle имеется так называемый механизм *выявления мертвых соединений* (dead connection detection). Его можно задействовать, чтобы понять, когда произошло падение клиента. После его включения сервер базы данных начинает через определенные интервалы посылать пакеты проверки связи. Ответ клиента дает базе

данных понять, что он все еще работает. Если же после отправки нескольких пакетов ответа не поступает, сервер базы данных предполагает, что клиент прекратил свою работу, и освобождает все реализующие соединение ресурсы.

В данном случае нас не интересовала работоспособность клиента, но каждый пакет проверки связи сбрасывал у фаервола время, прошедшее с момента отправки «последнего пакета», поддерживая соединение «живым». Механизм выявления мертвых соединений сохранял жизнь нашему соединению и давал мне возможность спать по ночам.

А сейчас мы поговорим о проблемах протоколов на базе HTTP, включая веб-службы.

HTTP-протоколы

Сервис-ориентированная архитектура (Service-Oriented Architecture, SOA) в наши дни остается актуальной, конечно, если слушать производителей серверов приложений. Одной из причин интереса к SOA является снова появившаяся надежда на повторное применение, которого не обеспечивают технологии RPC, OOP, CORBA и EJB. Также зачастую говорят о более эффективном использовании ресурсов центров обработки и хранения данных за счет предоставления наиболее востребованным службам общего аппаратного обеспечения. Другие организации хотят обещаемых технологий SOA гибкости и проворства.

Общей чертой сервис-ориентированной архитектуры, базирующейся на протоколах семейства WS-I, SOAP, XML-RPC или REST, является HTTP¹. Во всех случаях в конечном счете посылается небольшой фрагмент XML-кода в виде HTTP-запроса, на который ожидается HTTP-ответ.

Разумеется, все протоколы на основе HTTP используют сокет, а значит, уязвимы для описанных ранее проблем. HTTP порождает и дополнительные сложности, связанные преимущественно с клиентской библиотекой. Любой Java-разработчик имеет встроенный HTTP-клиент, доступный через классы `java.net.URL` и `java.net.URLConnection`.

```
Line 1 URL url = new URL("http://www.google.com/search?q=foo");
2 URLConnection conn = url.openConnection();
3 HttpURLConnection httpConnection = (HttpURLConnection)conn;
4 httpConnection.setRequestProperty("User-Agent",
5 "Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.8.0.1) "
6 + "Gecko/20060111 Firefox/1.5.0.1");
7 InputStream response = httpConnection.getInputStream();
```

¹ Технически протокол SOAP и протоколы семейства WS-I допускают другие варианты передачи сообщений, но на практике ими пользуются только поклонники TIBCO и IBM MQ.

В языке Java обобщенный класс `URL` пытается скрыть разницу между HTTP, HTTPS, FTP и другими протоколами. В строке 1 мы конструируем URL-запрос к сайту Google. Открывающая соединение строка 2 запрос не посылает; она действует как фабричный метод, создавая реальный подкласс класса `URLConnection`, выполняющий всю работу. Нужно выполнить понижающее приведение возвращенного значения `URLConnection` к этому конкретному классу, чтобы в строке 4¹ вызвать метод `setRequestProperty()`. Наконец, в строке 7 класс `HttpURLConnection` действительно откроет сокет для соединения с удаленным хостом, отправит HTTP-запрос, проанализирует полученный ответ и вернет класс `InputStream` в теле ответа.

В строке 7 происходит очень многое. Именно здесь класс `HttpURLConnection` выполняет подключение к удаленному серверу. Все, что было до этого, относится к настройке локальной памяти. До Java 1.5 возможности задавать параметры не было, поэтому приходилось прибегать к таким трюкам, как установка нестандартного интерфейса `SocketImplFactory`. К счастью, сейчас JDK позволяет контролировать таймаут соединения, но времени ожидания чтения до сих пор не существует. Удаленная система может в течение следующих десяти лет передавать по одному байту в секунду, и ваш поток будет обслуживать этот вызов. Скептически настроенная система никогда не станет мириться с таким незащищенным вызовом. К счастью, другие доступные HTTP-клиенты дают нам куда больший контроль. Например, пакет `HttpClient` из проекта Jakarta Common от Apache предлагает детальное управление временем ожидания как соединения, так и чтения, не говоря уже о заголовках запросов, заголовках ответов и политик в отношении файлов cookie.

API-библиотеки от производителей

Было бы приятно думать, что производители коммерческих программ *как следует* защищают свою продукцию от ошибок, ведь она продается множеству клиентов. Для продаваемого ими серверного программного обеспечения это, возможно, и так, но о клиентских библиотеках сказать подобное сложно. Обычно для API-библиотек, предоставляемых заказчику производителем, характерны многочисленные проблемы, и зачастую они являются скрытой угрозой стабильности. Эти библиотеки представляют собой всего лишь код, написанный обычными разработчиками. И их качество, стиль написания и степень защиты варьируются, как у любого другого образца кода.

Хуже всего в этих библиотеках то, что вы практически не можете ими управлять. Максимум, что можно сделать, — произвести декомпиляцию кода, найти проблемы и отправить отчет производителю. Если у вас хватит влияния надавить

¹ Нам приходится лгать о нашем пользовательском агенте, иначе Google вернет ошибку 403 «доступ запрещен».

на производителя, возможно, вы получите исправленную версию библиотеки, разумеется, при условии, что вы пользуетесь последней версией предлагаемого им программного обеспечения. Мне в прошлом приходилось самостоятельно разбираться с ошибками и собственноручно компилировать программу для временного использования в ожидании исправленной версии от производителя.

Основная угроза стабильности, которую несут API-библиотеки от производителя, касается блокировок. Будь то внутренний пул ресурсов, вызовы для чтения из сокета, HTTP-соединения или старая добрая Java-сериализация, все эти библиотеки используют небезопасные принципы написания кода.

Вот классический пример. При наличии программных потоков, которые нужно синхронизировать на нескольких ресурсах, всякий раз возникает опасность взаимной, или мертвой, блокировки. Поток 1 удерживает ресурс *A* и хочет захватить ресурс *B*, в то время как поток 2 удерживает ресурс *B* и хочет захватить ресурс *A*. Чтобы избежать подобной ситуации, рекомендуется всегда получать ресурсы в одном порядке и освобождать их в обратном порядке. Разумеется, это помогает только в случае, если вы *знаете*, что конкретный поток захватит оба ресурса, и можете контролировать порядок этого захвата. Рассмотрим пример на языке Java. Эта иллюстрация может принадлежать библиотеке промежуточного уровня, ориентированной на работу с сообщениями:

```
stability_anti_patterns/UserCallback.java
public interface UserCallback {
    public void messageReceived(Message msg);
}
stability_anti_patterns/Connection.java
public interface Connection {
    public void registerCallback(UserCallback callback);
    public void send(Message msg);
}
```

Уверен, что вы узнаете этот код. Является ли он безопасным? Понятия не имею. Не зная, зачем вызывается метод `messageReceived()` потока, вы не можете сказать, какие ресурсы он будет удерживать. В стеке уже может быть десяток синхронизированных методов. Перед вами минное поле взаимных блокировок.

Более того, несмотря на то что интерфейс `UserCallback` не объявляет метод `messageReceived()` как синхронизированный (вы не можете объявить синхронизированный интерфейсный метод), реализация может сделать его таковым. В зависимости от потоковой модели внутри клиентской библиотеки и времени выполнения метода обратного вызова синхронизация этого метода может привести к блокировке потоков. И подобно засорившейся трубе для слива воды, эти заблокированные потоки могут привести к блокировке потоков, вызывающих метод `send()`. Вероятнее всего, это означает выведение из строя потоков, отвечающих за обработку запросов. И как обычно бывает в подобных случаях, ваше приложение просто перестает работать.

Борьба с проблемами точек интеграции

Автономная система в наши дни является редкостью, кроме того, она почти бесполезна. Каким же образом можно повысить безопасность точек интеграции? Наиболее эффективны в деле борьбы с отказами в этих точках паттерны предохранителя (Circuit Breaker) и разделения связующего ПО (Decoupling Middleware).

Для борьбы с отказами в точках интеграции пользуйтесь паттернами предохранителя и разделения связующего ПО.

Еще помогает тестирование. Скептически построенное программное обеспечение должно уметь справляться с нарушениями форм и функций, такими как неверно отформатированные заголовки или внезапно оборванное соединение. Чтобы убедиться в том, что ваша программа построена достаточно скептически, требуется *тестовая программа* (test harness) — симулятор с контролируемым поведением — для проверки каждой из точек интеграции. Настроив эту программу на выдачу шаблонных ответов, вы облегчите функциональное тестирование. Заодно это обеспечит вам изоляцию от объекта управления. Наконец, каждая такая программа позволит вам имитировать различные виды системных и сетевых отказов.

Эта тестовая программа немедленно поможет вам с функциональным тестированием. Для проверки стабильности также нужно поменять состояние всех переключателей, когда система будет под значительной нагрузкой. Такую нагрузку может обеспечить набор рабочих станций, на которых запущен инструмент JMeter или Marathon, но для этого определенно потребуется куда больше, чем группа тестеров, щелкающих кнопками мыши.

ЗАПОМНИТЕ

Будьте осторожны с этим неизбежным злом

Любая точка интеграции в конечном счете приведет к отказу, и вы должны быть к этому готовы.

Подготовьтесь к разным формам отказов

Отказы в точках интеграции принимают разные формы, от сетевых до семантических ошибок. И сопровождать их будут вовсе не аккуратные отчеты об ошибках, предоставляемые вам в соответствии с определенным регламентом; вместо этого вы будете сталкиваться с нарушениями этого регламента, задержками сообщений или вообще с зависанием.

Готовьтесь перейти на более низкий уровень абстракции

Отладка отказов в точках интеграции обычно требует понижения уровня абстракции. Отказы высокоуровневых протоколов зачастую невозможно отладить на уровне приложения. Вам помогут перехватчики пакетов и другие средства сетевой диагностики.

Отказы быстро распространяются

При недостаточно защищенном коде отказ в удаленной системе быстро станет вашей проблемой, как правило, через каскадные отказы.

Для предотвращения проблем с точками интеграции пользуйтесь паттернами

Безопасное программирование на основе паттернов предохранителя (Circuit Breaker), таймаутов (Timeouts), разделения связующего ПО (Decoupling Middleware) и квитирования (Handshaking) поможет вам избежать опасностей точек интеграции.

4.2. Цепные реакции



В разделе 8.1 мы будем обсуждать две основные разновидности масштабирования: горизонтальное и вертикальное. В случае *горизонтального масштабирования* дополнительная производительность достигается путем увеличения количества серверов. Этот подход практикуют Google и Amazon. Пример такого подхода — веб-ферма. Каждый сервер при этом добавляет примерно ту же производительность, что и предыдущий. Альтернативное *вертикальное масштабирование* означает создание все более крупных серверов: замена одноядерных компьютеров x86 серверами с набором многоядерных процессоров. Это подход практикует Oracle. Каждый тип масштабирования подходит под свои условия.

Если в вашей системе применяется горизонтальное масштабирование, значит, это ферма со сбалансированной нагрузкой или кластеры, в которых на каждом сервере работает один и тот же набор приложений. Большое число машин обеспечивает устойчивость к сбоям за счет избыточности. Один процесс или машина могут выйти из строя, но остальные продолжают обслуживать транзакции.

Единой точкой отказа (Single Point Of Failure, SPOF) называют любое устройство, узел или кабель, выход которого из строя приводит к отказу всей системы. К примеру, в эту категорию попадают серверы только с одним источником электропитания и отсутствием резервных сетевых коммутаторов.

И хотя для горизонтальных кластеров наличие единых точек отказа не характерно (исключая случай самостоятельно спровоцированных отказов в обслуживании, которые рассматриваются в разделе 4.6), у них возможен режим отказа, связанный с нагрузкой. Если один из узлов в группе со сбалансированной нагрузкой выходит из строя, остальные узлы должны взять эту нагрузку на себя. К примеру, в ферме из восьми серверов, показанной на рис. 6, каждый узел обрабатывает 12,5 % общей нагрузки.

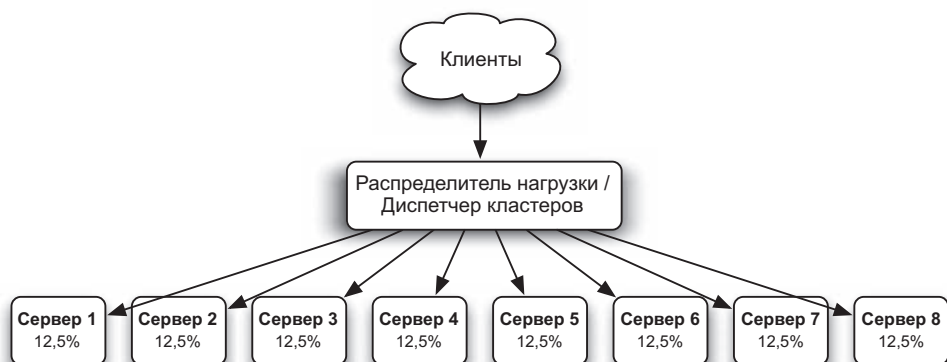


Рис. 6. Горизонтальная ферма из восьми серверов

Распределение нагрузки после выхода одного из серверов из строя показано на рис. 7. Теперь каждый из оставшихся семи серверов должен справиться с 14,3 % общей нагрузки. Каждый из серверов должен принять дополнительно всего 1,8 % общей нагрузки, после чего нагрузка на отдельные серверы составит более 15 %. В случае отказа в кластере, состоящем всего из двух узлов, нагрузка на выживший сервер удваивается. К исходной нагрузке (50 % от общей) добавится нагрузка с умершего узла (50 % от общей).

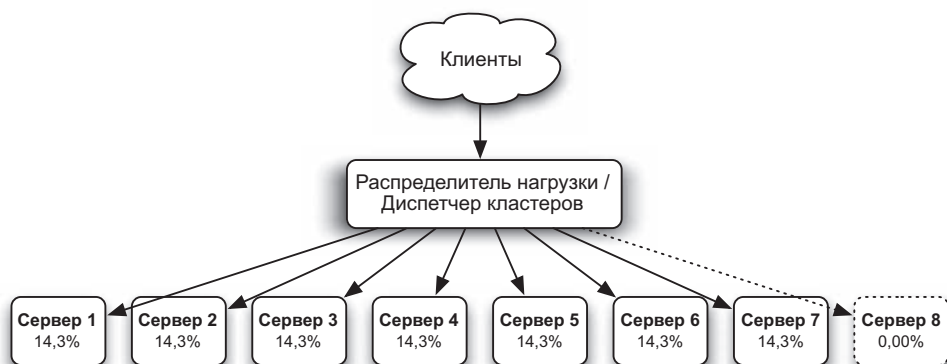


Рис. 7. Бывший кластер из восьми серверов

Когда первый сервер выходит из строя по причине, связанной с нагрузкой, например из-за утечек памяти или периодически возникающего состояния гонок, повышается вероятность выхода из строя остальных узлов. С прекращением работы каждого следующего сервера нагрузка на оставшиеся серверы все больше возрастает, а значит, растут шансы, что и они прекратят работу.

Цепные реакции возникают при наличии дефектов приложения — обычно таких, как утечка ресурсов или сбой, связанный с нагрузкой. В случае однородного слоя дефект будет присутствовать на каждом из серверов.

То есть единственным способом исключения цепной реакции является устранение исходного дефекта. Разбиение слоя на несколько пулов, как в паттерне переборок (Bulkheads), иногда превращает одну цепную реакцию в две, идущие с разными скоростями.

Какой эффект цепная реакция может оказать на остальную часть системы? Начнем с того, что цепная реакция отказов в одном слое может легко привести к каскадным отказам в вызывающем слое.

ПОИСК

Однажды мне пришлось работать с основным сетевым брендом одного из продавцов. Это был огромный каталог — полмиллиона единиц складского учета (Stock Keeping Unit, SKU) в ста различных категориях. В таких случаях поиск представляет собой не только полезную, но и обязательную функцию. Для обслуживания клиентов в праздничные дни продавец использовал десятки поисковых служб с аппаратным выравнивателем нагрузки. При этом серверы приложения были настроены на подключение к виртуальным IP-адресам¹, а не к конкретным поисковым службам. После этого аппаратный выравниватель нагрузки распределял запросы серверов приложений по этим службам. Одновременно он выполнял проверку работоспособности, чтобы понять, какие из серверов живы и отвечают. Это позволяло отправлять запросы только к функционирующим поисковым службам.

Оказалось, что эти проверки работоспособности имели смысл. В поисковой системе была ошибка, приводившая к утечкам памяти. При обычном трафике (не в праздничные дни) поисковые службы отключались сразу после полудня. А так как по утрам каждая из служб получала одинаковую порцию нагрузки, все они переставали работать примерно в одно время. С выходом из строя каждой следующей службы распределитель нагрузки отправлял ее часть запросов на оставшиеся серверы, еще быстрее вызывая нехватку памяти. При взгляде на график временных меток «последних ответов» я сразу отметил нарастание числа сбоев. Между первым и вторым сбоем прошло пять или шесть минут. Второй и третий сбой разделяли две или три минуты. Последние два сбоя произошли с интервалом в несколько секунд.

Причиной прекращения работы этой системы также стали каскадные отказы и блокировка потоков. Утрата последнего поискового сервера привела к остановке всего входного интерфейса.

До получения от производителя системы эффективного исправления (которое появилось только через несколько месяцев) нам приходилось ежедневно перезагружать серверы в пиковое время: 11, 16 и 21 час.

Иногда цепные реакции вызываются заблокированными потоками. Это происходит при блокировке всех потоков приложения, которые отвечают за обработку

¹ Более подробно про распределение нагрузки и виртуальные IP-адреса вы узнаете в разделе 11.3.

запросов, после чего приложение перестает отвечать. После этого входящие запросы распределяются среди приложений на других серверах того же слоя, увеличивая вероятность их отказа.

ЗАПОМНИТЕ

Выход из строя одного сервера ставит под угрозу остальные

Цепная реакция возникает потому, что смерть одного из серверов заставляет остальные серверы принять на себя его нагрузку. А это повышает вероятность сбоя. Цепная реакция может быстро вывести из строя целый слой. Остальные связанные с ним слои должны иметь защиту, в противном случае они также прекратят работу в результате каскадных отказов.

Ищите утечки ресурсов

В большинстве случаев цепные реакции возникают, когда приложение сталкивается с утечками памяти. Как только один из серверов по причине нехватки памяти выходит из строя, остальные берут на себя его работу. Увеличившийся трафик означает, что они начнут терять память быстрее.

Ищите неявные ошибки синхронизации

Неявное состояние гонок также может быть спровоцировано трафиком. И снова, если один из серверов прекращает свою работу из-за взаимной блокировки, выросшая нагрузка на остальные серверы повышает вероятность того, что и они выйдут из строя по аналогичной причине.

Используйте для защиты паттерн переборок

Разделение серверов при помощи паттерна переборок (Bulkheads) может предотвратить распространение цепной реакции на весь сервис, хотя и не способно помешать сбоям в вызывающем коде, обращающемся к любой из частей. Для борьбы с этим злом используйте паттерн предохранителя (Circuit Breaker) на вызывающей стороне.

4.3. Каскадные отказы



Стандартная архитектура корпоративных систем, в том числе веб-сайтов и веб-служб, состоит из набора функционально различных ферм или кластеров, связанных между собой посредством определенной формы балансировки нагрузки. Мы обычно называем отдельные фермы *слоями*, как, например, на рис. 8, хотя на самом деле они могут располагаться в разных стеках.

В сервис-ориентированной архитектуре эта картинка мало похожа на традиционные слои, а больше напоминает ориентированный ациклический граф.

Системные отказы начинаются с единственной трещины, причина которой гнездится в какой-то фундаментальной проблеме. Существуют механизмы, позволяющие замедлить или остановить распространение трещин, и мы будем рассматривать их в следующей главе. Без этих механизмов трещина может распространяться

и разрастаться из-за различных структурных проблем. Каскадный отказ возникает, когда трещина в одном из слоев вызывает проблемы в вызывающем слое.

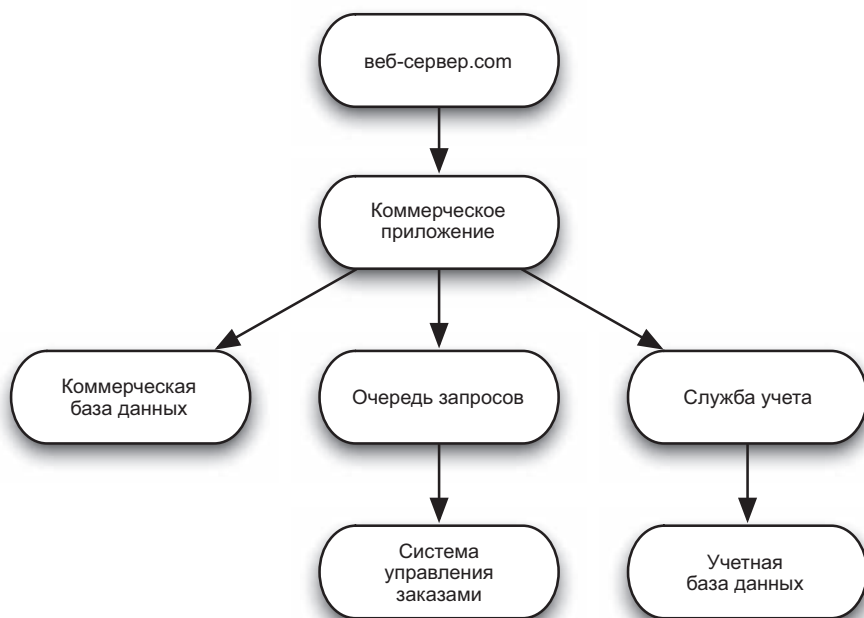


Рис. 8. Слои, часто встречающиеся в коммерческих системах

Каскадный отказ возникает, когда проблемы в одном из слоев провоцируют проблемы в вызывающих слоях.

Очевидным примером является отказ базы данных. При выходе из строя целого кластера базы данных любое обращающееся к ней приложение столкнется с проблемами. И если эти проблемы не обрабатываются должным образом, в слое приложения начинается сбой. Однажды я столкнулся с системой, которая обрывала любое JDBC-соединение, выбрасывая исключение `SQLException`. Каждый запрос страницы пытался создать новое соединение, получал исключение `SQLException`, пытался разорвать это соединение, получал еще одно исключение `SQLException`, после чего вываливал на пользователя отладочную информацию из стека.

Каскадные отказы возможны при наличии механизма перехода отказа с одного слоя на другой. Отказ «перепрыгивает со слоя на слой», когда неправильная работа в вызывающем слое провоцируется условием отказа в вызываемом. Часто каскадные отказы обуславливаются пулом ресурсов, который истощается по причине

отказа в нижележащем слое. Точки интеграции без механизмов контроля времени ожидания являются гарантированным средством порождения каскадных отказов.

И если точки интеграции лидируют в качестве источников трещин, каскадные отказы являются их основным ускорителем. Предотвращение каскадных отказов — это основной ключ к отказоустойчивости. А наиболее эффективными в борьбе с этим видом отказов являются паттерны предохранителя (Circuit Breaker) и таймаутов (Timeouts).

ЗАПОМНИТЕ

Не давайте трещинам пересекать границы слоев

Каскадные отказы возникают, когда трещина переходит из одной системы (или слоя) в другую, обычно из-за недостаточно защищенных точек интеграции. Также причиной каскадных отказов может стать цепная реакция в нижележащем слое. Ваша система, безусловно, будет обращаться к другим корпоративным системам; постарайтесь сделать так, чтобы она сохранила работоспособность в случае, если эти системы прекратят работать.

Внимательно изучайте пулы ресурсов

Источником каскадных отказов часто становятся пулы ресурсов, например пул соединений, который истощается, когда отправляемые им вызовы остаются без ответа. Получающие соединение потоки блокируются навсегда; все остальные потоки блокируются в ожидании соединений. Безопасный пул ресурсов всегда ограничивает для потоков время ожидания проверки ресурсов.

Пользуйтесь паттернами предохранителя и таймаутов

Каскадный отказ возникает **после** того, как что-то пошло не так. Паттерн предохранителя (Circuit Breaker) защищает систему, не давая обращаться к проблемным точкам интеграции. Применение паттерна таймаутов (Timeouts) гарантирует сохранение работоспособности системы, если в процессе обращения к другому слою или к другой системе возникают проблемы.

ПРИМЕЧАНИЕ

Механизм перехода между слоями часто принимает форму заблокированных потоков, но я встречал и обратную ситуацию — наличие чрезмерно агрессивного потока. Один раз вызывающий слой столкнулся с ошибкой, которую в силу отсутствия исторического прецедента расценил как невоспроизводимую временную ошибку нижележащего слоя. В какой-то момент в нижележащем слое возникало состояние гонок, которое вполне могло спровоцировать однократную ошибку без достаточных на то оснований. Когда это произошло, вышестоящий слой решил повторить вызов. К сожалению, нижележащий слой предоставил недостаточно информации, чтобы отличить однократную ошибку от более серьезной. В результате как только нижележащий слой столкнулся с реальными проблемами (из-за прекратившего работу коммутатора стал терять пакеты, идущие от базы данных), вызывающая сторона увеличила количество запросов. Чем хуже реагировал нижележащий слой, тем больше поводов для жалоб давал ему верхний слой, усиливая свой натиск. В конечном счете вызывающий слой задействовал 100 % ресурсов процессора на вызовы нижележащего слоя и регистрацию происходящих отказов в журнале. Паттерн предохранителя (Circuit Breaker) позволяет избежать подобной ситуации.

4.4. Пользователи



Пользователи — ужасные существа¹. Без них системы были бы намного более стабильными. Пользователи систем имеют особый талант к творческому разрушению. Когда система балансирует на грани катастрофы, как автомобиль на краю обрыва в боевике, некоторые пользователи напоминают садящихся на крышу чаек. И все падает вниз! Пользователи всегда делают самое худшее в самый неудобный момент.

Пользователи — ужасные существа.

Куда хуже, когда другие системы настойчиво обращаются к нашей, игнорируя то, что она находится на грани сбоя.

Трафик

Каждый пользователь потребляет ресурсы системы. Если ваша система не является одноранговой, такой как BitTorrent, ее производительность ограничена. Ее масштабирование зависит от количества имеющегося аппаратного обеспечения и полосы пропускания, а не от числа привлеченных пользователей. Постоянно растущий трафик в какой-то момент начинает превышать возможности системы². После этого возникает самый большой вопрос: как система среагирует на возросший спрос?

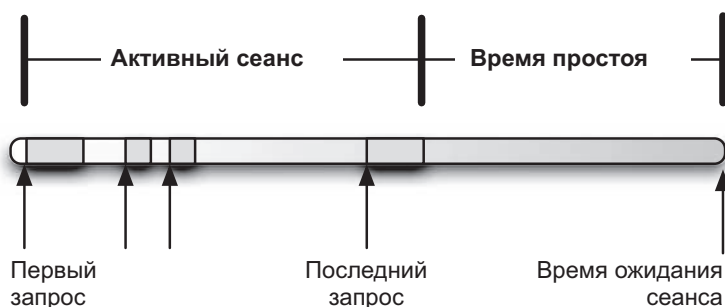
Каждый пользователь требует дополнительной памяти.

В разделе 8.1 вы найдете определение производительности: если выполнение транзакций занимает слишком много времени, значит, загрузка вашей системы превышает ее производительность. Но внутри вашей системы существуют более жесткие ограничения. И выход за их пределы приводит к появлению трещин, которые под нагрузкой начинают распространяться быстрее. Одно из таких ограничений — это количество доступной памяти, особенно в системах Java или J2EE. Превышение трафика создает нагрузку на память несколькими способами. Прежде всего, в веб-системах для каждого пользователя существует свой сеанс. И этот сеанс остается в резидентной памяти некоторое время после последнего

¹ Разумеется, я говорю в ироническом смысле. Конечно, пользователи создают определенную угрозу стабильности, но без них наших систем не существовало бы.

² Если трафик не растет, значит, у вас есть другие проблемы, о которых стоит беспокоиться.

пользовательского запроса. И каждый следующий пользователь требует дополнительной памяти.



Во время простоя сеанс все еще занимает ценную память. Сеанс — это не магический «бездонный мешок»¹. Каждый объект в сеансе связывает драгоценные байты, которые могли бы послужить какому-нибудь другому пользователю.

При нехватке памяти становятся возможными удивительные вещи. К наиболее безобидным из них можно отнести исключение `OutOfMemoryError`, с которым сталкивается пользователь. Если дело заходит слишком далеко, система протоколирования утрачивает возможность даже зарегистрировать ошибку. К примеру, библиотека `Log4j` и пакет `java.util.logging` создают объекты, представляющие событие протоколирования. Если памяти для этого не хватает, регистрации событий не происходит. (Кстати, это сильный аргумент за внешний мониторинг в дополнение к исследованию журнала регистрации.) Ситуация с нехваткой памяти, которую можно исправить, быстро превращается в серьезную проблему стабильности.

На самом деле, при внутренних вызовах нехватка памяти вызывает сбой функции выделения памяти во внутреннем коде, например внутри драйвера `Type 2 JDBC`. Создается впечатление, что мало кто из пишущих внутренний код программистов как следует выполняет проверку ошибок, потому что я видел, как результатом внутренних вызовов на фоне нехватки памяти становится падение JVM.

Лучше всего по возможности сделать количество сеансов минимальным. К примеру, не стоит хранить внутри сеанса весь набор результатов поиска для их разбиения на страницы. Лучше требовать каждую следующую страницу от поискового механизма. Учитывайте, что каждый оказавшийся в сеансе объект может больше никогда не потребоваться. И следующие тридцать минут он будет бесполезным грузом болтаться в памяти, подвергая риску вашу систему.

¹ Тем, кто не знаком с игрой «Подземелья и драконы», объясню, что «бездонный мешок» внутри больше, чем снаружи. Положенные в него вещи доступны, но при этом почти ничего не весят. Это объясняет, каким образом персонажи могут постоянно носить с собой пару палашей, кистень, полные доспехи и полмиллиона золотых.

Было бы здорово, если бы существовал способ сохранять объекты в сеансе (а значит, в памяти) при избытке памяти, но автоматически включать режим экономии, когда ее не хватает. Могу вас обрадовать! Такой способ есть. Объекты `java.lang.ref.SoftReference` сохраняют ссылки на другие объекты, несущие полезную нагрузку.

Конструируется объект `SoftReference` с большим или ценным объектом в качестве аргумента. Вот он-то и является нашим бездонным мешком, который хранит полезную информацию для дальнейшего использования.

```
MagicBean hugeExpensiveResult = ...; SoftReference ref = new
SoftReference(hugeExpensiveResult);

session.setAttribute(EXPENSIVE_BEAN_HOLDER, ref);
```

Это не очевидное изменение. Любые обращающиеся к этому объекту JSP-страницы или сервлеты будут знать, что они проходят через обходной слой. При нехватке памяти сборщик мусора получает разрешение затребовать содержимое объекта `SoftReference`, как только на него не будет ни одной жесткой ссылки.

```
Reference reference =
    (Reference)session.getAttribute(EXPENSIVE_BEAN_HOLDER);
MagicBean bean = (MagicBean) reference.get();
```

Где находится точка добавления этого обходного слоя? Как только возникает нехватка памяти, сборщик мусора получает возможность требовать любые «мягко достижимые» объекты. В эту категорию объект попадает, если единственная ссылка на него содержится в объектах `SoftReference`. Ценный объект на рис. 9 мягко достижим. А вот объект с рис. 10 в эту категорию *не* попадает. Он доступен непосредственно, так как на него есть ссылка из сервлета.

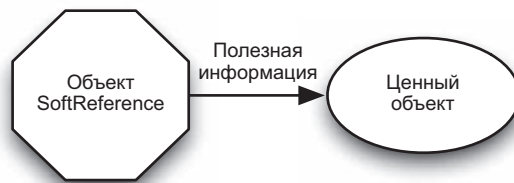


Рис. 9. Объект `SoftReference` и его полезные данные

Решение о том, когда затребовать мягко достижимые объекты, сколько их затребовать, а сколько оставить в резерве, принимает сборщик мусора. Можно гарантировать только то, что все мягко достижимые объекты будут затребованы до выброса исключения `OutOfMemoryError`.

Другими словами, сборщик мусора, прежде чем прекратить свою работу, использует всю помощь, которую вы ему оказываете. Имейте в виду, что сборке подвергается не сам объект `SoftReference`, а только содержащаяся в нем полезная информация. И после этого все вызовы метода `SoftReference.get()` вернут значение `null`, как показано на рис. 11. И любой код, который использует объект с ценной информацией, должен уметь обращаться с нулевым содержимым. Он может повторно вычислить ценный результат, перенаправить пользователя к другой транзакции или предпринять любую другую меру защиты.

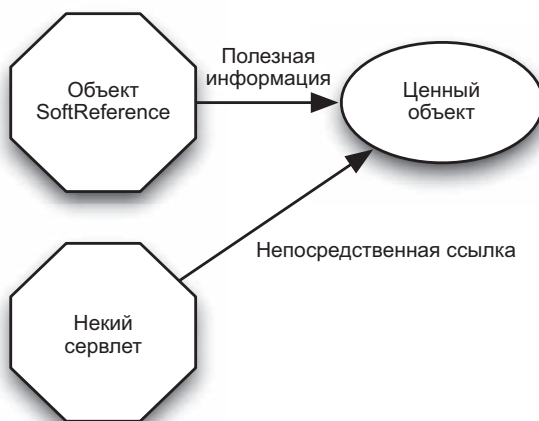


Рис. 10. Непосредственно доступный полезный объект



Рис. 11. Объект `SoftReference` после того, как полезная информация отправлена в мусор

Объект `SoftReference` позволяет реагировать на изменение состояния памяти, но добавляет сложности. В общем случае лучше всего просто держать объекты за пределами сеанса. Пользуйтесь этим объектом, только когда большие или ценные объекты держать за пределами сеанса невозможно. Объект `SoftReferences` позволяет обслуживать большее число пользователей без увеличения объема памяти.

Расходы на обслуживание

Некоторые пользователи значительно более требовательные, чем остальные. По иронии судьбы именно они представляют для вас наибольшую ценность. К примеру, в системе для розницы пользователи, которые просматривают пару-тройку страниц, возможно, используют поиск, а затем уходят, составляют основную массу посетителей и их проще всего обслуживать. Необходимую им информацию обычно можно кэшировать (тем не менее внимательно прочитайте раздел 10.2 с важными предостережениями). Обслуживание их страниц обычно не затрагивает внешних точек интеграции. Скорее всего, вам придется взглянуть на данные пользователя и проследить историю его посещений, но не более того.

Но есть и пользователи, которые твердо намерены что-то купить. Если у вас не стоит скрипт покупки за один клик, потребителю для заказа потребуется посетить четыре или пять страниц. Именно столько посещает типичный пользователь за сеанс. Помимо этого процедура может потребовать тех самых ненадежных точек интеграции: авторизации кредитной карты, расчета налога с продаж, стандартизации адреса, поиска необходимого товара и отгрузки. По сути, дополнительные покупатели угрожают стабильности не только входной системы, но и внутренней системы и следующим за ней в цепочке системам (см. раздел 4.8). Увеличение коэффициента обращаемости посетителей в покупателей положительно сказывается на статистике прибылей и убытков, но, без сомнения, тяжело отражается на системах.

Коэффициент обращаемости — это процент посетителей сайта, реально что-то купивших.

Эффективной защиты от пользователей, требующих большого количества ресурсов, не существует. Прямой угрозы стабильности они не несут, но создаваемая ими дополнительная нагрузка повышает вероятность возникновения сбоев по всей системе. Тем не менее бороться с ними нельзя, ведь обычно именно они приносят доход. Так что же делать?

ПОЛНАЯ ОБРАЩАЕМОСТЬ

Если мало — плохо, то много должно быть хорошо, не так ли? Другими словами, почему не провести тестирование для коэффициента обращаемости 100 %? В качестве теста стабильности — это не самая плохая идея. Но я бы не рекомендовал пользоваться его результатами при планировании вычислительной мощности для обычного производственного трафика. По определению такие транзакции требуют больше всего ресурсов. Следовательно, средняя нагрузка на систему будет меньше, чем показатели этого теста. Построение системы, предназначенной для обработки только самых ресурсоемких транзакций, потребует десятикратных затрат на аппаратное обеспечение.

Лучшее, что в этом случае можно сделать, — это прибегнуть к активному тестированию. Идентифицируйте наиболее затратные транзакции и удвойте или даже утройте их долю. Если в вашей торговой системе ожидается коэффициент обращаемости 2 % (стандартный показатель для розничной торговли), нагрузочные тесты стоит провести для показателей в 4, 6 или 10 %.

Нежелательные пользователи

Нам спалось бы намного крепче, если бы единственными, о ком приходится заботиться, были пользователи, сообщающие номера своих кредитных карт. В соответствии с поговоркой «В жизни порой случаются ужасные вещи» в этом мире существуют ужасные, злонамеренные пользователи.

Некоторые из них не знают о своей вредоносности. К примеру, я видел, как плохо настроенные прокси-серверы начинали снова и снова требовать последний URL-адрес пользователя. Я смог по файлам cookie идентифицировать сеанс пользователя и узнать, кому он принадлежал. Записи в журнале показали, что это был зарегистрированный пользователь. По какой-то причине через пятнадцать минут после его последнего запроса этот запрос снова начал появляться в журнале. Сначала он повторялся через каждые тридцать секунд. При этом темп его появления ускорялся. Через десять минут мы уже наблюдали от четырех до пяти запросов *в секунду*. Эти запросы обладали файлом cookie, идентифицирующим пользователя, но не его сеанс. То есть каждый из них открывал новый сеанс. Все это сильно напоминало DDoS-атаку, но происходило через прокси-сервер одной военно-морской базы.

DDoS (Distributed Denial-of-Service Attack — распределенная атака отказа в обслуживании): множество компьютеров набрасывается на сайт с целью переполнения полосы пропускания, захвата ресурсов процессора или памяти серверов сайта. Представьте себе Гулливера, атакованного толпой лилипутов.

Мы снова отметили, что эти сеансы являются ахиллесовой пятой веб-приложений. Хотите вывести из строя практически любое динамическое веб-приложение? Выберите на сайте внешнюю ссылку и начните ее запрашивать, не отправляя файлы cookie. Ответа можно не ждать; бросайте подключение через сокет сразу же после отправки запроса. Веб-серверы никогда не сообщат серверам приложений, что конечный пользователь прекратил ждать ответа. Сервер приложений просто продолжит обработку запроса. А отправленный им ответ веб-сервер просто складывает в «битоприемник». При этом 100 байт HTTP-запроса заставляют сервер приложений открыть сеанс (для чего может потребоваться несколько килобайтов памяти). Каждый настольный компьютер на широкополосном соединении может сгенерировать сотни тысяч сеансов на сервере приложений.

В экстремальных случаях, таких как поток сеансов с военной базы, можно столкнуться с более серьезными проблемами, чем нагрузка на память. В нашем случае заказчики хотели знать, насколько часто возвращаются их постоянные клиенты. Разработчики написали небольшой перехватчик, который обновлял время «последнего захода в систему», когда профиль пользователя загружался в память из базы данных. Но во время этого потока сеансов запросы содержали файл cookie с идентификатором пользователя, но без сеанса. Поэтому каждый запрос трактовался как новый вход в систему, требующий загрузки профиля из базы данных и обновления времени «последнего захода».

Представьте 100 000 транзакций, каждая из которых пытается обновлять одну и ту же строку в одной и той же таблице базы данных. Какая-то из них должна попасть в состояние взаимной блокировки. Как только хотя бы одна транзакция, связанная с профилем пользователя, зависнет (из-за необходимости подключений из различных пулов ресурсов), блокируются и все остальные транзакции, связанные с этой строкой базы данных. Довольно скоро эти фиктивные входы в систему выбирают все потоки, обрабатывающие запросы. Сразу же после этого сайт прекращает свою работу.

Итак, вредоносные пользователи первого вида просто занимаются своими делами, оставляя после себя катастрофические последствия. Но есть и действительно злонамеренные люди, которые специально производят нестандартные действия, имеющие нежелательные последствия. Представители первой группы вредят непреднамеренно; причиняемый ими ущерб случаен. А вот представители второй группы составляют отдельную категорию.

Существует целая паразитическая индустрия, функционирующая за счет ресурсов с сайтов других компаний. Это так называемые компании *конкурентной разведки* (competitive intelligence), которые высасывают данные из вашей системы.

Эти компании утверждают, что их деятельность ничем не отличается от отправки в конкурирующий универсальный магазин человека со списком товаров и блокнотом для записи цен. На самом деле это не так. Учитывая скорость, с которой они запрашивают страницы, процесс больше напоминает отправку в соседний магазин целого батальона с блокнотами. И когда эта толпа заполонит все проходы, обычные покупатели просто не смогут войти внутрь.

Еще хуже то, что эти скоростные сборщики информации не используют файлы cookie для сеансов, и если вы не прибегаете к перезаписи URL-адресов для отслеживания сеансов, каждый новый запрос к странице будет открывать новый сеанс. Как и в случае флэшмоба, проблема производительности быстро превратится в проблему стабильности. Батальон, занимающийся списыванием чужих цен, может реально парализовать работу магазина.

Ограничить легальным роботам доступ к содержимому сайта достаточно легко при помощи файла `robots.txt`¹. Но следует помнить, что этот файл представляет собой

¹ См. <http://www.w3.org/TR/html4/appendix/notes.html#h-B.4.1.1>.

всего лишь запрос вашего сайта к входящему роботу. Робот должен обратиться к файлу и добровольно учесть ваши пожелания. Это общественный договор, а не стандарт и не условие, подлежащее исполнению. Некоторые сайты предпочитают переадресовывать роботов и пауков в соответствии с заголовком User-Agent. При этом в лучшем случае они попадают на статическую копию сайта или же сайт генерирует страницы без цен. Идея состоит в том, что сайт остается доступным для крупных поисковых служб, не показывая цен. Это дает возможность персонализировать цены, запускать пробные предложения, выделять отдельные страны или некую аудиторию для проведения испытаний в условиях рынка и т. п. В худшем случае робот отправляется на пустую страницу.

ОТСЛЕЖИВАНИЕ СЕАНСОВ

HTTP — крайне странный протокол. Если бы перед вами была поставлена задача создать протокол, продвигающий искусства, науки, коммерцию, свободу слова, тексты, изображения, звуки и видео, способный сшить громаду человеческих знаний и творчества в одну сеть, в результате вряд ли бы появился HTTP. Начнем с того, что он не хранит состояния. С точки зрения сервера каждый инициатор запроса появляется из тумана и дает какую-то команду, например «GET /site/index.jsp». Получив ответ, он исчезает в тумане, не утруждая себя изъяснениями благодарности. Если один из этих грубых, требовательных клиентов появится снова, сервер не поймет, что видел его раньше.

Некоторые умные люди из фирмы Netscape нашли способ вставить в этот протокол дополнительный бит данных. Изначально в Netscape задумали эти данные, называемые cookies, как средство передачи состояния от клиента к серверу и обратно. Файлы cookie представляют собой ловкий трюк. Они работали со всеми видами новых приложений, например с персональными порталами (в те времена это было очень круто) и торговыми сайтами. Однако бдительные разработчики приложений быстро сообразили, что нешифрованные данные файлов cookie могут управляться враждебными клиентами. В конце концов, тот факт, что некоторые браузеры отправляют строку User-Agent, в которой написано «Mozilla», не означает, что за этим действительно стоит браузер Mozilla. (Например, с версии 7 beta 1 браузер Internet Explorer все еще претендует на это имя и, вероятно, всегда будет претендовать. Его строка User-Agent гласит «Mozilla/4.0 [compatible; MSIE 7.0b; Windows NT 6.0]».) Поэтому система безопасности требует, чтобы файлы cookie либо не содержали реальных данных, либо шифровались. Одновременно крупные сайты обнаружили, что передача в файлах cookie реального состояния требует большой полосы пропускания и изрядного количества процессорного времени. Шифрование этих файлов было правильным шагом.

В итоге эти файлы стали использоваться для меньших фрагментов данных, объем которых позволял только пометить пользователя постоянным или временным образом для идентификации сеанса.

Сеанс представляет собой абстракцию, упрощающую построение приложений. На самом деле пользователь отправляет всего лишь набор HTTP-запросов. Веб-сервер их получает и после серии манипуляций отправляет HTTP-ответ. Не существует запроса «начать сеанс», по которому браузер понимает, что пришло время отправки запросов, как нет и запроса «завершить сеанс». (В любом случае браузер такому запросу доверять не будет.)

Сеансы связаны с кэшированием данных в памяти. Ранние CGI-приложения не нуждались в сеансах, так как для каждого нового запроса они начинали новый процесс (обычно это был сценарий на языке Perl). Это прекрасно работало. Не было ничего более безопасного, чем последовательность директив «fork, run и die». Но для охвата больших объемов разработчики и производители перешли к долгоиграющим серверам приложений, например серверам Java-приложений, и к долгоиграющим процессам Perl, воспользовавшись для этого модулем `mod_perl`. И вместо ожидания нового процесса при каждом запросе сервер теперь находится в непрерывном ожидании запросов. В этом случае можно кэшировать состояние от одного запроса к другому, уменьшая количество обращений к базе данных. Но требуется способ идентифицировать запрос как часть сеанса. С этой задачей прекрасно справляются файлы `cookie`.

Серверы приложений самостоятельно реализуют механизмы работы с файлами `cookie`, предоставляя вам симпатичный программный интерфейс, напоминающий карту или словарь. Но, как обычно бывает при скрытой обработке алгоритмов, при неверном использовании система может выйти из строя.

Когда эти скрытые алгоритмы включают в себя слой обходных механизмов, призванных сделать так, чтобы HTTP-протокол выглядел как настоящий протокол приложения, все может кончиться плохо. К примеру, не все сервисы сравнения цен корректно работают с файлами `cookie` сеансов. Каждый запрос приводит к открытию нового сеанса и напрасной трате памяти. Если веб-сервер настроен запрашивать у сервера приложений все URL-адреса, а не только входящие в контекст подключения, сеансы могут создаваться запросами с несуществующих страниц. Как вы увидите в разделе, посвященном убийцам производительности, полный контроль над сеансами жизненно важен для масштабируемости.

ЛОВУШКА ДЛЯ ПАУКА

Году в 1998, во времена расцвета поисковой системы AltaVista, я столкнулся с сайтом, который прекратил свою работу из-за слишком «умного» разработчика. Этот разработчик построил так называемую ловушку для паука, которая представляла собой страницу с набором случайным образом сгенерированных ссылок. Текстом для ссылок служили правдоподобно звучащие фразы, создаваемые генератором на основе цепи Маркова. URL-адрес содержал большой случайный хэш, но фактически вел на одну и ту же страницу. В итоге индексатор видел страницы с разным содержанием, ведущие к новым ссылкам. Создатель этой ловушки думал, что она не даст пауку увидеть остальную часть сайта и в поисковых службах не появятся ссылки на его содержимое. Чтобы паук гарантированно не мог вырваться из ловушки, на каждую страницу было помещено пять или шесть таких случайных ссылок.

Ловушка сработала великолепно. Как только система AltaVista нашла этот сайт, паук принялся запрашивать по случайным ссылкам страницу за страницей. При этом запустилось множество потоков, так как мы видели, что количество запросов растет в геометрической прогрессии. Вот в этом-то и была проблема. Вам следует запомнить этот абсолютный закон сети. У поисковых систем полоса пропускания всегда будет шире, чем у вас. В итоге индексатор использовал всю полосу пропускания компании. Сначала передача данных шла на гарантированной скорости, затем были использованы допустимые возможности ее увеличения, а потом был достигнут лимит трафика. К моменту, когда мы поняли, почему пользователи жалуются на медленную работу

сайта, ловушка для паука обошлась нам более чем в 10 000 долларов оплаты за трафик. Ловушка для паука напоминала машину Руба Голдберга, настроенную нажимать спуск курка у ружья. По сути, это был продуманный способ устроить самим себе DDoS-атаку (см. раздел 4.6).

Думаю, этот разработчик в итоге продал свой генератор текстов на основе цепи Маркова в качестве машины написания бизнес-планов.

Словом, роботы, которые с уважением относятся к файлу `robots.txt`, как правило, способны генерировать вам трафик (и доход), вредоносные же роботы полностью его игнорируют.

Я видел всего два работающих подхода.

Первый подход технический. Как только вы обнаружите программу-анализатор, заблокируйте ее в вашей сети. Если вы пользуетесь сетью доставки контента, например Akamai, в ней эта возможность является встроенной. В противном случае можно воспользоваться внешними фаерволами. Порой запросы от вредоносных роботов приходят с легальных IP-адресов с реальными записями в обратной зоне DNS. В такой ситуации вам поможет интернет-регистратор ARIN (<http://www.arin.net>). Такие роботы легко блокируются.

Но есть и те, кто маскирует адрес отправителя или отправляет запросы с десятка различных адресов. Некоторые из них доходят до смены строки User-Agent при каждом следующем запросе. Когда с одного IP-адреса в течение пяти минут приходят запросы из браузеров Internet Explorer в Windows, Opera в Mac и Firefox в Linux, тут явно что-то не так. Конечно, это может быть прокси-сервер на уровне провайдера или запущенная кем-то куча виртуальных эмуляторов. Но если эти запросы касаются поиска по целой категории продуктов, скорее всего, перед вами программа-анализатор. В конце концов, может потребоваться блокировка изрядного количества подсетей. Для сохранения производительности фаерволов имеет смысл периодически удалять старые блокировки. Это вариант паттерна предохранителя (Circuit Breaker).

Второй подход юридический. Напишите для своего сайта «условия использования», в которых будет говориться, что пользователи могут просматривать содержимое только в личных и некоммерческих целях. Обнаружив программы-анализаторы, обратитесь к юристам. Но этот метод эффективен только при сильной юридической поддержке. Ни одно из этих решений не дает перманентного результата. Смотрите на это как на борьбу с вредителями — стоит остановиться, и паразиты появляются снова.

Пользователи-злоумышленники

Последняя группа нежелательных пользователей является по-настоящему вредоносной. Эти гадкие люди *жаждут* убить ваше детище. Ничто не возбуждает

их больше, чем разрушение вещей, созданных чужими кровью, потом и слезами. Лично мне кажется, что в детстве кто-то разрушал замки из песка, которые они строили. И глубоко укоренившееся горькое чувство заставляет их поступать с другими так же.

По-настоящему талантливые взломщики, способные проанализировать защиту вашего сайта, разработать нестандартную атаку и незаметно просочиться в систему, слава богу, встречаются крайне редко. По различным причинам вы можете стать мишенью такого взломщика, но шансы невелики. Для этого нужно целенаправленно возбудить его гнев.

А подавляющее большинство пользователей-злоумышленников — *взломщики-дилетанты*.

Взломщик-дилетант (script kiddie): человек, который не создает собственных инструментов для атаки, а загружает и использует инструментарий, созданный «настоящими» хакерами.

Однако уничижительное название не должно вводить вас в заблуждение. Дилетанты опасны своей многочисленностью. Шансы, что ваши системы привлекут внимание настоящего хакера, крайне низки, а вот разнообразные дилетанты, скорее всего, атакуют их прямо сейчас.

Эта книга не об информационной безопасности и не о приемах нападения и защиты в сети. Рассмотрение надежных вариантов защиты и обороны от атак выходит за рамки темы данного издания. Наше обсуждение ограничивается гранью безопасности и стабильности, так как это относится к архитектуре систем и программного обеспечения.

Основную угрозу стабильности несет ставшая к настоящему моменту классикой DDoS-атака (распределенная атака отказа в обслуживании). Атакующий заставляет многочисленные широко распределенные по сети компьютеры генерировать нагрузку для вашего сайта¹. Иногда эта нагрузка принимает форму прямых TCP-соединений, без участия протоколов уровня приложения. Другие атакующие пытаются вывести из строя ваши сетевые устройства, различными способами вмешиваясь в протокол TCP/IP. От подобных атак защищает хорошо настроенное современное сетевое оборудование.

¹ Как правило, эта нагрузка исходит из *ботнета* (сети зараженных компьютеров). Демон на зараженном компьютере подписывается на IRC-канал, через который организатор ботнета подает команды. Встречались хакеры-дилетанты, управляющие ботнетами из десятков тысяч узлов, и ходят слухи о сетях из миллионов зараженных компьютеров. По большей части эти узлы представляют собой персональные компьютеры с устаревшей версией операционной системы Windows.

ЗАПОМНИТЕ**Пользователи потребляют память**

Каждый пользовательский сеанс требует некоторого объема памяти. Минимизируйте этот объем, чтобы увеличить производительность. Используйте сеанс только для кэширования, чтобы в случае нехватки памяти можно было удалить оттуда контент сеанса.

Пользователи делают странные вещи

Реальные пользователи делают вещи, которые невозможно предсказать (а иногда и понять). Если в приложении есть уязвимая точка, кем-то из многочисленных пользователей она будет обнаружена. Тестовые сценарии подходят для функционального тестирования, но слишком предсказуемы при проверке стабильности. Для более реалистичного тестирования лучше взять группу шимпанзе и заставить бить по клавиатуре.

Существуют злонамеренные пользователи

Как следует изучите проект вашей сети; это поможет в предотвращении атак. Убедитесь, что ваши системы легко допускают обновления и исправления, — вам часто придется к ним прибегать. Используйте новейшие среды разработки и занимайтесь собственным образованием. После 2007 года уже нельзя оправдать пропущенную успешную атаку SQL-инъекций.

Пользователей бывает слишком много

Иногда пользователи появляются огромными толпами. Представьте редактора новостного сайта Slashdot, который, хихикая, показывает пальцем на ваш сайт и кричит: «Всем туда!» Толпы пользователей могут вызвать зависания, взаимные блокировки и неявные состояния гонок. Проводите специальные нагрузочные испытания для проверки внешних ссылок и популярных URL-адресов.

Более новый вектор атак направлен уже против приложений, а не против сетевого оборудования. Такие атаки переполняют вашу исходящую полосу пропускания, мешают обслуживанию авторизованных пользователей и приводят к огромным счетам за трафик. Как мы уже видели, управление сеансами является самой уязвимой точкой веб-приложений на базе J2EE или Rails. Серверы приложений особенно восприимчивы к DDoS-атакам, поэтому перенасыщение полосы пропускания может оказаться не самой опасной из вставших перед вами проблем. Ограничить разрушения, причиняемые любым конкретным хостом, позволяет специализированный паттерн предохранителя (Circuit Breaker). Одновременно он защищает от случайных переполнений трафика.

У Cisco, Juniper, CheckPoint и других производителей сетевого оборудования существуют продукты, распознающие и ослабляющие DDoS-атаки. Но важна правильная конфигурация и мониторинг такого оборудования. К примеру, многие администраторы ограничивают количество подключений с одного IP-адреса пятнадцатью в минуту (основываясь на примере из сопроводительной документации Cisco). Если взять этот параметр за норму, любое AJAX-приложение становится

источником DDoS-атаки. (Так как мне случилось встречать некорректно работающие приложения, созданные восторженными AJAX-разработчиками, считаю, что это утверждение недалеко от истины.)

4.5. Блокированные программные потоки



Интерпретируемые языки, такие как Java и Ruby, практически никогда не вызывают по-настоящему аварийных ситуаций. Разумеется, ошибки приложений возможны, но ситуация сбоя интерпретатора или виртуальной машины возникает относительно редко. Я до сих пор помню, как нестандартный указатель на языке C превратил компьютер в кучу заикленного на себе железа. (Кто-нибудь еще помнит ошибки «Гуру медитирует» в операционной системе Amiga?) Но интерпретируемые языки тоже могут попасть в заколдованный круг. Интерпретатор может работать, а приложение при этом оказывается полностью заблокированным.

Как это часто бывает, усложнение с целью решения одной проблемы создает риск введения совершенно новых режимов отказов. Многопоточность обеспечивает серверам приложений достаточную для работы с крупнейшими сайтами масштабируемость, но при этом возникает вероятность ошибок, связанных с параллелизмом. Наиболее распространенным режимом отказа приложений, написанных на этих языках, является «уход в себя» — при благополучно работающем интерпретаторе все потоки замирают в ожидании Годо¹. Многопоточность — достаточно сложное явление, которому посвящены целые книги². Отход от модели «fork, run и die» дает намного большую производительность, но одновременно с этим возникают новые угрозы стабильности.

Большинство системных отказов, с которыми мне пришлось иметь дело, не сопровождалось явным отказом в обслуживании. Процесс шел, но при этом ничего не происходило, так как все потоки, которые могли бы заняться обработкой транзакций, блокировались в ожидании некоего невероятного события.

Заблокированные потоки могут появиться при любой выгрузке ресурсов из пула подключений, при работе с кэшем или реестром объектов, при обращении к внешним системам. Если код структурирован корректно, блокировка потоков временами возникает при попытке двух (и более) потоков одновременно обратиться к одному и тому же важному разделу. Это нормально. При условии, что код написан человеком, хорошо разбирающимся в многопоточном программировании, можно гарантировать, что эти потоки в конечном счете выйдут из состояния блокировки

¹ Герои пьесы Сэмюэля Беккета словно завязли во времени, ожидая некоего Годо, встреча с которым, по их мнению, внесет смысл в их бессмысленное существование.

² Единственной книгой, которая на самом деле требуется Java-программистам, является книга Дага Ли *Concurrent Programming in Java*.

и продолжают свою работу. Если у вас возникает именно такая ситуация, то вы относитесь к этому хорошо информированному меньшинству.

Проблема состоит из четырех частей:

- ❑ Ситуации сбоя и исключения порождают слишком много вариантов выполнения для исчерпывающего тестирования.
- ❑ Неожиданные взаимодействия могут привести к проблемам с ранее безопасным кодом.
- ❑ Большое значение имеет согласование по времени. Вероятность зависания приложения возрастает с увеличением числа параллельных запросов.
- ❑ Разработчики никогда не проверяют, как их приложение ведет себя при одновременном получении 10 000 запросов.

Все это означает, что в процессе разработки крайне сложно обнаружить, в каких ситуациях приложение виснет. На «тестирование вне системы» полагаться нельзя. Лучше всего с самого начала крайне аккуратно подойти к написанию кода. Пользуйтесь небольшим набором примитивов из известных паттернов. Имеет смысл загрузить качественную проверенную библиотеку¹.

Кстати, это еще одна причина, по которой я выступаю против самостоятельного написания классов пула подключений. Написать надежный, безопасный и высокопроизводительный пул подключений намного сложнее, чем кажется. Если вы когда-нибудь пытались писать модульные тесты, доказывающие безопасность параллелизма, вы знаете, как сложно получить надежные результаты в случае пула. Попытка выйти на нужные показатели превращает запуск собственного пула подключений из забавного упражнения в рутинную и монотонную обязанность.

С подозрением относитесь к синхронизированным методам в объектах домена.

Если вам приходится синхронизировать методы на доменных объектах, конструкцию системы, вероятно, стоит пересмотреть. Найдите способ сделать так, чтобы каждый поток получал собственную копию такого объекта. Это важно по двум причинам. Во-первых, если вы синхронизируете методы, чтобы гарантировать целостность данных, приложение сможет работать только на одном сервере. Однозначность в памяти перестает иметь значение при наличии второго сервера, вносящего изменения в данные. Во-вторых, ваше приложение будет лучше масштабироваться, если потоки, обрабатывающие запросы, перестанут блокировать друг друга.

¹ Если вы пишете на языке Java 5, но не пользуетесь примитивами из пакета `java.util.concurrent`, вам должно быть стыдно. Остальные могут загрузить библиотеку `util.concurrent` с адреса <http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html>. Это вариант библиотеки до ее принятия JCP.

СИСТЕМА НЕ СЛОМАЛАСЬ! ОНА ПРОСТО В ДЕПРЕССИИ!

Мне, наверное, уже сто раз приходилось объяснять разницу между крахом и зависанием системы. В конце концов я прекратил это делать, осознав, что разницу может заметить разве что инженер. Это все равно что физику пытаться объяснить, в какую из двух щелей во время эксперимента по квантовой механике проходит фотон. Имеет значение только один наблюдаемый параметр: может система проводить транзакции или нет. А в устах бизнесмена этот вопрос прозвучит так: «Приносит ли система доход?»

С точки зрения пользователя, совершенно неважно, как выглядит внутри неработающая система. Тот факт, что серверный процесс запущен, не поможет пользователю закончить свою работу, сделать покупку, заказать билет на нужный рейс и т. п.

Вот почему я выступаю за добавление внешнего мониторинга к внутреннему (изучению содержимого журналов, мониторингу процессов и портов). Суррогатный клиент в каком-то месте (не в том же самом центре обработки и хранения данных) на регулярной основе может запускать искусственные транзакции. Этот клиент будет видеть систему глазами обычного пользователя. Невозможность проведения искусственной транзакции означает наличие проблемы вне зависимости от работы серверных процессов.

Поиск блокировки

Можете найти блокирующий вызов в следующем фрагменте кода?

```
String key = (String)request.getParameter(PARAM_ITEM_SKU);
Availability avl = globalObjectCache.get(key);
```

Можно заподозрить, что синхронизация происходит в классе `globalObjectCache`. Вы будете правы, но ведь в вызывающем коде ничто не указывает на то, что один из вызовов блокируется, а другой нет. По сути, интерфейс, реализуемый классом `globalObjectCache`, ничего не сообщает о синхронизации.

В языке Java подкласс может объявить метод синхронизированным, но в определении его суперкласса или интерфейса он синхронизированным не будет. Пуристы от объектно-ориентированного программирования скажут, что это нарушает принцип подстановки Барбары Лисков. И будут совершенно правы.

СИНХРОНИЗАЦИЯ И ПРИНЦИП ПОДСТАНОВКИ БАРБАРЫ ЛИСКОВ

В теории объектно-ориентированного программирования существует принцип замещения Барбары Лисков, гласящий, что если объекты типа `T` обладают неким свойством, то оно должно наблюдаться и у объектов всех подтипов `T`. Другими словами, метод без побочных эффектов в базовом классе не должен иметь побочных эффектов и в производных классах. Метод, приводящий к исключению `E` в базовых классах, должен генерировать только исключения типа `E` (или подтипов `E`) в производных классах.

В языке Java запрещены объявления, нарушающие этот принцип. Непонятно, является ли возможность добавления синхронизации в подкласс намеренным ослаблением принципа Лисков или же это просто недосмотр.

В явном виде заменить экземпляр суперкласса синхронизированным подклассом невозможно. Как бы ни казалось, что я придираюсь, это может оказаться жизненно важным.

Базовая реализация интерфейса `GlobalObjectCache` представляет собой относительно простой реестр объекта:

```
public synchronized Object get(String id) {
    Object obj = items.get(id);
    if(obj == null) {
        obj = create(id);
        items.put(id, obj);
    }

    return obj;
}
```

Ключевое слово `synchronized` в методе сразу должно вас насторожить. Ведь пока этот метод выполняется одним потоком, все остальные вызывающие его потоки будут заблокированы. В данном случае синхронизация метода оправдана¹. Он выполняется быстро, и даже при наличии конкурирующих потоков они будут обслуживаться с приемлемой скоростью.

Но должен предостеречь, что класс `GlobalObjectCache` легко может стать ограничителем вычислительной мощности, если каждая транзакция начнет активно его использовать. В разделе 9.1 вы найдете пример того, каким образом заблокированные запросы влияют на производительность.

Часть системы должна была проверять наличие товаров на складе, делая ресурсоемкие запросы к удаленной системе. Исполнение этих внешних запросов занимало несколько секунд. Результаты были актуальны в течение как минимум 15 минут — так работает система инвентаризации.

Так как практически 25 % запросов касались самых ходовых товаров недели, а количество одновременных запросов к небольшой, работающей на пределе системе порой доходило до 4000 (в худшем случае), разработчик решил кэшировать итоговый объект `Availability`.

С точки зрения разработчика, правильной схемой был просмотр кэша. В случае попадания возвращался кэшированный объект. В случае промаха запрос выполнялся, его результат кэшировался, а затем возвращался клиенту. Следуя принципам правильного объектно-ориентированного программирования, разработчик создал

¹ Некоторые Java-программисты могли слышать выражение «блокировка с двойной проверкой». Этот паттерн позволяет избежать синхронизации всего метода. К сожалению, он не работает. По адресу <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html> вы найдете подробный отчет, почему он не работает и почему провалились все попытки его исправления.

расширение класса `GlobalObjectCache`, переопределяя метод `get()` для совершенного удаленных вызовов. Проектное решение как из учебника. В качестве нового класса `RemoteAvailabilityCache` выступал кэширующий прокси, как описано на с. 111–112 книги *Pattern Languages of Program Design 2*. На кэшированных записях имела даже временная метка, делавшая недействительными записи со слишком старой датой. Проект был элегантным, но этого было недостаточно.

Проблема состояла в том, что поведение этого проекта не было функциональным. Нет, объект `RemoteAvailabilityCache` был прекрасно сконструирован. Но под нагрузкой возникал отвратительный отказ. Система инвентаризации имела недостаточный размер (см. раздел 4.8), поэтому как только нагружалась входная часть системы, ее внутренняя часть переполнялась запросами. В конечном счете она прекращала свою работу. В этот момент любой поток, вызывающий метод `RemoteAvailabilityCache.get()`, блокировался, потому что всего один поток оказывался внутри вызова `create()`, ожидая ответа, который никогда не придет. В итоге все сидели, как Эстрагон и Владимир, в бесконечном ожидании Годо.

Этот пример демонстрирует разрушительное взаимодействие антипаттернов, ускоряющее рост трещин. Предпосылки для сбоев возникают за счет блокирующихся потоков и несбалансированных мощностей. Отсутствие таймаутов в точках интеграции позволяет сбою выйти за пределы одного слоя и стать каскадным. В конечном счете комбинация этих факторов привела к прекращению работы сайта. Заказчик сайта рассмеялся бы, если бы его спросили: «Должен ли сайт падать при отсутствии возможности проверить наличие выбранного товара на складе?» Архитекторы и разработчики в ответ на вопрос: «Упадет ли сайт, если не сможет проверить доступность товара?» — начали бы уверять, что такого просто не может быть. Даже разработчик класса `RemoteAvailabilityCache` не ожидал, что сайт повиснет, как только система инвентаризации прекратит отвечать. Никто не закладывал в проект эту возможность отказа, но никто не позаботился и о том, чтобы ее *исключить*.

Никто не закладывал в проект эту возможность отказа, но никто не позаботился и о том, чтобы ее исключить.

Библиотеки сторонних производителей

Библиотеки сторонних производителей печально известны как источники блокировки потоков. По иронии судьбы клиентские библиотеки для программного обеспечения корпоративного класса часто имеют внутри собственный пул ресурсов. И зачастую при возникновении проблем потоки, выполняющие запросы, просто навсегда блокируются. Разумеется, никто не даст вам возможность настроить там

режимы отказов, указав, к примеру, что делать, если все подключения окажутся связанными в ожидании ответа, который никогда не придет.

Первым делом вам нужно определить, как именно ведет себя такая библиотека. Я рекомендую писать небольшие тестовые сценарии, которые намеренно пытаются прервать работу библиотеки. Свяжите такой сценарий со средством тестирования (о них можно прочитать в разделе 5.7), которое соберет воедино все подключения, и посмотрите, как поведет себя библиотека при увеличении числа вызывающих потоков. Попробуйте организовать двадцать параллельных одинаковых запросов или вызовов и наблюдайте за реакцией. Если библиотека сторонних производителей работает с собственным пулом подключений, как только число запросов превысит размер этого пула, вы увидите падение пропускной способности. (Прочитать, как выглядит этот процесс, можно в разделе 9.1.) Скорее всего, вы спровоцируете взаимную блокировку внутри библиотеки. Это грустный факт, с которым ничего нельзя поделать. Даже если в процессе тестирования блокировка не возникнет, эта уязвимость все равно существует.

Если библиотека легко выходит из строя, нужно защитить потоки, отвечающие за обработку запросов. Установите время ожидания, если библиотека позволяет это сделать. В противном случае, возможно, придется прибегнуть к сложной структуре, например внешнему по отношению к библиотеке пулу рабочих потоков, которому потоки, обрабатывающие запросы, будут поручать выполнение опасных операций. Если вызов вовремя пройдет через библиотеку, рабочий поток встретится с исходным потоком запроса в результирующем объекте. Если же за указанное время вызов не завершится, поток, обрабатывающий запрос, откажется от этого вызова, даже если он был завершён рабочим потоком. Однако вступив на эту территорию, будьте осторожны. Чтобы справиться с задачей, потребуется хорошее знание параллельного программирования, организации пула потоков и особенностей модели потоков в конкретном языке, и все равно это будет обходной вариант. Перед тем как вступить на этот путь, имеет смысл попытаться получить у производителя лучший вариант клиентской библиотеки.

Блокированные потоки часто обнаруживаются рядом с точками интеграции. Они могут быстро привести к цепным реакциям. Зabloкированные потоки и медленное реагирование могут создать положительную обратную связь, превратив небольшую проблему в настоящую катастрофу.

ЗАПОМНИТЕ

Зabloкированные потоки являются непосредственной причиной большинства отказов

Отказы приложений практически всегда тем или иным образом связаны с заблокированными потоками. Сюда относится и неизменно популярное «постепенное снижение производительности» и «зависший сервер». Блокировка потоков ведет к цепным реакциям и каскадным отказам.

Внимательно изучайте пулы ресурсов

Подобно каскадным отказам, появление заблокированных потоков обычно связано с пулами ресурсов, особенно с пулами соединений с базами данных. Взаимная блокировка в базе данных может навсегда разорвать соединение и привести к некорректной обработке исключений.

Используйте проверенные примитивы

Изучите и применяйте безопасные примитивы. Может показаться, что реализовать собственную очередь производитель/потребитель достаточно легко, но это не так. Любая библиотека программ, работающих с параллельными процессами, протестирована куда лучше, чем ваша свежая разработка.

Защищайтесь с помощью таймаутов

Гарантировать, что ваш код не содержит взаимных блокировок, невозможно, но можно позаботиться о том, чтобы ни одна блокировка не продолжалась вечно. Избегайте бесконечного Java-метода `wait()`, пользуйтесь версией метода, позволяющей задать время ожидания. Всегда добавляйте время ожидания, даже если при этом вам придется перехватывать исключение `InterruptedException`.

Остерегайтесь использовать код, который вы не можете увидеть

В коде сторонних производителей могут скрываться самые разные проблемы. Будьте очень осторожны. Лично тестируйте такой код. По возможности старайтесь получить этот код и исследуйте возможные сюрпризы и режимы отказов¹.

4.6. Самостоятельно спровоцированный отказ в обслуживании



У людей самопожертвование иногда считается добродетелью, но к системам это не относится. Термин *самостоятельно спровоцированный отказ в обслуживании* (self-denial attack) описывает ситуацию, в которой система — или расширенная система, включающая в себя людей, — интригует против себя самой.

Классическим примером такой атаки является письмо от маркетологов «избранной группе пользователей», содержащее некую конфиденциальную информацию или предложение. Такие вещи распространяются быстрее, чем вирус Анна Курникова (или червь Морриса, который помнят старые программисты). Любое специальное предложение, предназначенное для группы из 10 000 пользователей, гарантированно привлечет миллионы. Сообщество сетевых любителей выгодных

¹ Именно по этой причине вы можете предпочесть библиотеки с открытым исходным кодом закрытым исходникам, как это делаю я.

сделок может обнаружить и распространить многоразовый купонный код за миллисекунды.

Великолепный случай подобного отказа в обслуживании произошел, когда игровая приставка Xbox 360 стала доступной для предварительного заказа. Было ясно, что в США спрос многократно превысит предложение, поэтому когда основные продавцы электроники разослали по электронной почте письма с рекламой предварительных заказов, они услужливо указали там точную дату и время начала акции. Эта реклама одновременно достигла сервиса сравнения цен FatWallet, портала объявлений о продаже электроники и, возможно, других сайтов, любимых охотниками за скидками. Также реклама предусмотрительно содержала внешнюю ссылку, которая случайно шла в обход сервиса Akamai, гарантируя, что каждое изображение, каждый JavaScript-файл и каждая электронная таблица будут загружаться непосредственно с серверов-источников.

За минуту до назначенного времени сайт вспыхнул, как свечка, а затем отключился. Для этого потребовалось всего шестьдесят секунд.

Проблемы, связанные с продажей приставки Xbox 360, испытал и сайт Amazon. В ноябре 2006 года он решил предложить 1000 экземпляров этой приставки всего за 100 долларов. Новость об этом распространилась по всему Интернету. Неудивительно, что тысяча приставок была продана за пять минут. Но, к сожалению, за это время не было продано больше ничего, потому что миллионы посетителей упорно нажимали кнопку **Reload**, пытаясь загрузить страницу со специальным предложением и воспользоваться огромной скидкой.

Судя по всему, Amazon не создал отдельный кластер серверов для обработки специального предложения (см. раздел 5.3). Специальное предложение, которое, скорее всего, означало продажу себе в убыток с целью привлечения покупателей или задумывалось для расширения трафика, куда больше испортило репутацию, чем принесло доход. Остается надеяться, что компания Amazon получила ценные сведения о слабых местах в архитектуре своего сайта — хотя обнаружение таких вещей за день до Черной пятницы того явно не стоит.

У любого, кто когда-либо имел дело с торговлей через Интернет, есть подобные истории. Иногда это код купона, который используют тысячи раз, иногда ошибка в цене, из-за которой одну товарную позицию заказывают больше, чем все прочие продукты вместе взятые. Как говорит Пол Лорд: «Хороший маркетинг может убить вас в любой момент».

Хороший маркетинг может убить вас в любой момент.

Но далеко не за каждый самострел вину можно переложить на отдел маркетинга (хотя, разумеется, можно попробовать это сделать). В горизонтальном слое при

наличии общих ресурсов один вышедший из повиновения сервер может сломать все остальное. Например, в инфраструктуре на базе ATG¹ всегда имеется диспетчер блокировок, отвечающий за управление распределенными блокировками, чтобы гарантировать согласованность кэша. (Любой сервер, желающий обновить класс `RepositoryItem` при включенном распределенном кэшировании, должен получить блокировку, обновить элемент, освободить блокировку и оповестить о недействительности кэша для данного элемента.) Этот диспетчер блокировок является уникальным ресурсом. При горизонтальном масштабировании сайта он превращается в бутылочное горлышко и в итоге становится угрозой. При случайном изменении популярного элемента (например, из-за ошибки в программе) все может кончиться тысячами сериализованных потоков обработки запроса на сотнях серверов, которые ждут блокировки на запись в этот элемент.

Избежать инициируемой машинами атаки типа «отказ в обслуживании» можно, построив архитектуру, в которой не будет места совместному доступу (см. врезку «Нет общим ресурсам» в разделе 4.7). Там, где это невозможно практически, для уменьшения влияния избыточного спроса применяйте разделение связующего ПО или сделайте ресурсы общего использования горизонтально масштабируемыми за счет избыточности и дополнительного протокола синхронизации. Также можно спроектировать резервный режим, который будет применяться, когда ресурсы общего доступа недоступны или не отвечают. К примеру, если диспетчер блокировок, обеспечивающий пессимистическую блокировку, недоступен, приложение может перейти на использование *оптимистической блокировки*.

Оптимистическая блокировка не ограничивает модификацию объектов сторонними сеансами, но при их сохранении проверяет на наличие коллизий. Пессимистическая блокировка накладывается перед предполагаемой модификацией данных на все строки, которые такая модификация предположительно затрагивает. Она более безопасна, но работает медленнее и требует больших согласований.

При наличии времени на подготовку и применении аппаратных средств балансировки нагрузки для управления трафиком можно сделать отдельный фрагмент инфраструктуры, обрабатывающий всплески трафика. Разумеется, это работает только в случае, когда экстраординарный трафик нацелен на часть вашей системы. Тогда даже если выделенный фрагмент прекратит работу, остальная часть системы продолжит функционировать в обычном режиме.

В этом случае, как только выделенные серверы прекратят работу, используйте подход с быстрым отказом (см. раздел 5.5). Это освободит входные ресурсы, такие

¹ ATG Commerce Suite — конкурирующий с J2EE сервер приложений и коммерческий фреймворк (см. <http://www.atg.com>).

как соединения веб-сервера и распределителя нагрузки, от ожидания бесполезного или несуществующего ответа.

Что же касается атак, спровоцированных людьми, ключом к их устранению являются обучение и донесение информации. Если вы держите каналы связи открытыми, можно надеяться, что вы сможете защитить систему от резких всплесков. Возможно, вы сможете помочь отделу маркетинга достичь своих целей, не ставя под угрозу систему.

ЗАПОМНИТЕ

Держите каналы связи открытыми

Самостоятельно спровоцированные отказы в обслуживании зарождаются внутри вашей собственной организации, когда умные маркетологи планируют свои флэшмобы и всплески трафика. Помочь и содействовать их усилиям, одновременно защитив систему, можно только при условии, что вы знаете о том, что вас ждет. Убедитесь, что никто не проводит рассылку с внешними ссылками. Создайте статические страницы в качестве «посадочной зоны» для первого перехода по таким ссылкам. Отслеживайте встроенные идентификаторы сеансов в URL-адресах.

Защищайте общие ресурсы

Ошибки в программе, неожиданные эффекты масштабирования и общие ресурсы — все это источники риска при резком увеличении трафика. Избегайте ситуаций, в которых увеличившаяся входная нагрузка вызывает экспоненциальный рост внутренних вычислений.

Будьте готовы к быстрому распространению любого заманчивого предложения

Любой, кто делает специальное предложение для ограниченного контингента, напрашивается на неприятности. Такой вещи, как распространение среди ограниченного контингента, попросту не существует. Даже если вы ограничиваете количество активаций фантастической сделки, вас все равно сметет толпа людей, безнадежно надеющихся приобрести игровую приставку Xbox 360 за 99 долларов¹.

4.7. Эффекты масштабирования



Применяемый в биомеханике закон квадрата-куба объясняет, почему не существует пауков размером со слона. Вес паука увеличивается вместе с его объемом, то есть пропорционально кубу коэффициента масштабирования или $O(n^3)$. А длина ног увеличивается пропорционально их квадрату, то есть $O(n^2)$.

¹ Если вы читаете эти строки тогда, когда цена на Xbox 360 уже опустилась ниже 99 долларов, подставьте вместо этой игровой приставки консоль следующего поколения.

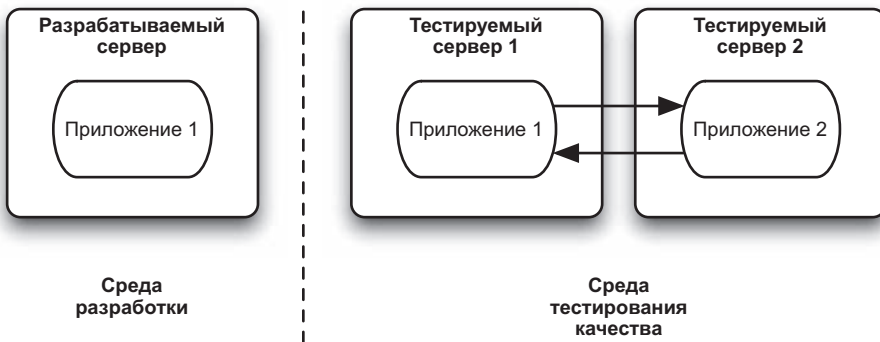
При десятикратном увеличении паука соотношение его веса и размера составит одну десятую от нормального соотношения, и его ноги просто не выдержат такой нагрузки.

Мы то и дело сталкиваемся с эффектами масштабирования. Они могут настичь нас каждый раз, когда при соотношении «многие к одному» или «многие к нескольким» одна из сторон увеличивается. Например, сервер базы данных, который отлично справляется с обращениями к нему двух серверов приложений, может полностью рухнуть после добавления еще восьми серверов приложений.

В среде разработки каждое приложение выглядит как один сервер. В процессе контроля качества практически любая система выглядит как один или два сервера. Но при переходе к реальным условиям выясняется, что некоторые приложения весьма компактны, в то время как другие могут быть средними, большими или огромными. А так как в процессе разработки и тестирования истинные размеры систем воспроизводятся редко, предугадать, где вы столкнетесь с эффектами масштабирования, невозможно.

Двухточечные связи

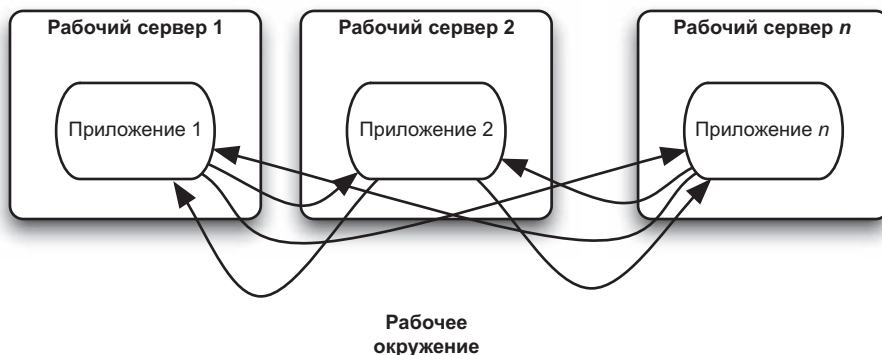
Одним из узких мест, где можно столкнуться с эффектами масштабирования, является двухточечная связь. Такая связь между серверами приложений, возможно, прекрасно работает в случае всего двух взаимодействующих узлов, как показано на рисунке.



При двухточечном соединении каждый экземпляр должен непосредственно общаться с соседом, как показано ниже.

Общее количество соединений пропорционально квадрату от числа узлов. Увеличьте число узлов до ста, и коэффициент масштабирования в $O(n^2)$ раз принесет вам немало головной боли. Это эффект многократного усиления, обусловленный

количеством экземпляров приложения. Приемлемость масштабирования в $O(n^2)$ раз зависит от конечного размера вашей системы. В любом случае, о подобных эффектах следует помнить до начала эксплуатации.



К сожалению, если вы не работаете в Microsoft или Google, вряд ли у вас получится построить тестовую ферму, повторяющую размер готовой системы. Дефекты такого типа невозможно выявить во время тестирования; систему следует конструировать таким образом, чтобы их там просто не было.

Это один из тех случаев, когда «лучшего» варианта не существует, есть только вариант, подходящий для конкретного набора условий. Если приложение будет работать исключительно на паре серверов, двухточечная связь вполне подойдет¹. Рост количества серверов требует другой стратегии взаимодействия. В зависимости от инфраструктуры можно заменить двухточечную связь следующими вариантами:

- ☐ широковещательная UDP-рассылка;
- ☐ групповая TCP- или UDP-рассылка;
- ☐ передача сообщений публикации/подписки;
- ☐ очереди сообщений.

Широковещательная рассылка (broadcast) дает необходимый эффект, но неэффективно расходует полосу пропускания. Кроме того, она дает дополнительную нагрузку на серверы, не заинтересованные в получаемых сообщениях, так как сетевая плата сервера, получив рассылку, должна уведомить стек TCP/IP. Групповая рассылка (multicast) более эффективна, так как распространяет сообщения только среди заинтересованных серверов. Еще лучше передача сообщений публикации/подписки, так как сервер может получать сообщения, даже если не слушал сеть в момент отправки. Разумеется, внедрение подобного решения зачастую сопровождается

¹ При условии, что взаимодействие настроено таким образом, чтобы не блокироваться при выходе из строя одного из серверов (см. раздел 4.5).

изрядными вложениями в инфраструктуру. Это прекрасная возможность применить принцип экстремального программирования, гласящий: «Выбирай самый простой из работающих вариантов».

Общие ресурсы

Другим эффектом масштабирования, ставящим под угрозу стабильность системы, является эффект «общих ресурсов». Часто встречающийся в сервис-ориентированной архитектуре ресурс общего доступа представляет собой нечто, что необходимо всем членам горизонтально масштабируемого слоя. В случае некоторых серверов приложений к таким ресурсам относится диспетчер кластера или диспетчер блокировки. При перегрузке он превращается в узкое место, ограничивающее вычислительную мощность (см. раздел 8.1). Рисунок 12 дает представление о том, как вызывающие стороны могут навредить общему ресурсу.

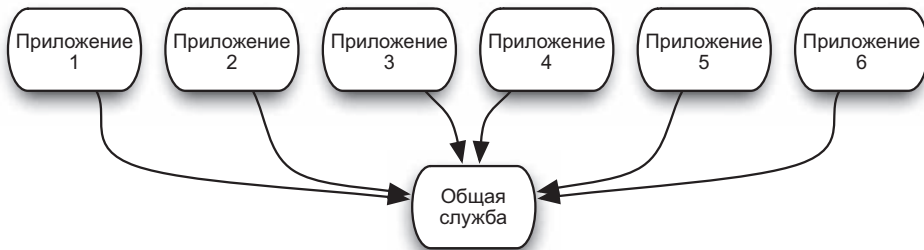


Рис. 12. Зависимость типа «многие к одному»

Если общий ресурс является избыточным и неисключительным, что подразумевает способность обслужить несколько потребителей за один раз, проблемы нет. При его насыщении достаточно его добавить, расширив тем самым узкое место.

Но зачастую общий ресурс выделяется для исключительного использования в то время, как клиент выполняет некую часть работы. В таких случаях вероятность конкуренции пропорциональна числу обрабатываемых слоем транзакций и числу клиентов в слое. (Эффект, оказываемый конкуренцией на пропускную способность, иллюстрируется в разделе 9.1.) Насыщение общего ресурса приводит к накоплению незавершенных соединений. Как только их число превышает прослушиваемую сервером очередь, появляются незавершенные транзакции. И в этот момент может произойти практически что угодно. Все зависит от того, какую функцию предоставляет общий ресурс. В частности, в случае с диспетчерами кэша (обеспечивающими согласованность распределенных кэшей) незавершенные транзакции приводят к устареванию данных или, что еще хуже, к нарушению их целостности.

ЗАПОМНИТЕ**Сравнивайте среду тестирования с условиями эксплуатации, чтобы обнаружить эффекты масштабирования**

Проблемы с эффектами масштабирования начинаются при переходе от небольшой однозначной среды разработки и тестирования к полноразмерной среде эксплуатации. При этом паттерны, которые прекрасно работали, могут сильно замедлиться или вообще перестать функционировать.

Осторожно используйте двухточечные связи

Двухточечные связи плохо масштабируются, так как количество связей пропорционально квадрату числа участников. Учета будущего роста системы вполне достаточно. Как только число серверов, с которыми вы имеете дело, начинает исчисляться десятками, переходите к какому-нибудь из вариантов связи типа «один ко многим».

Следите за общими ресурсами

Общие ресурсы могут превратиться в узкое место, ограничитель вычислительной мощности и угрозу стабильности. Если ваша система нуждается в таком типе ресурсов, как следует протестируйте их под нагрузкой. Также убедитесь, что клиенты смогут продолжить работу в случае замедления функционирования или блокировки общих ресурсов.

НЕТ ОБЩИМ РЕСУРСАМ

Лучше всего масштабируется **архитектура без общих ресурсов**. Каждый сервер функционирует независимо, не прибегая к координации действий или вызовам каких бы то ни было централизованных служб. В такой архитектуре производительность более-менее линейно пропорциональна числу серверов.

Проблема архитектуры без общих ресурсов состоит в том, что улучшение масштабирования достигается за счет перехода на резервный ресурс. К примеру, рассмотрим такой переход для сеанса. Пользовательский сеанс находится в памяти сервера приложений. Когда этот сервер прекращает работу, следующий запрос пользователя направляется другому серверу. Очевидно, что в идеале такая транзакция должна пройти прозрачно для пользователя, а значит, сеанс нужно загрузить на новый сервер приложений. Это требует некой координации между исходным сервером и **каким-то** устройством. Сервер приложений после каждого запроса страницы мог бы отправлять сеанс пользователя на резервный сервер. Либо сериализовать сеанс в таблицу базы данных или обрабатывать его совместно с другим специально выделенным сервером приложений. Существует несколько стратегий перевода сеанса на резервный ресурс, но все они включают вывод данных пользовательского сеанса за пределы исходного сервера. В большинстве случаев для этого требуется некий уровень общих ресурсов.

Приблизиться к архитектуре без общих ресурсов можно, сократив число обращающихся к общим ресурсам серверов. В примере с переключением сеанса это достигается путем выделения пар серверов, каждый из которых служит резервным для другого.

4.8. Несбалансированные мощности



Отраслевые издания переполнены чудесными историями о предоставлении вычислительных ресурсов так же, как коммунальных услуг. Суть такова: по мере роста спроса на ваше приложение оно автоматически получает больше ресурсов процессора, памяти и ввода-вывода. Кто добавляет эти ресурсы? Некая сущность в инфраструктуре следит за производительностью приложений, и обнаружив, что производительность не обеспечивает требуемого уровня обслуживания, выделяет дополнительные мощности. Эта «главная управляющая программа» работает в фоновом режиме, измеряя производительность системы и динамически перераспределяя ресурсы. Она должна гарантировать, что система никогда не будет испытывать их дефицита. Разумеется, за ее использование приходится платить, аналогично тому, как мы платим за воду и электричество. Звучит все фантастически. В смысле как фантастика. Отраслевые издания находятся в сговоре с производителями, которые чувствуют, что могут продать изрядное количество товаров для воплощения данной идеи. Создание настоящих вычислительных центров, предоставляющих подобные услуги, пока только в перспективе, а то, что мы имеем на текущий момент, слабо напоминает представленную картину.

В мире, где обитает 99,9 % из нас, производственные системы разворачивают на относительно фиксированном наборе ресурсов. Приложения запускаются в операционных системах, которые, в свою очередь, работают на некоем физическом оборудовании¹. Это оборудование имеет сетевые интерфейсы с подключенными к ним кабелями. Другим концом эти кабели вставляются в коммутатор. В традиционных центрах хранения и обработки данных для увеличения производительности работающей системы требуются недели. Нужно утвердить требования к аппаратному обеспечению; проверить доступность портов, мощность системы охлаждения, энергоемкость и пространство для серверных стоек; закупить оборудование; разобраться с заявками на изменения файлов; установить и подключить все устройства; установить операционную систему; обновить управляющую ресурсами базу данных; выделить LUN-адреса в сети хранения данных; настроить файловые системы; развернуть приложения; добавить приложения в кластер. В условиях кризиса это делается за несколько дней, особенно если удастся ограбить на время чьи-нибудь серверы, тем самым пропустив фазы закупки и установки. Три года назад я видел, как шесть дополнительных серверов были переустановлены и перенастроены на работу с новыми приложениями за 36 часов единственным блестящим инженером. Тодд, снимаю перед тобой шляпу.

¹ Все чаще физическое оборудование представляет виртуальная машина. Тем не менее на практике добавление, удаление и перенос виртуальных машин в ответ на изменившийся спрос происходит медленнее, чем вы думаете. Самая инициативная из известных мне рабочих групп производила балансировку виртуальных машин только раз в день и ожидала замедления их работы, потому что миграция виртуальных машин была вовлечена в процесс более тщательного контроля над изменениями.

Эти шесть дополнительных серверов спасли запуск системы, превратив формулировку «полный провал» во вполне достойную: «мероприятие, проведенное с незначительными недостатками».

Все это означает, что — исключая кризисы в их экстремальном проявлении — вам приходится иметь дело только с теми ресурсами, которые у вас уже есть. Вы не можете наращивать мощность для обслуживания краткосрочных пиковых нагрузок (длящихся несколько часов или дней). Производительность системы может меняться со временем по причине выхода новых версий кода, настройки, оптимизации, изменения конфигурации сети или архитектуры, но в каждый конкретный момент времени она по своей сути статична.

Производительность аппаратного обеспечения в течение коротких промежутков времени является фиксированной.

В рамках предприятия это повышает вероятность отказов многоуровневых систем или систем, полагающихся на другие приложения.

На рис. 13 для входного веб-сайта доступно 3000 потоков, обрабатывающих запросы. Во время пиковой загрузки большая часть из них будет обслуживать страницы каталога продуктов или результаты поиска. Некоторое небольшое количество будет заниматься различными корпоративными «информационными» страницами. Несколько потоков будут отвечать за оформление заказов и оплаты. Из них небольшая часть займется запросами к системе планирования, выясняя, нельзя ли установить продукт заказчику силами местной сервисной группы. Математика позволяет предсказать, сколько потоков одновременно могут обращаться к системе планирования. Расчет в данном случае несложен, хотя и опирается на статистические данные и ряд допущений, что, как известно, легко подтасовать. Но пока система планирования в состоянии в соответствии с предсказанным спросом обслуживать достаточное количество одновременных запросов, результат расчета можно считать удовлетворительным.

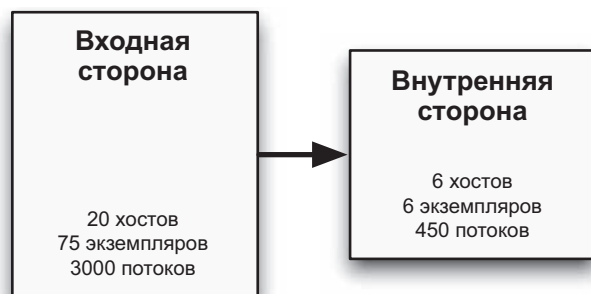


Рис. 13. Несбалансированные мощности

Но на самом деле это не так.

Предположим, отдел маркетинга самостоятельно провоцирует атаку типа «отказ в обслуживании», предложив на один день бесплатную установку любого дорогостоящего оборудования. И внезапно вместо небольшой части входных потоков, вовлеченных в запросы планирования, вы обнаруживаете, что их в два, в четыре или даже в десять раз больше. Дело в том, что входная часть системы всегда может перегрузить внутреннюю часть запросами, так как мощности входной и внутренней сторон системы не сбалансированы между собой.

По многим причинам равное выделение вычислительной мощности каждой системе непрактично. В рассматриваемом примере наращивание системы планирования до размеров веб-сайта на случай, если она когда-то кому-то понадобится, является напрасной тратой средств. Ведь эта инфраструктура будет простаивать 99 % времени, исключая один день в пять лет!

Поэтому если вы не можете построить достаточно большую систему планирования, чтобы удовлетворить потенциально чрезмерный спрос входной стороны, нужно обеспечить обеим сторонам устойчивость перед цунами запросов. В случае входной стороны снизить нагрузку на внутреннюю часть при замедлении ответов или отклонении соединений поможет паттерн предохранителя (Circuit Breaker). В случае внутренней стороны паттерн квитирования (Handshaking) будет информировать входную сторону о необходимости снижения частоты запросов. Не забудьте также и про паттерн переборок (Bulkheads), чтобы резервировать некоторую долю вычислительной мощности внутренней стороны для транзакций других типов.

Решение проблем на этапе тестирования

Проблема несбалансированных мощностей во время тестирования практически никогда не возникает. В основном потому, что для тестирования каждая система, как правило, масштабируется всего до двух серверов. В итоге в ходе интегрального тестирования два сервера представляют входную систему, а еще два — внутреннюю, что дает соотношение один к одному. А в процессе эксплуатации это соотношение может составить десять к одному или того хуже.

Следует ли сделать для тестирования точную уменьшенную копию всего предприятия? Это было бы здорово, но такой возможности у нас нет. Но вы можете воспользоваться тестовой программой (см. раздел 5.7). Путем имитации проседающей под нагрузкой внутренней части тестовая программа позволит удостовериться, что входная сторона системы будет терять функциональность постепенно и предсказуемым образом.

В то же время, предлагая свою внутреннюю систему, вы, вероятно, ожидаете «нормальной» нагрузки. То есть вы резонно ждете, что сегодняшнее распределение спроса и типов транзакций примерно совпадет со вчерашней нагрузкой. Если все

остальное останется без изменений, значит, ваше допущение было верным. К изменению нагрузки на систему могут привести многие факторы: маркетинговые кампании, публичное упоминание сайта, новые варианты кода для входных систем и даже ссылки на портале новостей Slashdot или новостных социальных сайтах Fark и Digg. Как поставщик внутренней системы вы слабо понимаете стремления отдела маркетинга, который намеренно вызывает все эти изменения трафика. Еще менее предсказуемы результаты освещения в прессе.

Так что же можно сделать, если ваша система сталкивается с такими непредсказуемыми наплывами? Будьте готовы ко всему. Во-первых, смоделируйте производительность своей системы, дабы убедиться, что вы представляете, что происходит, по крайней мере, примерно. Три тысячи потоков, инициирующих вызовы семидесяти пяти потоков, — это *нереальная* ситуация. Во-вторых, тестируйте систему не только под стандартной нагрузкой. Посмотрите, что получится, если взять количество вызовов, которое может прийти с входной стороны, удвоить его и нацелить на самую ресурсоемкую транзакцию. Устойчивая система замедлит свою работу или даже быстро завершит ее, если не сможет обработать транзакцию за выделенное время, но после устранения нагрузки она снова начнет функционировать в обычном режиме. Аварийное завершение, зависшие потоки, пустые отклики или даже бессмысленные ответы показывают, что система не выживет, а возможно, даже инициирует каскадный отказ.

ЗАПОМНИТЕ

Учитывайте количество серверов и потоков

В процессе разработки и тестирования ваша система, скорее всего, выглядит как один или два сервера, как и все прочие версии остальных систем. В режиме эксплуатации вместо соотношения один к одному может возникнуть соотношение десять к одному. Учитывайте количество входных и внутренних серверов, а также количество потоков, которые каждая из сторон может обработать.

Учитывайте эффекты масштабирования и пользователей

Несбалансированная производительность представляет собой особый случай эффектов масштабирования: одна из сторон масштабируется намного сильнее другой. Изменения в трафике — сезонные, обусловленные рынком или освещением в прессе — могут стать причиной того, что совершенно корректная входная система приведет к переполнению внутренней системы аналогично тому, как посты на сайтах Slashdot или Digg порождают трафик, с которым сайты не могут справиться.

Нагружайте обе стороны интерфейса

Если вы предоставляете внутреннюю систему, исследуйте, что происходит, если внезапно увеличить нагрузку в десять раз против обычной, выполняя самую ресурсоемкую транзакцию. Прекратит ли система свою работу? Или работа замедлится, а потом вернется на обычный уровень? Если вы предоставляете внешнюю систему, проверяйте, что происходит в случае, когда обращения к внутренней стороне остаются без ответа или ответ приходит крайне медленно.

4.9. Медленная реакция



Как отмечено в разделе 4.1, медленная генерация ответа намного хуже, чем отказ в соединении или возвращение ошибки, особенно в ситуации со службами промежуточного уровня в сервис-ориентированной архитектуре.

Быстрый отказ позволяет вызывающей системе оперативно завершить обработку транзакции. Результат этого завершения — успешный или неудачный — зависит от логики приложения. А вот медленный ответ связывает ресурсы как вызывающей, так и вызываемой системы.

Причиной медленных ответов, как правило, является избыточный спрос. Когда все доступные обработчики запросов заняты, резерва для приема новых запросов нет. Кроме того, медленные ответы могут указывать на более глубокую проблему. Например, именно так проявляются утечки памяти, ведь виртуальной машине все тяжелее выделять достаточное для обработки транзакции пространство. Это выглядит как высокая загрузка процессора, хотя на самом деле обусловлено сборкой мусора, а не обработкой транзакций.

Еще я сталкивался с медленной реакцией из-за перегрузки сети. В LAN такая ситуация возникает относительно редко, а вот в WAN она вполне реальна, особенно если протокол слишком «разговорчив». Но чаще я сталкивался с приложениями, которые полностью опустошают буфер передачи, при этом позволяя переполнить буфер приема, из-за чего возникает открытое, но не работающее TCP-соединение. Такое обычно случается в «самопальных» низкоуровневых протоколах сокета, в которых подпрограмма `read()` не входит в цикл, не получив всего содержимого буфера.

Медленное реагирование, как правило, распространяется вверх от слоя к слою, как постепенно нарастающий вид каскадного отказа.

Если система имеет возможность следить за собственной производительностью (подробно мы поговорим об этом в главе 17), она сможет сообщить, когда выполнение договора о сервисном обслуживании станет невозможным. Представьте, что ваша система является поставщиком услуги и обязана реагировать в течение ста миллисекунд. Как только скользящий средний показатель в течение двадцати последних транзакций превысит это значение, система может отказаться выполнять запросы. Это может произойти на уровне приложения, где система начнет возвращать ошибку внутри определенного протокола. Или на уровне соединений, отказываясь от новых подключений к сокету. Разумеется, любой из этих вариантов отказа должен быть детально описан в документации и не являться неожиданностью для вызывающих сторон. Так как разработчики подобных систем, без сомнения, прочитают эту книгу, к отказам они будут готовы и их системы смогут аккуратно справиться с данной ситуацией.

ЗАПОМНИТЕ**Медленные ответы вызывают каскадные отказы**

Вышележащие системы, столкнувшись с медленной реакцией, также замедляют свою работу и, когда время реакции превысит их собственное время ожидания, могут оказаться уязвимыми в плане стабильности.

Медленные ответы ведут к росту трафика сайтов

Ожидающие загрузки страниц пользователи часто щелкают на кнопке перезагрузки страницы, генерируя дополнительный трафик для и так перегруженной системы.

Используйте принцип быстрого отказа

Если система следит за собственной скоростью отклика¹, она сможет определить факт замедления работы. Рассмотрите возможность отправки сообщения об ошибке в случаях, когда среднее время ответа превышает допустимое в данной системе (или, по крайней мере, когда среднее время ответа превышает время ожидания вызывающей стороны).

Ищите утечки памяти и конфликты ресурсов

Конкуренция из-за недостаточного количества подключений к базе данных приводит к медленным ответам. А медленные ответы, в свою очередь, усугубляют эту конкуренцию, создавая положительную обратную связь. Утечки памяти заставляют сборщик мусора работать с чрезмерной нагрузкой, что опять-таки замедляет реакцию системы. К такому же результату приводят неэффективные низкоуровневые протоколы.

4.10. SLA-инверсия



Соглашение об уровне обслуживания (Service-Level Agreement, SLA) представляет собой контракт, оговаривающий, как именно организация должна предоставлять свои услуги. Существуют количественные показатели оказания услуг с финансовыми санкциями в случае их нарушения. Объединение некоторых тенденций повышает важность SLA. Основным фактором является привлечение людей, инфраструктуры и операций. Интерес к SLA подогревает знакомство с библиотекой ИТ-инфраструктуры (IT Infrastructure Library, ITIL), которая скоро будет причислена к стандарту ISO 20000², и средой управления ИТ-услугами (IT Service Management Framework, itSMF)³. Кроме того, в сфере информационных технологий существует общая направленность на повышение степени профессионализма. Администраторы информационных систем считают себя поставщиками важных услуг, необходимых для успешного функционирования их предприятий. Им требуются количественно выраженные соглашения об уровне обслуживания, по-

¹ Эта тема раскрывается в главе 17.

² См. https://ru.wikipedia.org/wiki/ISO_20000 и <http://www.itil.co.uk/>.

³ См. <https://ru.wikipedia.org/wiki/ITSM> и <http://www.itsmf.com/>.

звolyющие выделять ресурсы в соответствии с производственной необходимостью, чтобы не отвечать на стандартные жалобы: «Мое приложение работает слишком медленно, сделайте что-нибудь».

На рис. 14 новый сайт компании — проект Frammitz — создан для высокой доступности. Это критически важно, поэтому избыточность присутствует на всех уровнях: электропитание, сеть, хранение, серверное оборудование, приложения. Его архитектура является неразделяемой (см. врезку «Нет общим ресурсам» в разделе 4.7), чтобы обеспечить максимальную горизонтальную масштабируемость без узких мест. В требованиях SLA указано, что доступность сайта Frammitz должна составить 99,99 %. То есть в месяц допускается чуть более четырех минут простоя.

Но несмотря на тщательное проектирование, сайт Frammitz может удовлетворить требования SLA только по счастливой случайности.



Рис. 14. Архитектура проекта Frammitz

Сам по себе проект системы предполагает высокую доступность, но он завязан на ряд других служб. Автономная веб-система, не связанная с расчетной системой, бухгалтерской системой, системой учета ресурсов или системой инвентаризации,

вряд ли сможет много продать. Добавим средства распознавания мошенничества, интеграции с торговыми партнерами, передачи ключевых сервисов внешним подрядчикам, защиты от спама, услуги геокодирования, проверку адресов и авторизацию кредитных карт и получим настоящую паутину. И каждая из этих зависимостей на другом конце уязвима с точки зрения SLA. Рисунок 15 демонстрирует системы, от которых зависит проект Frammitz, и соответствующие им показатели SLA.

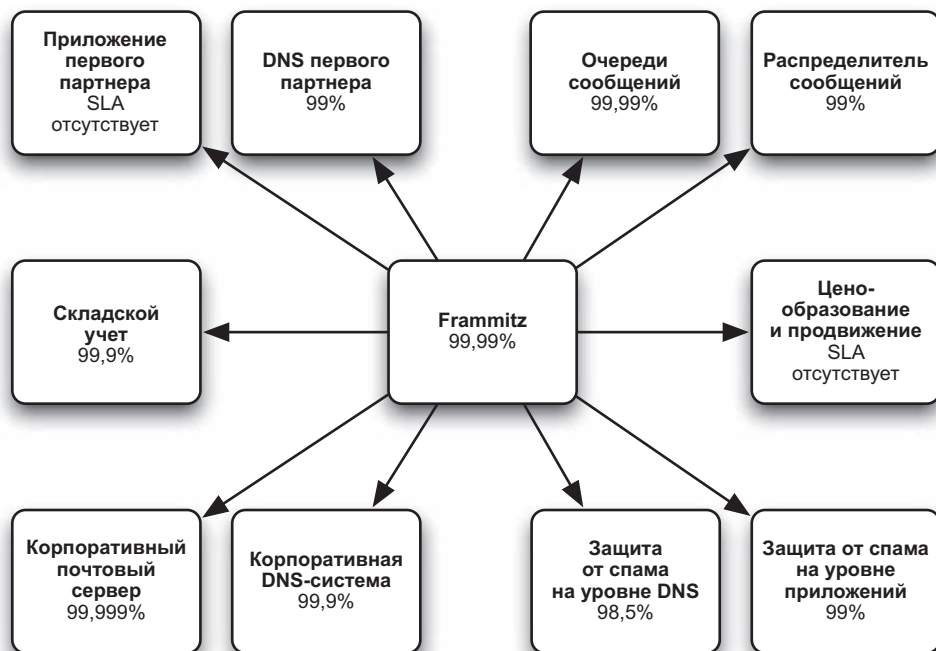


Рис. 15. Внешние зависимости проекта Frammitz

С любой службой — как внутри, так и вне вашей компании — система зависит от доступности транспортного уровня, служб именования (DNS) и протоколов уровня приложения. Отказать может любой из этих слоев при любых внешних соединениях. И если каждая из ваших зависимостей не предусматривает то же самое соглашение об уровне обслуживания, за выполнение которого вы взялись, то максимум, что вы можете обещать, — это обеспечить выполнение SLA самого худшего из ваших провайдеров. Так как проект Frammitz зависит от партнера 1, а также ценообразования и продвижения, не предусматривающих никаких SLA, то, строго говоря, в своем варианте SLA он не может предложить гарантированный уровень доступности.

Закон вероятности рисует еще более удручающую ситуацию. Если подойти упрощенно, вероятность отказа проекта Frammitz равна суммарной вероятности отказа в любом его компоненте или в любой службе. То есть единственного отказа в зависимости достаточно, чтобы сайт Frammitz перестал работать. Можно написать:

$$P(\text{сайт работает}) = (1 - P(\text{внутренний отказ})) \times P(\text{партнер 1 работает}) \times \\ \times P(\text{складской учет работает}) \dots$$

Если сайт Frammitz работает с пятью внешними службами, доступность каждой из которых равна 99,9 %, то *максимум* доступности, который его владельцы могут обещать, составит 99,5 %.

Обращение к сторонним поставщикам всегда снижает уровень обслуживания.

Если полностью отсоединить сайт Frammitz от внешних систем, вероятность прекращения его работы вычисляется просто по формуле $P(\text{внутренний отказ})$. Отказ большинства систем происходит где-то в этом промежутке.

В этом и состоит SLA-инверсия: система, обязанная обеспечить высокую доступность, зависит от систем с меньшей доступностью. И для обеспечения оговоренной в SLA высокой доступности приходится выдавать желаемое за действительное. Возможны две основные реакции на SLA-инверсию. Прежде всего, можно изолироваться от систем с более низкими показателями SLA. Убедитесь, что ваше приложение сможет продолжить функционирование без удаленной системы. Действуйте аккуратно. Удаление промежуточного ПО может дать прекрасные результаты, но эта возможность зависит от характера удаленного сервиса. Бывают ситуации, когда подобный вариант отпадает. Тогда желательно задействовать некие предохранители, защищающие приложение от всех его союзников/потенциальных врагов. Кроме того, будьте осторожны при составлении соглашения об уровне обслуживания. Ни в коем случае не пишите там «доступность 99,99 %». (Подробнее эта тема рассматривается в главе 13.)

Вместо этого сведите обсуждение к доступности отдельных функций или программных компонентов системы. Не зависящие от сторонних служб программные компоненты могут иметь максимально высокие показатели SLA. У связанных со сторонними службами программных компонентов этот параметр может быть равен уровню доступности, предлагаемому третьей стороной, минус вероятность отказа в вашей собственной системе. Это ИТ-эквивалент второго закона термодинамики¹: уровень сервиса всегда только понижается.

¹ Энтропия всегда возрастает.

ЗАПОМНИТЕ**Не давайте пустых обещаний**

SLA-инверсия означает, что вы выдаете желаемое за действительное: вы обещаете уровень сервиса, достичь которого можно только при большом везении.

Исследуйте все зависимости

SLA-инверсия подстерегает вас в самых неожиданных местах, в частности в сетевой инфраструктуре. К примеру, каково соглашение об уровне обслуживания у вашего корпоративного DNS-кластера? (Я надеюсь, что у вас там кластер.) Как насчет SMTP-сервиса? Очередей сообщений и почтовых служб? Сети хранения данных предприятия? Зависимости подстерегают нас везде.

Изолируйте систему, оговоренную в вашем соглашении об уровне обслуживания

Сделайте так, чтобы система работала даже при отключении зависимостей. Если отказ зависимой службы приводит к отказу вашей системы, значит, с математической точки зрения доступность вашей системы всегда будет ниже, чем доступность зависимой службы.

4.11. Огромные наборы результатов



Подходите к проектированию с изрядной долей скептицизма, и вы добьетесь устойчивости. Задайте себе вопрос: «Каким образом система X может причинить мне неприятности?», а потом ищите способ, позволяющий избежать любых неожиданностей, которые может преподнести вам предполагаемый союзник.

Если ваше приложение напоминает большинство других, скорее всего, оно относится к серверу баз данных с избыточным доверием. Я собираюсь убедить вас, что здоровая доля скептицизма поможет вашему приложению избежать некоторых неприятностей.

Повсеместно используется следующая структура кода: базе данных отправляется запрос, а затем результаты просматриваются в цикле с обработкой каждой строки. Зачастую обработка строки означает добавление нового объекта данных в существующий набор. Что произойдет, если база данных внезапно вернет пять миллионов строк вместо привычной сотни? Если ваше приложение в явном виде не ограничивает набор результатов, которое оно собирается обрабатывать, все может закончиться истощением памяти или попыткой довести до конца цикл тогда, когда пользователь уже давно потерял интерес к затянувшемуся процессу.

Черный понедельник

Вы когда-нибудь открывали удивительные черты в старых знакомых? Обнаруживали внезапно, что самый скучный парень в офисе увлекается прыжками с парашютом? У меня такое случилось с моим любимым сервером электронной

коммерции. Однажды безо всякого предупреждения все экземпляры фермы — более сотни отдельных серверов со сбалансированной нагрузкой — стали плохо себя вести. Все казалось почти случайным. С сервером все было в порядке, а через пять минут он вдруг начинал полностью загружать процессор. А еще через три-четыре минуты он прекращал работу с ошибкой памяти в виртуальной машине HotSpot. Рабочая группа перезагружала серверы с максимальной быстротой, но на запуск и предварительную загрузку кэша требовалось несколько минут. Иногда очередной сервер прекращал работу до завершения запуска. Мы не могли поднять производительность выше 25 %.

Представьте (или вспомните, если вы с таким сталкивались), как в 5 часов утра вы пытаетесь бороться с совершенно новым режимом отказа, одновременно участвуя в телеконференции на двадцать человек. Некоторые участники конференции рапортовали о текущем состоянии, другие пытались придумать быстрые способы восстановления работоспособности, кто-то старался докопаться до первопричины, а кто-то нес околесицу.

Мы послали системного администратора и сетевого инженера искать следы атак отказа в обслуживании. Администратор нашей базы данных отчитался, что с базой все в порядке, но она испытывает большую нагрузку. Это было понятно, потому что при запуске каждый сервер посылал сотни обращений, готовя свои кэши к приему запросов. Но некоторые серверы прекращали работу еще до этого момента, то есть ситуация не была связана с входящими запросами. Высокая загрузка процессора заставила меня подумать о сборке мусора, и я сказал группе, что проблемы следует искать в функционировании памяти. Разумеется, когда я взглянул на доступную динамическую память на одном из серверов, то выяснилось, что ее объем стремится к нулю. И через некоторое время после достижения этого значения виртуальная Java-машина (Java Virtual Machine, JVM) получила ошибку HotSpot.

Обычно нехватка памяти для виртуальной Java-машины сопровождается исключением `OutOfMemoryError`. Полное прекращение работы возникает только при выполнении некоего нативного кода, не проверяющего на значение `NULL` после вызова метода `malloc()`. Единственный известный мне нативный код принадлежал JDBC-драйверу типа 2¹. Драйверы этого типа используют тонкий Java-слой для обращения к базе данных нативной API-библиотеки производителя. Разумеется, вывод дампа стека показал, как исполняется код глубоко внутри драйвера базы данных.

Но как сервер взаимодействовал с базой данных? Я попросил администратора базы отследить поступающие от серверов приложений обращения. Вскоре очередной сервер прекратил работу, и мы смогли посмотреть, чем он занимался перед «входом

¹ Для людей, не занимающихся Java-программированием, поясню: *нативный код* означает полностью скомпилированные инструкции к главному процессору. Обычно это код на языке C или C++ в динамически подключаемых библиотеках. Нативный код печально известен как источник отказов под нагрузкой.

в сумеречную зону». Но все обращения выглядели совершенно безобидно. Рутинные процедуры. Ни следа кустарных SQL-монстров, которые я иногда встречал (запросы на восемь полей с пятью объединениями в каждом подзапросе и пр.). Последнее обнаруженное мной обращение относилось к таблице сообщений, которую сервер использовал в качестве своей JMS-реализации, умеющей работать с базой данных. Ее экземпляры в основном информировали друг друга, когда следует опустошить соответствующий кэш. Размер таблицы не должен был превышать 1000 строк, но администратор базы данных сказал, что она возглавляла список самых ресурсоемких запросов.

По какой-то причине в этой обычно небольшой таблице было более десяти миллионов строк. Так как серверы приложений были запрограммированы извлекать из таблицы все строки, каждый из них пытался получить десять с лишним миллионов сообщений. Строки блокировались, потому что сервер приложений выдал обращение «выбрать для обновления». При попытке превратить сообщение в объекты он израсходовал всю доступную память и, в конечном счете, прекратил работу. После этого база данных откатила транзакцию назад, освободив блокировку. Затем счастья попытал следующий сервер приложений, запросив ту же таблицу. Таким образом, нам пришлось проделать огромную работу, просто чтобы нейтрализовать отсутствие в обращении сервера приложений предложения LIMIT. К моменту, когда систему удалось стабилизировать, черный понедельник закончился... наступил вторник.

В конечном счете мы поняли, откуда в таблице взялись десять миллионов сообщений, но это была уже другая история.

Этот вид отказа может возникнуть при обращении к базам данных, доступе к определенным объектам или веб-службам. Появляется он и при AJAX-запросах к веб-серверам. Распространенная форма такого отказа наблюдается при просмотре связей «главный-подчиненный». Так как на стадии разработки используются, как правило, небольшие базы данных, разработчики вряд ли имеют шанс столкнуться с подобными последствиями. Но после эксплуатации системы в течение года даже простой просмотр «выборки заказов клиента» может вернуть огромный набор результатов.

Отвлеченно говоря, огромный набор результатов возникает, когда вызывающая сторона позволяет другой стороне диктовать условия. Это отказ в квитировании. В любом API или протоколе вызывающая сторона всегда должна обозначать, какое количество результатов она готова принять. Протокол TCP делает это в строке заголовка «window». Интерфейс поисковых служб позволяет указывать, сколько результатов следует вернуть и каким должно быть начальное смещение. Стандартного синтаксиса SQL для указания предельного числа результатов не существует. Инструменты ORM (такие, как Hibernate и iBatis) поддерживают параметр запроса, ограничивающий число возвращаемых результатов, но при обработке ассоциаций (таких, как содержимое контейнера) этого не происходит. Следовательно, стоит опасаться любых соотношений, позволяющих накапливать

неограниченное количество потомков, например соотношения заказов к строкам заказа или пользовательских профилей к посещениям сайта. Под подозрение попадают и сущности, занимающиеся регистрацией изменений.

Если вы пишете собственный SQL-код, используйте один из следующих рецептов ограничения количества извлекаемых строк:

```
-- Microsoft SQL Server
SELECT TOP 15 colspec FROM tablespec

-- Oracle (since 8i)
SELECT colspec FROM tablespec
WHERE rownum <= 15

-- MySQL and PostgreSQL
SELECT colspec FROM tablespec
LIMIT 15
```

Неполным решением (но это лучше, чем ничего) будет запрос всех результатов с выходом из обрабатывающего их цикла при достижении некоторого максимального количества строк. Это обеспечивает серверу приложений дополнительную стабильность, хотя и за счет впустую расходуемой производительности базы данных.

Огромные наборы результатов являются распространенной причиной медленной реакции. Они могут быть следствием нарушения стабильного состояния.

ЗАПОМНИТЕ

Используйте реалистичные объемы данных

Наборы данных, с которыми приходится иметь дело на стадиях разработки и тестирования, как правило, слишком малы для возникновения данной проблемы. Чтобы посмотреть, что произойдет, когда запрос вернет миллион строк, которые следует превратить в объекты, вам нужен набор данных, получаемый во время эксплуатации. Побочной выгодой от тестирования с подобным набором данных является более точная информация о производительности системы.

Не полагайтесь на поставщиков данных

Даже если вы думаете, что ответом на запрос всегда будет небольшой набор результатов, помните, ситуация может измениться без предупреждения из-за какой-то другой части системы. Единственные значения, имеющие смысл, это «ноль», «один» и «много». Так что если в запросе не указано, что он должен возвращать ровно одну строку, он потенциально может вернуть слишком много строк. Не стоит полагаться на поставщика данных, считая, что он вернет ограниченный набор. Рано или поздно он может сойти с ума и без всяких поводов переполнить таблицу. И что вы тогда будете делать?

Вставляйте ограничители в другие протоколы прикладного уровня

Обращения к веб-службам, RMI, DCOM, XML-RPC — все эти вещи могут вернуть вам огромный набор объектов, что потребует слишком большого объема памяти.

5

Паттерны стабильности

Теперь, когда вы знаете, каких антипаттернов следует избегать, взглянем на другую сторону медали. В этой главе вы познакомитесь с некоторыми паттернами, которые являются противоположностью описанных в предыдущей главе «убийц». Восемь полезных паттернов помогут вам в построении архитектуры и проектировании, позволяя уменьшить, устранить или смягчить влияние трещин на систему. Они не только помогут программному обеспечению пройти контроль качества, но и гарантируют вам после его запуска ночной сон и ничем не прерываемое общение с семьей.

Но не следует думать, что система с большим числом паттернов превосходит систему, в которой их меньше. «Количество примененных паттернов» не является показателем качества. Вам нужно выработать мышление, направленное в первую очередь на восстановление работоспособного состояния. Рискуя показаться занудой, повторю еще раз: всегда ждите отказа. И грамотно применяйте паттерны для снижения вреда, причиняемого отдельными отказами.

5.1. Задавайте таймауты



В былые времена проблемы, связанные с сетями, задевали только программистов, работавших с низкоуровневым программным обеспечением: операционными системами, сетевыми протоколами, удаленными файловыми системами и т. п. Сейчас практически все, кроме самых тривиальных приложений, в той или другой форме задействует сети, а значит, каждого приложения касается фундаментальное правило: сети подвержены отказам. Поврежденным

может оказаться кабель, маршрутизатор, коммутатор или компьютер, к которому вы обращаетесь. И даже если соединение уже установлено, любой из этих элементов может отказать в любую минуту. Если подобное произойдет, ваш код не сможет просто замереть и ждать ответа, который, возможно, никогда не придет; раньше или позже он должен прекратить свою работу. Ожидание не относится к методам проектирования.

Отныне и навсегда сети останутся ненадежными.

Таймаут — это простой механизм, который позволяет прекратить ожидание ответа, если вы думаете, что он не придет. Как-то был у меня проект по переносу библиотеки сокетов BSD в UNIX-среду на базе мейнфрейма. Я был вооружен стопкой документов RFC и массой непонятного программного кода для UNIX версии System V Release 4. На протяжении всего проекта мне не давали покоя два вопроса. Во-первых, интенсивное использование блоков `#ifdef` для различных архитектур создавало впечатление, что эту систему перенести куда-либо сложнее, чем двадцать разных операционных систем, перемешанных друг с другом. Во-вторых, сетевой код был перенасыщен процедурами обработки ошибок на базе таймаутов разных видов. Тем не менее к концу выполнения проекта я смог понять и оценить значение этих таймаутов.

Запрос для обсуждения (Request For Comments, RFC) — это документ из серии пронумерованных информационных документов Интернета, содержащих технические спецификации и стандарты, широко применяемые во всемирной сети.

ДЖО СПРАШИВАЕТ: ВСЕ ЭТО НАГРОМОЖДЕНИЕ ДЕЙСТВИТЕЛЬНО НУЖНО? —

Вам может показаться, как казалось мне во время переноса библиотеки сокетов, что обработка всех возможных таймаутов необоснованно усложняет код. Разумеется, уровень сложности при этом возрастает. Может оказаться, что половина вашего кода будет посвящена обработке ошибок, а не необходимым функциональным возможностям. Но я утверждаю, что направленность на эксплуатацию, а не просто на прохождение тестирования, определяется способностью справляться с возникающими проблемами. Корректно реализованная обработка ошибок повышает устойчивость системы. И хотя вы не дождетесь за это благодарности от пользователей, потому что стабильно работающую систему считают само собой разумеющейся вещью, зато никто не будет будить вас по ночам.

Удобно расположенные обработчики таймаутов обеспечивали локализацию неисправностей, в результате проблема в другой системе, подсистеме или устройстве не становилась вашей проблемой. К сожалению, с повышением уровня абстракции и отходом от низменного мира аппаратного обеспечения корректная

расстановка таймаутов становится все большей редкостью. Во многих высокоуровневых API почти или полностью отсутствует возможность явного задания параметров времени ожидания. Предоставляемые производителями клиентские библиотеки печально известны полным отсутствием таймаутов. API-библиотеки часто обрабатывают связи между сокетами от имени приложения. Скрывая от приложения реальный сокет, они мешают приложению установить жизненно необходимые таймауты.

Таймауты имеют значение и в рамках одного приложения. Истощиться может любой пул ресурсов. Принято блокировать вызывающий поток, пока не будет проверен один из ресурсов (см. раздел 4.5).

Важно, чтобы любой пул ресурсов, блокирующий потоки, имел время ожидания, гарантирующее, что в конечном счете блокировка будет снята вне зависимости от доступности ресурсов.

Также будьте осторожнее с методом `java.lang.Object.wait()`. Используйте такую форму, в которой существует возможность указать время ожидания в качестве аргумента, вместо более простой формы, не имеющей такой возможности. То же самое касается классов новой библиотеки `java.util.concurrent`¹. Всегда пользуйтесь такими версиями методов `poll()`, `offer()` и `tryLock()`, которые принимают время ожидания в качестве аргумента. В противном случае вы можете попасть в ситуацию вечного ожидания.

Для удобства работы с вездесущими таймаутами нужно разбить длительные операции на набор примитивов, которые могут использоваться в разных местах. Предположим, вам нужно выгрузить соединение с базой данных из пула ресурсов, запустить запрос, преобразовать ответ `ResultSet` в объекты и вернуть соединение в пул. По меньшей мере в трех точках этой последовательности взаимодействий может произойти зависание на неопределенное время. Вместо того чтобы писать весь этот код в десятке мест с сопутствующей обработкой таймаутов (не говоря уже о других видах ошибок), создайте объект `QueryObject` (см. книгу *Patterns of Enterprise Application Architecture* Мартина Фаулера), который будет представлять изменяемую часть взаимодействия.

В качестве шаблона для обработки подключений, обработки ошибок, исполнения запросов и обработки результатов используйте обобщенный компонент `Gateway` (см. класс `JdbcTemplate` фреймворка Spring²). Объединение этого общего паттерна взаимодействия в один класс облегчает применение паттерна предохранителя (Circuit Breaker).

¹ Появилась в Java 5. Работающие с более ранними версиями могут загрузить эту библиотеку и соответствующим образом изменить имена пакетов.

² См. <http://static.springframework.org/spring/docs/1.2.x/api/org.springframework.jdbc.core/JdbcTemplate.html>.

Таймаутам часто сопутствуют повторения. По принципу «приложим максимум усилий» программа пытается повторить операцию, время ожидания которой истекло. Повторная попытка, сделанная сразу после отказа, имеет ряд последствий, и далеко не все из них положительные. Если причиной отказа стала какая-то серьезная проблема, немедленное повторение, скорее всего, приведет к аналогичному отказу. Хотя в случае некоторых временных отказов (например, пропущенных пакетов в беспроводной WAN) подобная стратегия помогает. Тем не менее в стенах центра хранения данных причина отказа, скорее всего, кроется в неполадках на другом конце соединения. Несмотря на рекламируемые фирмой Cisco «самовосстанавливающиеся сети», мой опыт показывает, что проблемы с сетью или с другим сервером имеют обыкновение затягиваться. Поэтому повторные попытки, как правило, заканчиваются все тем же отказом.

С точки зрения клиента затянувшееся время ожидания — это плохо. Если операцию по каким-то причинам завершить невозможно, лучше вернуть результат. Это может быть отказ, успех или постановка задачи в очередь для последующего исполнения (если клиенту важно, что именно происходит). В любом случае, какой-то ответ дать требуется. Заставив клиента ждать, пока вы повторяете операцию, можно переполнить чашу его терпения.

В то же время постановка задания в очередь, позволяющая позднее повторить попытку его выполнения, — вещь хорошая, повышающая надежность системы. Представьте, что было бы, если бы электронная почта доходила до адресата только при условии, что оба почтовых сервера подключены к сети, готовы к обработке сообщений и отвечают в течение 60 секунд. Вариант с промежуточным хранением является куда более осмысленным. В случае отказа удаленного сервера очередь и повторение операций гарантируют, что после восстановления работоспособности этого сервера восстановится и вся система. Задание не должно пропадать из-за того, что часть большой системы перестала функционировать. Какая скорость в этих условиях является достаточной? Все зависит от вашего приложения и пользователей. Для веб-сайта с сервис-ориентированной архитектурой «достаточно быстро» — это в пределах 250 миллисекунд. После этого вы начинаете терять производительность и пользователей.

Таймауты естественным образом сочетаются с предохранителями. Алгоритм предохранителя способен фиксировать число превышений времени ожидания, переходя в «нерабочее» состояние, если их оказывается слишком много.

Оба паттерна — таймаутов (Timeouts) и быстрого отказа (Fail Fast) — решают проблемы, связанные с задержкой. Первый используют, когда требуется защитить систему от чужих отказов. Второй применяется, когда нужно сообщить, почему вы не можете обработать некоторые транзакции. Паттерн быстрого отказа применяется к входящим запросам, в то время как паттерн таймаутов в основном работает с исходящими. Это две стороны одной медали. В случае выдачи слишком большого набора результатов таймауты не дают клиенту обработать весь набор, хотя это и не самый эффективный подход к решению данной проблемы. Лично я считаю его всего лишь временной мерой.

Паттерн таймаутов применим к целому классу проблем. Именно установка времени ожидания помогает системам восстанавливаться после различных неожиданных событий.

ЗАПОМНИТЕ

Паттерн применяется при наличии точек интеграции, блокированных потоков и медленных ответов

Паттерн таймаутов предотвращает превращение обращений к точкам интеграции в заблокированные потоки. Это позволяет избежать каскадных отказов.

Паттерн применяется для восстановления после неожиданных отказов

Порой, когда какая-то операция выполняется слишком долго, нам не важно, почему так происходит... достаточно прервать ее и продолжить работу. Именно это нам позволяет сделать паттерн таймаутов.

Помните об отложенных повторных вызовах

В большинстве случаев причиной превышения времени ожидания являются проблемы в сети или в удаленной системе, которые невозможно решить быстро. Немедленные повторные вызовы, скорее всего, столкнутся с той же проблемой и приведут к очередному таймауту. Это только увеличит время ожидания пользователем сообщения об ошибке. В большинстве случаев имеет смысл поставить операцию в очередь и повторить ее позднее.

5.2. Предохранители



Когда в домах только появилась электрическая проводка, многие пострадали из-за незнания законов физики. Несчастные включали в цепь слишком много приборов. А каждый прибор потреблял определенное количество тока. Из-за сопротивления выделяется тепло, пропорциональное квадрату силы тока, умноженному на сопротивление (I^2R). Так как провода не обладали сверхпроводимостью, такая скрытая взаимозависимость между электрическими приборами приводила к нагреву проводов, и порой такому сильному, что это становилось причиной пожара. Пшшш... И дома нет.

Формирующаяся энергетическая промышленность нашла частичное решение проблемы резистивного нагрева в виде предохранителей. Предохранитель должен был сгореть раньше, чем загорится дом. Этот компонент был спроектирован так, чтобы быстро выходить из строя, тем самым контролируя общий режим отказа. Гениальное устройство прекрасно работало, но имело два недостатка. Во-первых, предохранители были одноразовыми и могли попросту закончиться. Во-вторых, предохранители, устанавливаемые в американских жилых домах, имели тот же самый диаметр, что и медная монетка в один цент. Эти недостатки привели к тому,

что многие начинали экспериментировать с самодельными предохранителями из монеток (то есть с медными дисками в три четверти дюйма), сопротивление которых было слишком низким. Пшш... И дома нет.

Бытовые предохранители прошли путь дисковых телефонов. Их современные версии все так же защищают от пожара дома граждан, чрезмерно любящих всякие устройства. Они работают по тому же принципу: распознают чрезмерное потребление, выходят из строя и размыкают цепь. Выражаясь более абстрактно, предохранитель существует для того, чтобы дать одной подсистеме (электрической цепи) выйти из строя (в случае чрезмерного потребления тока, например из-за короткого замыкания), не уничтожая целую систему (дом). Более того, как только опасность минует, предохранитель можно вернуть в исходное состояние, полностью восстановив функционирование системы.

Аналогичный подход можно применить к программному обеспечению, добавив к опасным операциям компонент, который позволял бы миновать вызовы, когда с системой что-то не в порядке. От повторных вызовов эта ситуация отличается тем, что предохранитель служит для прерывания операций, а не для дополнительных попыток их выполнения.

В нормальном («замкнутом») состоянии предохранителя операции выполняются в обычном режиме. Это могут быть как обращения к другой системе, так и внутренние операции, попадающие в сферу действия таймаутов или других отказов исполнения. В случае успешного вызова ничего экстраординарного не происходит. А вот в случае неудачи предохранительный механизм делает пометку об отказе. Как только число этих отказов (либо их частота, либо какой-то более сложный параметр) превышает некий порог¹, предохранитель вызывает аварийный отказ, «размыкая» цепь, как показано на рис. 16. При этом все обращения к предохранителю становятся невозможными, не делается даже попытки выполнить операцию. Через некоторое время предохранитель решает, что у операции появился шанс на успешное завершение, и переходит в «полукоткрытое» состояние. При этом следующий вызов получает возможность выполнить опасную операцию. В случае успеха предохранитель возвращается в исходное состояние, «замыкая» цепь и возвращая ее в обычный режим функционирования. В случае же неудачного вызова цепь снова оказывается разомкнутой, пока не истечет очередной таймаут.

При открытом предохранителе все вызовы немедленно обрываются. Вероятно, это должно обозначаться каким-то типом исключения. Для удобства пользователя на подобный случай имеет смысл выделить отдельное исключение. Это позволит вызывающему коду по-другому подойти к его обработке.

¹ Счетчик такого типа прекрасно реализован в алгоритме дырявого ведра (Leaky Bucket), описанном в книге *Pattern Languages of Program Design 2*, автором которой является член «банды четырех» Джон Влиссидс.

В зависимости от деталей реализации системы предохранитель может по-разному отслеживать разные типы отказов. К примеру, порог размыкания цепи в случае отказа «истечения времени ожидания вызова удаленной системы» можно установить ниже, чем при «отказе подключения».

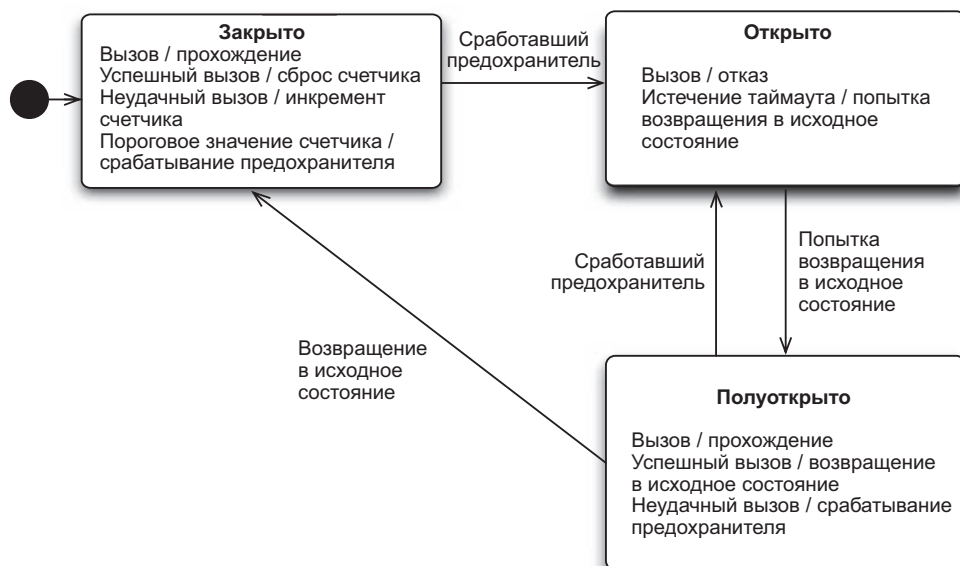


Рис. 16. Состояния предохранителя и переходы между ними

Предохранители — это средство автоматически ухудшить функциональность системы под нагрузкой, что может повлиять на коммерческий аспект. Поэтому к выбору способа обработки запросов при «открытом» предохранителе важно привлечь заказчиков системы. К примеру, должен ли приниматься заказ, если невозможно проверить наличие товаров, заказанных клиентом? Как быть при отсутствии возможности проверить кредитную карту клиента или указанный им адрес доставки? Разумеется, подобные обсуждения могут касаться не только использования предохранителей, но лучше обговорить возможность добавления предохранителя, чем требовать от клиента исчерпывающей регламентной документации.

Состояние предохранителей в системе важно еще для одной ключевой стороны: проводимых операций. Изменения этого состояния всегда должны фиксироваться, а текущее состояние должно быть доступным для запросов и мониторинга (см. главу 17). По сути, полезно вести график этих изменений во времени, так как это — ведущий индикатор проблем на предприятии в целом. Кроме того, операциям нужен способ, позволяющий непосредственно отключать предохранитель и возвращать его в рабочее состояние.

Предохранители эффективны при защите от отказов в точках интеграции, при каскадных отказах, а также при несбалансированной производительности и медленных ответах. При работе они тесно взаимодействуют с таймаутами и зачастую отслеживают ошибки, связанные с превышением времени ожидания, а не с отказами исполнения.

ЗАПОМНИТЕ

Не делайте того, что приводит к проблемам

Предохранитель является основным паттерном, защищающим систему от всех типов проблем, связанных с точками интеграции. Как только в одной из таких точек возникает проблема, отменяйте все ее вызовы.

Используйте предохранители вместе с таймаутами

Паттерн предохранителя (Circuit Breaker) позволяет избежать обращений к точке интеграции, в которой произошел отказ. Паттерн таймаутов (Timeouts) указывает на наличие отказа в точке интеграции.

Выявляйте изменения состояний, отслеживайте их и сообщайте о них

Срабатывание предохранителя всегда указывает на серьезную проблему. И операции должны быть о нем осведомлены (см. главу 17). Все срабатывания должны быть зафиксированы, классифицированы и оценены на повторяемость.

5.3. Переборки



Конструкция судна предусматривает металлические *переборки* (bulkheads), позволяющие разделить его на отдельные водонепроницаемые отсеки. После закрытия люков именно они не дают воде перемещаться из одного отсека в другой. Благодаря им единичный пробой обшивки не означает, что судно обречено. Переборки реализуют принцип изоляции повреждений.

Вы можете задействовать аналогичный прием. Разделив систему на части, вы не дадите отказу, возникшему в одной части, разрушить все. Наиболее распространенной формой переборок является физическая избыточность. Если у вас есть четыре независимых сервера, аппаратный отказ в одном из них не сможет повлиять на остальные. Аналогично, если на сервере есть два экземпляра приложения, при отказе одного из них второе продолжит свою работу (разумеется, тут не рассматривается случай, когда отказ первого приложения происходит под действием внешних факторов, которые также затрагивают второе).

Защитите важных клиентов, предоставив им отдельный пул для вызовов.

В более крупном масштабе можно реализовать критически важные службы как несколько независимых серверных ферм, часть из которых будет зарезервирована под важные приложения, в то время как остальные предоставлены обычным пользователям. Например, система продажи билетов может выделить отдельные серверы для регистрации клиентов. На эти серверы никак не будет влиять, например, тот факт, что другие серверы совместного использования переполнены запросами об откладывании рейсов (что порой случается при плохой погоде). Подобное разделение позволило бы авиакомпаниям из главы 2 продолжить регистрацию пассажиров, даже если партнеры по продажам не в состоянии найти информацию о стоимости билетов на текущие рейсы.

На рис. 17 системы Foo и Bar пользуются корпоративной службой Baz. Зависимость от общей службы делает каждую из систем чувствительной к состоянию другой. Если система Foo внезапно прекратит работу под нагрузкой, пойдет вразнос из-за какого-то дефекта или спровоцирует ошибку в службе Baz, заодно пострадает система Bar и ее пользователи. Подобные варианты неявной взаимозависимости крайне осложняют диагностирование проблем (а особенно проблем с производительностью) в системе Bar. Планирование перерывов на технологическое обслуживание службы Baz требует координации с системами Foo и Bar, а найти промежуток времени, подходящий обоим клиентам, может оказаться крайне сложно.

Если системы Foo и Bar являются критически важными и подчиняются строгому соглашению об уровне обслуживания, безопаснее разделить службу Baz на части, как показано на рис. 18.

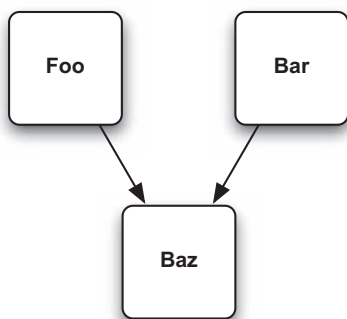


Рис. 17. Скрытые связи

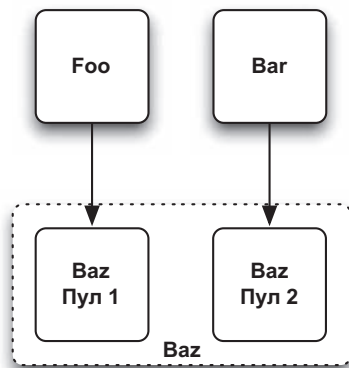


Рис. 18. Разделенная система

Выделение каждому критически важному клиенту некоторой доли вычислительной мощности избавляет от большей части скрытых связей. Скорее всего, они все равно будут совместно использовать базу данных, а значит, останется опасность взаимной блокировки между экземплярами, но это уже другой антипаттерн.

Разумеется, лучше было бы полностью сохранить функциональность. Но предполагая, что отказ неизбежно произойдет, нужно подумать, как минимизировать его последствия. Это далеко не просто, и правила на все случаи не существует. Поэтому следует проанализировать, какое влияние оказывает на бизнес потеря каждой конкретной функциональной возможности, и рассмотреть полученные данные в рамках архитектуры системы. Это позволит вам естественным образом разделить систему на части способом, который будет одновременно технически осуществимым и финансово выгодным. Границы такого разделения могут быть связаны с вызывающими сторонами, с функциональностью или с топологией системы.

ПЕРЕБОРКИ И ВЫЧИСЛИТЕЛЬНАЯ МОЩНОСТЬ

Из-за того, что каждому клиенту выделяется собственный пул серверов, служба Baz должна намного точнее оценивать спрос. А системы Foo и Bar должны быть аккуратнее со своими потребностями, так как количество доступных им ресурсов строго ограничено. При объединении серверного пула избыточный спрос от одного клиента может быть скомпенсирован излишками мощности второго. Это обеспечивает большую надежность.

Если пиковые сезоны — или даже пиковые часы — систем Foo и Bar не совпадают, мощность совместно используемого пула может быть меньше суммы индивидуальных мощностей. Именно такая эффективность является одной из основных причин продвижения сервис-ориентированной архитектуры. Как сохранить ее при добавлении переборок, повышающих безопасность?

В первую очередь, можно воспользоваться виртуализацией. Служба Baz может задействовать единственный набор физического оборудования для создания виртуальных серверов, выделенных системам Foo и Bar. После этого перераспределение мощности между клиентами сводится к административной настройке виртуальных машин. Загрузка дополнительной виртуальной машины занимает всего несколько минут. Последние версии VMware ESX позволяют автоматически перемещать виртуальные машины в зависимости от политик уровня обслуживания. Перемещение работающей виртуальной машины, как правило, занимает менее десяти секунд.

Виртуальные серверы предлагают отличные механизмы реализации переборок.

В меньшем масштабе примером разделения при помощи переборок является привязка к процессору. Привязка процесса к процессору или к группе процессоров гарантирует, что операционная система выделит потоки этого процесса только предусмотренному для этого процессору или группе. Так как привязка к процессору уменьшает потери кэша, возникающие при миграции процессов между процессорами, она часто рассматривается как параметр производительности. Если процесс выходит из-под контроля и начинает использовать все циклы процессора, он обычно выводит из строя весь компьютер. Я встречал сервер с восемью процессорами, занятыми единственным процессом. Если бы этот процесс был привязан к одному конкретному процессору, он использовал бы только его доступные циклы.

Потоки внутри одного процесса можно распределить по группам, ориентированным на различные функции¹. К примеру, часто бывает полезно зарезервировать для административных целей пул потоков, отвечающих за обработку запросов. В этом случае даже при зависании на сервере приложений всех потоков, обрабатывающих запросы, сервер все равно будет отвечать на административные запросы, такие как, например, запрос аварийного дампа для анализа или запрос на завершение работы.

Переборки эффективны в деле поддержания работоспособности службы или ее компонента, несмотря на отказы. Особенно они полезны в сервис-ориентированной архитектуре, где потеря одной службы может повлиять на работу всего предприятия. Фактически служба в SOA представляет собой единую точку отказа для системы в целом.

ПЕРЕБОРКИ И ПРОИЗВОДИТЕЛЬНОСТЬ

Привязка сервера многопоточного приложения к одному процессору может снизить его производительность, особенно если он тестировался изолированно. При наличии нескольких экземпляров серверов приложений на одном компьютере ситуация становится не столь очевидной. Если потоки каждого сервера приложений могут распределяться по всем процессорам, снизить общую производительность могут операции переключения контекста и очистки кэша. Оптимизация производительности в таких условиях превращается в многовариантную проблему: требуется сбалансировать корректное число экземпляров сервера, корректное число потоков на один сервер, объем выделенной памяти и вероятность конфликтов внутри самого приложения. Обязательно следует проверить общую пропускную способность, а не только самое быстрое время ответа отдельного сервера.

ЗАПОМНИТЕ

Сохраните часть судна

Паттерн переборок (Bulkheads) позволяет распределить вычислительную мощность с целью сохранения частичной функциональности в случае проблем.

Решите, стоит ли соглашаться на менее эффективное использование ресурсов

Когда система не подвергается опасности, разбиение серверов на группы означает, что каждая группа нуждается в большем количестве резервной вычислительной мощности. При объединении всех серверов требуется меньше общей резервной вычислительной мощности.

Выберите подходящую степень разделения

Разбивать на группы можно пулы потоков внутри приложения, процессоры на сервере или серверы внутри кластера.

¹ К примеру, до появления версии 9 приложение WebLogic позволяло создавать набор потоков для определенных транзакций путем создания нестандартных очередей. В версии 9.x это называется объектом constraint в механизме work manager.

Паттерн переборки крайне важен при совместно используемых службах

В сервис-ориентированной архитектуре от вашего приложения может зависеть множество корпоративных систем. Если из-за цепной реакции ваше приложение выйдет из строя, остановится ли работа всей компании? В случае положительного ответа на этот вопрос лучше применить несколько переборок.

5.4. Стабильное состояние



Идеографический словарь Тезаурус Роже (третье издание) предлагает следующее определение слова *теребить*: «производить некие действия от нечего делать, по неосведомленности или деструктивно». В качестве синонимов предлагаются глаголы *отрывать по пустякам*, *небрежно обращаться*, *портить неумелым обращением*. За подобным действием часто следует момент, когда человек думает: «Зачем я это сделал?!» — тот краткий промежуток времени, когда вы понимаете, что нажали не ту кнопку, остановив сервер, удалили жизненно важные данные или другим способом нарушили мир и гармонию стабильных транзакций.

Каждое прикосновение человека к серверу дает возможность для, как говорят теннисисты, невынужденных ошибок — когда игроку ничто не мешает, ситуация простая, но он ошибается¹.

Лучше всего по возможности держать людей на максимальном расстоянии от работающих систем. Когда для работы системы требуется постоянная поддержка, администраторы часто приобретают привычку не выходить из системы. Это неизбежно ведет к тому, что систему начинают теребить по всякому поводу. А ведь в идеале она должна работать без вмешательства человека.

Не поощряйте вмешательство в систему по пустякам. Системы должны работать независимо, без вмешательства человека.

Если система не рухнет каждый день (в этом случае вам следует проверить ее на наличие антипаттернов стабильности), наиболее распространенной причиной для входа в нее является очистка журналов и удаление данных.

¹ Я знал одного инженера, который, пытаясь быть полезным, увидел, что нарушена синхронизация зеркала корневого диска сервера, и дал команду восстановить зеркальную копию. К сожалению, он сделал опечатку, результатом которой стала синхронизация корректного корневого диска с новым совершенно пустым диском, только что вставленным в качестве замены вышедшему из строя. Тем самым он мгновенно удалил с сервера операционную систему.

Любой накапливающий ресурсы механизм (журналы в файловой системе, строки в базе данных или кэш в памяти) напоминает ведро из школьных задач по математике. Оно заполняется с определенной скоростью, и если с той же самой или более высокой скоростью его не освобождать, неизбежно наступит переполнение. В системах при этом происходят плохие вещи: серверы отключаются, базы данных замедляют свою работу или выдают сообщения об ошибках, время отклика растягивается до бесконечности. В соответствии с паттерном стабильного состояния (Steady State) каждый механизм, накапливающий ресурсы, должен сопровождаться другим механизмом, который пускает эти ресурсы в оборот. Мы рассмотрим несколько типов накопления и способы, позволяющие с ними справляться, не теребя постоянно систему.

Удаление данных

Принцип удаления данных кажется достаточно простым. Вычислительные ресурсы всегда конечны, соответственно бесконечно увеличивать потребление невозможно. Тем не менее в спешке развертывания новых потрясающих приложений, критически важных для решения задач компании, корректной организации удаления данных, как правило, не уделяют достаточного внимания. В принципе, лучше по возможности вообще избегать непосредственного удаления данных. Другое дело, что зачастую проектировщики мечтают о том, чтобы система хоть как-то начала работать. И мысль о том, что их творение может проработать долгий срок и накопить столько данных, что с ними придется что-то делать, кажется им «проблемой высшего порядка» — проблемой, с которой они рады были бы столкнуться.

Как бы то ни было, в один прекрасный день ваша маленькая база данных начнет расти. Достигнув подросткового возраста — это происходит примерно за два года, — она начнет капризничать и проявлять строптивость. В худшем случае она негативно повлияет на работу системы в целом (и, скорее всего, будет жаловаться, что никто ее не понимает).

Наиболее очевидным симптомом накопления данных является неуклонный рост времени ввода-вывода на серверах базы данных. Также вы можете заметить растущую задержку при постоянной нагрузке.

Удаление данных — грязная и очень ответственная работа. Ограничения целостности данных в базе — это половина дела. Может оказаться, что полностью удалить устаревшие данные, не оставив потерянных строк, крайне сложно. Вторая часть задачи сводится к необходимости гарантировать работоспособность приложения после удаления данных.

К примеру, будет ли работать приложение, если из середины коллекции удалить какие-то элементы? (Подсказка: с библиотекой Hibernate — не будет!) Как следствие, удаление данных практически всегда оставляется до момента выхода

следующей версии. Для такого подхода существует следующее малоубедительное обоснование: «У нас есть шесть месяцев на внедрение этого программного компонента». (Почему-то они постоянно говорят «шесть месяцев». Примерно как программисты, оценивающие срок выполнения задания в «две недели».)

Удаление данных должно быть реализовано в первой же версии программного обеспечения, но, как правило, этого не делается.

Разумеется, после запуска всегда появляются аварийные версии с исправлением критических недостатков или с добавлением «необходимых» программных компонентов, заказанных маркетологами, уставшими ждать завершения работы над программой. Первые шесть месяцев могут пролететь очень быстро, но во время запуска самой первой версии приходится торопиться.

УДАЛЕНИЕ ДАННЫХ НА ПРАКТИКЕ

Я выступал с докладом на OTUG¹, что в конечном итоге привело к написанию этой книги. Я был польщен, увидев среди присутствующих большую часть группы, работавшей над моим проектом, в том числе нашего спонсора. Во время рассказа о важности удаления данных и о, как правило, пренебрежительном отношении к этой операции все они кивали (с открытыми глазами!). Словом, вы можете себе представить, как я был огорчен, когда первая версия нашей программы вышла без средств удаления данных.

В конечном счете мы внедрили очень хорошо проработанный процесс удаления данных, оценив по скорости их накопления, сколько времени нам потребуется. Время почти истекло, когда мы предоставили первую итерацию процедуры удаления, позволившую ликвидировать наиболее объемные элементы. Это дало нам дополнительное время. Следующие версии были снабжены более радикальными процедурами, удалявшими менее объемные накопления.

Журналы регистрации

Журнал регистрации за прошлую неделю так же интересен, как книга, полная страховых таблиц. Их тщательное изучение может доставить удовольствие только немногим избранным. Остальные относятся к ним с такой же теплотой, как к помойке за рестораном. Еще хуже обстоят дела с журналами регистрации за прошлый месяц. В основном они только занимают ценное место на диске.

¹ Пользовательская группа, посвященная объектной технологии и собирающаяся в городе Твин-Ситис штата Миннеаполис и городе Сан-Пол в Миннесоте. См. <http://www.otug.org/>.

Но если их не контролировать, они становятся чем-то большим, чем бессмысленная груда байтов. Неограниченный рост журналов регистрации приводит к тому, что в конечном счете они заполняют файловую систему. Это может быть пространство, выделенное под журналы, корневой диск или папка для установки приложений, но в любом случае это означает неприятности. Заполняя файловую систему, журналы регистрации ставят под угрозу ее стабильность. Например, в системах семейства UNIX последние 5–10 % (точный показатель зависит от конфигурации файловой системы) пространства зарезервированы под корневой каталог. Это означает, что при заполненной на 90 или на 95 % файловой системе приложение начнет сталкиваться с ошибками ввода-вывода. При этом запущенное с правами привилегированного пользователя приложение может израсходовать последние свободные байты. В операционной системе Windows такая ситуация наблюдается с любым приложением. И в каждом подобном случае об ошибке система сообщит именно приложению. В системах на базе Java это означает исключение `java.io.IOException`, в .NET это исключение `System.IO.IOException`, для C это значение `ENOSPC` макроса `errno`. (Поднимите руки те, кто проверяет макрос `errno` на значение `ENOSPC` после каждого вызова метода `write()`?) Практически во всех случаях библиотеки регистрации самостоятельно не обрабатывают исключение, связанное с вводом-выводом. Они помещают его в оболочку или производят его трансляцию и выбрасывают в код приложения новое исключение¹.

Что произойдет потом, можно только догадываться. В лучшем случае, когда файловая система, отвечающая за регистрацию, отделена от устройств хранения данных (например, транзакций), а код приложения защищает себя достаточно хорошо, пользователь в жизни не поймет, что что-то идет не так. Значительно менее приятной, но вполне терпимой является ситуация, когда корректно сформулированное сообщение об ошибке просит пользователя набраться терпения и повторить свои действия после того, как система придет в порядок. Куда хуже ситуация, когда пользователю предъявляются результаты трассировки стека.

А однажды я сталкивался с системой, разработчик которой добавил в процесс работы сервлета «универсальный обработчик ошибок», который должен был фиксировать все виды исключений. Он был реентерабельным, поэтому если в процессе регистрации одного исключения возникало другое, он пытался зарегистрировать *оба* исключения. После заполнения файловой системы этот несчастный обработчик сошел с ума, пытаясь справиться со все разрастающимся стеком исключений. Из-за наличия нескольких потоков, каждый из которых пытался зарегистрировать собственное сизифово исключение, сервер приложений смог полностью задействовать восемь процессоров UltraSPARC III, во всяком случае, на некоторое время. Исключения, размножающиеся как кролики в задаче Фибоначчи, быстро израсходовали всю доступную память. И через некоторое время виртуальная Java-машина прекратила свою работу.

¹ Приятным исключением в этом отношении является библиотека Log4J. Для ликвидации исключений во всех «аппендерах» она использует подключаемую политику `ErrorHandler`.

Менее серьезной проблемой больших журналов является неудовлетворительное соотношение сигнал/шум. Рассмотрим журналы доступа с веб-сервера. Кроме аналитической системы WebTrends, вряд ли что-то поможет вам найти нужное значение в журналах за последний месяц. При восьми миллионах запросов, каждый из которых соответствует (в зависимости от количества ресурсов на странице) 800 000–4 000 000 просмотрам страниц, общепринятый формат регистрации Apache приводит к ежедневному появлению в журнале более 1 Гбайт новых записей. Ни один человек не сможет найти в подобном объеме данных интересующее его событие. И кстати, нет никаких причин хранить эти файлы в работающей системе. Копируйте их в промежуточную область для последующего анализа.

Не оставляйте файлы журналов в работающих системах. Для последующего анализа копируйте их в промежуточную область.

Разумеется, всегда лучше не допускать заполнения файловой системы. Конфигурация, обеспечивающая ротацию журналов, выбирается за несколько минут.

Различные трансляции системы регистрации Log4J, в том числе Log4R (Ruby) и Log4Net (любой из языков .NET), поддерживают функцию `RollingFileAppender`, конфигурация которой позволяет осуществлять ротацию файлов журналов на основе их размеров. Всегда используйте аппендер `RollingFileAppender` вместо установленного по умолчанию `FileAppender`. В пакете `java.util.logging` установленный по умолчанию аппендер `FileHandler` также можно настроить на ротацию журналов по размеру, указав в свойстве `limit` максимальное количество байтов, которое можно записывать в текущий файл. Переменная `count` задает количество сохраняемых старых файлов. Произведение значений переменных `limit` и `count` показывает, сколько места требуется под файлы журналов.

В случае унаследованного, стороннего или не использующего ни одной из прекрасных доступных сред регистрации кода в операционных системах семейства UNIX повсеместно применяется программа `logrotate`. В случае Windows эту программу можно попытаться встроить под Cygwin или вручную написать для этого сценарий `.vbs` или `.bat`. Регистрация прекрасно помогает в обеспечении прозрачности. Убедитесь, что все файлы журналов подвергаются ротации и в конечном счете удаляются, в противном случае вам рано или поздно придется осваивать инструменты восстановления системы.

ДЖО СПРАШИВАЕТ: А КАК НАСЧЕТ ЗАКОНА САРБЕЙНЗА — ОКСЛИ?

Иногда можно услышать, как люди говорят о регистрации с точки зрения требований закона Сарбейнза — Оксли (Sarbanes — Oxley act, SOX). Этот закон предъявляет высокие требования к ИТ-инфраструктуре и ИТ-операциям. Одно из этих требований состоит в том, что компания должна быть в состоянии продемонстрировать адекватный контроль любой системы, производящей финансово значимую информацию. Другими словами, если система формирования счетов вносит вклад в финансовые

отчеты фирмы, фирма должна быть в состоянии показать, что подтасовать данные этой системы невозможно.

Для большинства сайтов, ориентированных на обслуживание клиентов, подобное физически неосуществимо, но часто воспринимается как необходимость. Финансовые показатели поступают из систем управления заказами или из систем расчета по кредитным картам, а не от веб-серверов и серверов приложений. Сайт не в состоянии даже на ленточном накопителе или DVD хранить журналы веб-сервера за многие годы в соответствии с требованиями SOX. Более того, каким образом журнал доступа к веб-серверу может доказать честность финансового контроля? Для этого нужно отслеживать сеансы регистрации администратора в системе.

К сожалению, юридически вопросы далеко не всегда решаются исходя из рационального анализа вероятностей, особенно в такой размытой и плохо определенной области, как соблюдение требований SOX. Лучше всего обсудить этот вопрос с директором по информационным технологиям. (Во многих компаниях есть специальные консультанты по SOX.) Они помогут определить способ построения системы, соответствующий правовым нормам. Чем раньше вы начнете это обсуждение, тем лучше. Так как вам придется привлекать к нему юридический отдел, отдел информационных технологий и финансовый отдел, не стоит ожидать быстрого решения данного вопроса.

Кэширование в памяти

Подробно кэширование рассматривается в разделе 10.2. Для долго работающих серверов память — все равно что кислород. Оставленный без присмотра кэш способен израсходовать его весь, до последней молекулы. Недостаток памяти представляет собой угрозу как для стабильности, так и для вычислительной мощности. Соответственно при построении кэша нужно ответить на следующие вопросы:

- ☐ Конечно или бесконечно пространство возможных ключей?
- ☐ Меняются ли когда-нибудь кэшированные элементы?

Если верхней границы возможных ключей не существует, размер кэша следует ограничить вручную. Если пространство ключей не ограничено, а ключи не статичны, требуется механизм, объявляющий кэш недействительным. Простейшим примером такого механизма служит привязанная ко времени очистка кэша. Вы также можете задействовать алгоритм вытеснения давно не используемых объектов (Least Recently Used, LRU) или модель рабочего набора, но в девяти случаях из десяти периодической очистки кэша будет вполне достаточно.

Некорректное кэширование является основной причиной утечек памяти, что, в свою очередь, приводит к таким ужасам, как ежедневная перезагрузка серверов. Ничто так не способствует выработке у администраторов привычки никогда не выходить из системы, как ежедневные или еженочные авралы. Накопление ненужных данных является основной причиной медленной реакции системы, а стабильное состояние помогает избежать этого антипаттерна. К тому же

стабильное состояние способствует упрочению производственной дисциплины, так как системному администратору не приходится то и дело регистрироваться на работающих серверах.

ЗАПОМНИТЕ

Старайтесь лишний раз не теребить систему

Человеческое вмешательство приводит к проблемам. Сделайте так, чтобы его необходимость исчезла. Ваша система должна проработать хотя бы в течение обычного цикла внедрения без очистки диска вручную и еженощных перезагрузок.

Удаляйте данные средствами приложений

Администраторы баз данных могут писать сценарии удаления данных, но они не всегда знают, как на эту операцию отреагирует приложение. Для поддержания логической целостности, особенно при использовании утилиты ORM, приложение должно удалять данные самостоятельно.

Ограничьте кэширование

Кэширование в память ускоряет работу приложений, пока не начнет замедлять ее. Ограничьте количество потребляемой кэшем памяти.

Убирайте лишние журналы

Не стоит хранить бесконечное количество файлов журналов. Настройте их ротацию в зависимости от размера. Если правовые нормы требуют их сохранения, используйте для этого отдельный сервер.

5.5. Быстрый отказ



Если медленная реакция хуже, чем ее отсутствие, то медленная реакция *на отказ* еще хуже. Это все равно что отстоять бесконечную очередь в автотранспортной инспекции только затем, чтобы узнать, что заполнять нужно было другой бланк, и вернуться в конец очереди. Может ли существовать более бесполезная трата системных ресурсов, чем сжигание циклов и тактов процессора с последующим выбрасыванием результата в мусорную корзину?

Если система может заранее определить, что некая операция приведет к отказу, всегда лучше, если этот отказ случится быстро. В результате вызывающая сторона не будет тратить вычислительную мощность на ожидание, а сможет заняться другой работой.

Как система может определить, случится ли отказ? Какое секретное эвристическое правило я собираюсь вам поведать? Неужели для приложений существует эквивалент придуманному в Intel прогнозированию уровня переходов?

На самом деле все куда прозаичнее. Существует большой класс отказов, связанный с недоступностью ресурсов. Например, когда распределитель нагрузки получает

запрос на подключение, но ни один из серверов в его сервисном пуле не функционирует, он должен немедленно отклонить запрос. Некоторые конфигурации на некоторое время ставят запрос на подключение в очередь в надежде, что какой-то из серверов быстро станет доступным. Это нарушает паттерн быстрого отказа (Fail Fast).

В любой сервис-ориентированной архитектуре приложение по запросу на обслуживание может примерно понять, какие соединения с базой данных и внешние точки интеграции ему потребуются. Служба может быстро выгрузить необходимые ей соединения, проверить состояние предохранителей у точек интеграции и приказать диспетчеру транзакций начать транзакцию. Аналогичным образом к работе готовятся повара в ресторанах — собирая все ингредиенты, которые могут понадобиться для выполнения будущих заказов. Если какой-то ресурс недоступен, лучше прервать операцию немедленно, а не после того, как половина работы будет сделана.

Проверяйте доступность ресурсов перед началом транзакции.

ВСЕ ЧЕРНОЕ

Один из моих наиболее интересных проектов выполнялся для фирмы, занимающейся студийной фотографией. Часть проекта включала работу над программным обеспечением, которое визуализировало изображения для печати с высоким разрешением. В предыдущей версии этой программы была ошибка: при недоступных цветовых профилях, изображениях, фонах или масках альфа-канала «визуализировалось» черное изображение, состоящее из пикселей с нулевыми значениями. Такое изображение распечатывалось, впуская расходуя бумагу, реактивы и время. Браковщики отправляли его в начало рабочей цепочки для диагностики, отладки и коррекции. В конечном счете проблему решили (вызовом разработчика), а плохой отпечаток переделали. Так как сроки сдачи поджимали, стандартный рабочий процесс был прерван ради срочного выполнения этого заказа.

Моя группа применила к этой программе паттерн быстрого отказа. При появлении задания по выводу на печать визуализатор начинал проверять наличие всех шрифтов (отсутствующие шрифты тоже приводили к необходимости переделки заказа, но не по причине черного изображения), изображения, фона и масок альфа-канала. Память для всех этих операций выделялась заранее, что исключало отказ в процессе ее выделения. Визуализатор немедленно отчитывался управляющей системе о любом обнаруженном дефекте, не давая ей впустую тратить время на обсчет. И, что лучше всего, «дефектные» заказы изымались из рабочего конвейера, что исключало появление в конце процесса наполовину сделанных версий. После запуска нового визуализатора количество переделок, вызванных программным обеспечением¹, упало до нуля.

Правда, мы не организовали для финального изображения предварительного выделения места на диске. Эти наши действия противоречили паттерну стабильного состояния, но

¹ Заказы все равно могли отправляться на переделку из-за других проблем с качеством выполнения: пыль на объективе камеры, плохая экспозиция, некорректное кадрирование и т. п.

заказчик утверждал, что у них есть свой абсолютно надежный механизм очистки данных. Потом оказалось, что «механизмом очистки» был один из сотрудников, который время от времени удалял кучу файлов. Меньше чем через год после нашего запуска места на жестких дисках почти не осталось. Как и следовало ожидать, единственное место, где мы отступили от принципа быстрого отказа, было тем самым местом, об отказе которого визуализатор не отправил отчет. В результате он визуализировал изображения — потратив на это несколько минут — и затем выбросил исключение `IOException` в файле журнала.

Другой способ организации быстрого отказа в веб-приложении заключается в базовой проверке параметров сервлетом, получившим запрос, или контроллером до загрузки EJB или доменных объектов. При этом следует быть осторожным, чтобы не нарушить инкапсуляцию последних. Если проверяется не просто значение `null/не-null` или форматирование чисел, проверку следует производить в доменных объектах или в интерфейсе приложения.

Даже в рамках паттерна быстрого отказа следует по-разному оповещать о системном отказе (недоступные ресурсы) и отказе приложения (ошибка в параметре или недопустимое состояние). Обобщенное «сообщение об ошибке» может привести к срабатыванию предохранителя в вышестоящей системе только из-за ввода пользователем некорректных данных или несколько раз подряд нажатой кнопки перезагрузки страницы.

Паттерн быстрого отказа повышает общую стабильность системы, не допуская медленных ответов. В комплексе с таймаутами быстрые отказы могут предотвратить надвигающиеся каскадные отказы. Также они помогают поддерживать вычислительную мощность системы, когда она находится под нагрузкой из-за частичных отказов.

ЗАПОМНИТЕ

Избегайте медленных ответов и быстро прекращайте работу

Если система не соответствует требованиям SLA, сразу же информируйте вызывающую сторону. Не заставляйте ее ждать сообщения об ошибке или истечения таймаута. Это всего лишь превращает вашу проблему в проблему вызывающей стороны.

Заранее резервируйте ресурсы и проверяйте точки интеграции

Чтобы избежать бесполезной работы, проверяйте возможность завершения транзакции до ее начала. Если важные ресурсы недоступны: например, в части системы, к которой требуется обратиться, сработал предохранитель, — не тратьте время на выполнение действий с этими ресурсами. Шансы, что между началом и серединой транзакции ситуация изменится, крайне малы.

Проверяйте вводимые данные

Еще до резервирования ресурсов выполняйте базовую проверку вводимых пользователем данных. Нет смысла тратить время на установление соединения с базой данных, выборку доменных объектов, их заполнение и вызов метода `validate()`, чтобы в результате обнаружить отсутствие нужного параметра.

5.6. Квитирование



Термин *квитирование* (handshaking) относится к передаче от одного устройства к другому сигнала, удостоверяющего наличие связи между этими устройствами. Протоколы последовательной передачи данных, например RS-232 (в настоящее время EIA-232C), полагаются на то, что приемник сообщит о своей готовности к приему информации. Аналоговые модемы использовали свой вариант квитирования для согласования скорости и кодирования передаваемых сигналов. Кроме того, как было показано ранее, протокол TCP реализует трехстороннее квитирование для установления соединений через сокет. При TCP-квитировании приемник также может отправить передатчику сигнал, останавливающий пересылку данных до момента, пока приемник не сообщит о своей готовности. Квитирование повсеместно используется в низкоуровневых протоколах взаимодействия, но практически не применяется на уровне приложений.

Грустная правда состоит в том, что протокол HTTP не очень хорошо выполняет квитирование. Протоколы на базе HTTP, такие как XML-RPC или WS-I Basic, имеют несколько вариантов, в которых этот процесс доступен. Протокол HTTP предоставляет код ответа, «503 сервис недоступен», указывающий на временное состояние¹. Но большинство клиентов коды состояний не различают. Если код отличается от «200 ОК»², «403 доступ запрещен» или «302 найдено (перенаправление)», клиент, скорее всего, воспримет ответ как критическую ошибку.

Аналогично плохо сигнализируют о своей готовности к работе протоколы, связанные с CORBA, DCOM и Java RMI.

Квитирование представляет собой попытку сервера защитить себя путем регулирования рабочей нагрузки. Чтобы не пасть жертвой поступающих запросов, сервер должен уметь отказаться от предлагаемой работы. Ближайшая аппроксимация, которой мне удалось достичь с серверами на базе HTTP, опиралась на взаимодействие между распределителем нагрузки и веб-серверами или серверами приложений. Веб-сервер уведомлял распределитель — который периодически проверял «работоспособность» его страниц — о своей занятости, возвращая либо страницу ошибки (код состояния HTTP 503 «недоступен» тут вполне подходит), либо HTML-страницу с сообщением об ошибке. После этого распределитель знал, что на этот конкретный веб-сервер посылать дополнительные задания не следует.

Конечно, этот вариант работает только для веб-служб и не срабатывает, когда все веб-серверы слишком заняты, чтобы обслужить еще одну страницу.

¹ См. <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html> и https://ru.wikipedia.org/wiki/Список_кодов_состояния_HTTP.

² Многие клиенты воспринимают как ошибку даже остальные коды серии 200.

В сервис-ориентированной архитектуре сервер может выполнять запрос «проверки здоровья» средствами клиента. В этом случае перед тем, как отправить запрос, клиент проверяет, в порядке ли сервер. Такой подход обеспечивает хорошее квитирование за счет удваивания числа соединений и запросов, которые приходится обрабатывать серверу. Большая часть времени при типичном обращении к веб-службе тратится на установку и разрыв TCP-соединения, поэтому проверка работоспособности перед отправкой реального запроса всего лишь удваивает непроизводительные затраты на соединение.

Квитирование становится особенно ценным, когда несбалансированные мощности ведут к медленной реакции системы. Если сервер способен определить, что он не в состоянии гарантировать выполнение SLA, он должен иметь возможность как-то попросить вызывающую сторону не посылать запрос. Находящийся за распределителем нагрузки сервер пользуется бинарным (вкл./выкл.) средством прекращения запросов, что, в свою очередь, приводит к удалению не отвечающего сервера из пула. Но это очень грубый механизм. Намного лучше встроить квитирование во все применяемые вами нестандартные протоколы.

Временным решением, к которому можно прибегнуть при обращении к службам, не допускающим квитирования, являются предохранители. В этом случае вместо того, чтобы вежливо спрашивать сервер, может ли он принять ваш запрос, вы просто отправляете вызов и следите, как все работает.

В целом квитирование относится к недостаточно используемым приемам, которые с большой пользой могли бы применяться в протоколах прикладного уровня. Это эффективный способ остановки распространения трещин по слоям в случае каскадного отказа.

ЗАПОМНИТЕ

Организуйте согласованное управление нагрузкой

Квитирование между клиентом и сервером позволяет отрегулировать нагрузку до приемлемого уровня. Этот механизм следует встроить как в клиент, так и в сервер. Большинство распространенных протоколов прикладного уровня, например HTTP, JRMPP, IIOP и DCOM, квитирование не поддерживает.

Помните о проверках работоспособности

Запросы, проверяющие работоспособность сервера, являются средством на прикладном уровне обойти тот факт, что протоколы не поддерживают квитирование. Их имеет смысл использовать в случаях, когда издержки на дополнительный вызов меньше издержек на вызов, завершающийся неудачей.

Встраивайте квитирование в собственные низкоуровневые протоколы

Если вы создаете собственные протоколы для сокетов, встраивайте в них механизм квитирования, чтобы конечные точки могли информировать друг друга о готовности к работе.

5.7. Тестовая программа



Как вы уже видели в предыдущих главах, распределенные системы обладают такими режимами отказа, которые чрезвычайно трудно воспроизвести в средах разработки и тестирования. Для более детальной проверки различных компонентов в комплексе часто прибегают к среде «интеграционного тестирования». В этой среде наша система полностью интегрируется со всеми системами, с которыми она должна взаимодействовать.

Но такое тестирование само по себе представляет проблему. Какие версии следует для этого взять? С одной стороны, кажется, что для тестирования следует взять версии зависимостей, которые активно использовались на момент выхода нашей системы. Но при помощи теории множеств можно математически доказать, что такой подход ограничивает *всю компанию* тестированием всего одного нового фрагмента ПО за один раз. Я мог бы это сделать, но оставляю это читателям в качестве упражнения. Кроме того, зависимость современных систем друг от друга создает такую тесно переплетенную сеть, что интеграционное тестирование становится по-настоящему унитарным — превращается в единый глобальный тест, охватывающий действующие системы целого предприятия. Такой унитарной среде требуется такой же строгий — или даже более строгий — контроль изменений, чем реальным производственным средам.

Существуют и более абстрактные сложности. (Вы можете спросить: «Более абстрактные, чем теория множеств?») Среда интеграционного тестирования в состоянии проверить действия системы при корректной работе всех ее зависимостей. Конечно, удаленную систему можно заставить вернуть ошибку, но все равно она будет работать более-менее в пределах заданных нормативов. Если в нормативно-технической документации сказано: «Система возвращает код ошибки 14916, если в запрос не включена дата последней антивирусной проверки телефона», вызывающая сторона без проблем может воспроизвести приводящие к ошибке условия. Но при этом удаленная система все равно будет работать в границах своей спецификации.

Однако основная идея этой книги состоит в том, что каждая система рано или поздно может выйти за эти границы; и поэтому жизненно важно протестировать поведение локальной системы при неполадках в удаленной системе. И если проектировщики удаленной системы не встроили в нее режимы, имитирующие весь спектр выходящих за пределы спецификации отказов, которые могут естественным образом возникнуть в процессе эксплуатации, значит, она будет демонстрировать варианты поведения, недоступные для проверки средствами интеграционного тестирования.

Куда лучше такой подход к интеграционному тестированию, который позволяет проверить все эти режимы отказов. Он должен сохранять или усиливать изоляцию систем, чтобы избежать привязки проблемы к конкретной версии и допустить тестирование в разных точках, а не в упомянутой унитарной тестовой среде, охватывающей все предприятие.

Для этого можно создать тестовую систему, эмулирующую удаленную систему с другой стороны от точки интеграции. Специалисты по аппаратному обеспечению

и инженеры-проектировщики уже давно используют такие системы. Разработчики ПО также их применяют, но не настолько интенсивно, как хотелось бы. Хорошая тестовая система должна быть изоциренной. Она должна вести себя так же жестко, как настоящая, и оставлять на тестируемой системе шрамы, чтобы выработать у последней скептицизм.

Представьте, что вы создаете тестовую систему, имитирующую удаленную часть, к которой обращаются все вызовы веб-служб. Так как все удаленные вызовы осуществляются по сети, подключение через сокет будет подвержено следующим отказам:

- ☐ соединение может быть отклонено;
- ☐ соединение может слушать очередь, пока у вызывающей стороны не истечет время ожидания;
- ☐ удаленная сторона может ответить сообщением с флагом SYN/ACK и больше не отправлять никаких данных;
- ☐ удаленная сторона может отправлять только пакеты RESET;
- ☐ удаленная сторона может отчитаться, что буфер приема переполнен, но данные не выгружаются;
- ☐ после установки соединения удаленная сторона может не прислать ни одного байта данных;
- ☐ после установки соединения могут теряться пакеты, что приводит к задержкам из-за повторной передачи;
- ☐ после установки соединения удаленная сторона может не признаваться в получении пакетов, вызывая бесконечные повторные передачи;
- ☐ служба может принять запрос, отправить заголовки ответа (например, HTTP), но тело ответа так и не придет;
- ☐ служба может пересылать в качестве ответа по одному байту каждые тридцать секунд;
- ☐ служба может отправить ответ в формате HTML вместо ожидаемого XML;
- ☐ служба может послать вам мегабайты данных, в то время как ожидаются только килобайты;
- ☐ служба может отвергнуть все учетные данные как не прошедшие проверку подлинности.

Эти отказы относятся к разным категориям: проблемы с сетевым транспортом, проблемы с сетевыми протоколами, проблемы с прикладными протоколами, проблемы с логической схемой приложения. Небольшое умственное упражнение поможет вам обнаружить режимы отказа на каждом из семи уровней модели OSI. Добавлять к приложению переключатели и флажки, позволяющие имитировать все эти отказы, — задача дорогостоящая и нетривиальная. Кому нужен риск случайного включения «имитации отказа» после запуска системы в эксплуатацию? Среда

интеграционного тестирования хорошо подходит только для проверки отказов на седьмом (прикладном) уровне модели OSI, а не на всех уровнях.

ДЖО СПРАШИВАЕТ: А ЧЕМ ПЛОХИ ФИКТИВНЫЕ ОБЪЕКТЫ?

Фиктивные объекты широко применяются в модульном тестировании¹. Такой объект предоставляет используемую тестируемым объектом альтернативную реализацию, которой может управлять модульный тест. Предположим, приложение задействует объект `DataGateway` в качестве внешнего слоя в целом для уровня долговременного хранения. Реальная реализация этого объекта будет иметь дело с параметрами подключения, сервером базы данных и набором тестовых данных. Это слишком сильная для одного теста взаимозависимость, зачастую приводящая к невоспроизводимым результатам тестирования или скрытым зависимостям между тестами. Фиктивный объект улучшает изоляцию модульного теста, убирая все внешние подключения. Такие объекты часто используются как границы между слоями.

Некоторые фиктивные объекты можно заставить выбрасывать исключения, когда тестируемый объект вызывает их методы. Это позволяет модульным тестам имитировать некоторые виды отказов, в частности те, которым соответствуют исключения (при условии, что базовый код в реальной реализации будет генерировать исключения).

Фиктивные объекты в отличие от тестовой системы можно настроить на воспроизводство поведения, соответствующего только определенному интерфейсу. Тестовая система запускается на отдельном сервере и поэтому не обязана согласовываться с какими бы то ни было интерфейсами. Она может провоцировать сетевые ошибки, ошибки протоколов и ошибки прикладного уровня. Если бы все низкоуровневые ошибки можно было гарантированно распознать, перехватить и представить как корректный тип исключения, в тестовых системах просто не было бы нужды.

Тестовая система «знает», что предназначена для тестирования; никаких других функций она не выполняет. В то время как реальное приложение никогда напрямую не обращается к низкоуровневым сетевым прикладным программным интерфейсам, тестовая система это делает. Поэтому она в состоянии посылать байты как слишком быстро, так и очень медленно. Она может установить чрезмерно глубокую очередь слушания. Она может подключиться к сокету и не обслужить ни одной попытки соединения. Тестовая система должна действовать как маленький хакер, пытающийся любыми способами вывести из строя вызывающие стороны.

Пусть ваша тестовая система действует как маленький хакер.

Многие виды плохого поведения будут соответствовать различным приложениям и протоколам. К примеру, отказ в подключении, медленное подключение и отсутствие ответа на принимаемые запросы характерны для любых сокетных протоколов: HTTP, RMI или RPC. В данном случае одна тестовая система может симитировать

¹ См. <http://www.junit.org>.

много вариантов плохого сетевого поведения. Мне нравится следующий трюк: сопоставлять различным вариантам плохого поведения разные номера портов. На порту 10200 соединение принимается, но ответ никогда не посылается. Порт 10201 получает соединение и отвечает, но ответ копируется из `/dev/random`. Порт 10202 открывает соединение и немедленно разрывает его. И так далее. Такой подход избавляет меня от необходимости менять режимы тестовой системы, а одна система может вызвать отказ многих приложений. Он даже помогает с функциональным тестированием в среде разработки, позволяя многочисленным разработчикам подключаться к тестовой системе со своих рабочих станций. (Разумеется, это также позволяет разработчикам запускать собственные экземпляры сбоев тестовых систем.)

Имейте в виду, что тестовая система может отлично выводить приложения из строя и даже убивать их. Поэтому имеет смысл вести журнал запросов тестовой системы на случай, если ваше приложение без звука умрет, никак не указав на то, что его убило.

Тестовая система может быть спроектирована как сервер приложений; она может иметь подключаемое поведение для тестов, связанных с реальным приложением. Единая среда для тестовой системы допускает разделение на подклассы, реализующие любой протокол прикладного уровня и любые отклонения от него, которые могут потребоваться.

ЗАПОМНИТЕ

Эмулируйте отказы, не указанные в спецификации

Обращение к реальному приложению позволяет протестировать только те ошибки, которые это приложение в состоянии сознательно произвести. Хорошая тестовая система дает возможность симитировать все виды сложных реальных режимов отказа.

Нагружайте вызывающую сторону

Тестовая система может сформировать ситуацию медленного ответа, отсутствия ответа или бессмысленного ответа. После этого остается посмотреть, как на это среагирует приложение.

Для распространенных отказов применяйте обобщенные системы

Вовсе не обязательно строить для каждой точки интеграции отдельную тестовую систему. Сервер-«убийца» может слушать несколько портов, воспроизводя различные режимы отказа в зависимости от того, к какому порту делается подключение.

Дополняйте, а не заменяйте другие методы тестирования

Паттерн тестовой программы (Test Harness) призван усилить другие методы тестирования. Он не заменит модульного тестирования, приемочного тестирования, среды автоматизированного тестирования и пр. Каждый из этих механизмов позволяет проверить функциональное поведение. Тестовая система проверяет «нефункциональное» поведение, поддерживая изоляцию от удаленных систем.

5.8. Разделение связующего ПО



Благозвучным термином *связующее программное обеспечение* (middleware) называют утилиты, населяющие самое запутанное пространство — место объединения систем, которые не предназначены для совместной работы. Под названием *интеграция приложений предприятия* связующее ПО несколько лет в начале 2000-х было ходовым товаром, а затем постепенно ушло со сцены. Связующее ПО заполняет существенные пустоты между корпоративными системами. Это соединительная ткань, перекидывающая мосты между отдельными островками автоматизации. (Не звучит ли это оксюмороном?)

Часто сравниваемое с «канализацией» — во всех смыслах — связующее программное обеспечение всегда остается по своей сути запутанным, так как предназначено для работы с разными бизнес-процессами, разными технологиями и даже разными определениями одного и того же логического понятия. Подобная «непривлекательность» до некоторой степени объясняет, почему сервис-ориентированная архитектура в настоящее время отвлекает ваше внимание от менее эффективной, но более нужной работы связующего ПО.

Хорошо сделанное связующее ПО одновременно интегрирует и разъединяет системы. Интеграция осуществляется путем передачи данных и событий между системами. Но при этом связующее ПО позволяет участвующим системам обращаться друг к другу, не зная точных характеристик. Так как основной причиной нестабильности являются именно точки интеграции, такое положение дел кажется привлекательным.

Любой вид синхронного вызова-ответа или метода запроса-ответа заставляет вызывающую систему остановить свои действия и заняться ожиданием. В этой модели вызывающая и принимающая системы должны быть активны одновременно — они синхронны во времени, — даже если они разнесены в пространстве. В эту категорию попадают вызовы удаленных процедур (RPC), технологии HTTP, XML-RPC, RMI, CORBA, DCOM и все остальные аналоги вызовов локальных методов. Тесно связанное связующее программное обеспечение усиливает воздействие на систему. Синхронные вызовы относятся к особенно сильным усилителям, стимулирующим каскадные отказы.

Менее тесно связанные формы связующего программного обеспечения позволяют вызывающей и принимающей системам обрабатывать сообщения в разных местах и в разное время. В эту категорию попадает старая добрая система управления очередями сообщений IBM MQseries, как и любая другая система передачи сообщений, работающая по принципу публикации/подписки, а также системы обмена сообщениями по протоколу SMTP или SMS. (Последними двумя протоколами часто пользуются информационные посредники, сделанные из органических компонентов, а не из кремния. Задержка при этом тоже, как правило, велика.)

Рисунок 19 демонстрирует диапазон взаимозависимостей различных технологий связующего программного обеспечения.

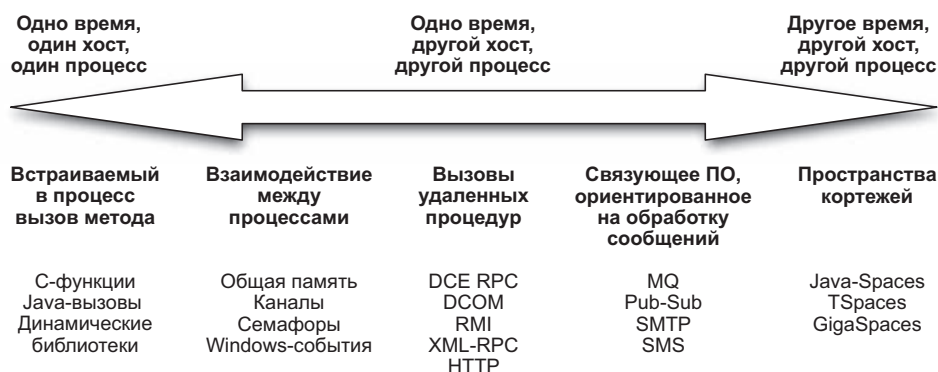


Рис. 19. Спектр взаимозависимостей связующего программного обеспечения

Связующее ПО, ориентированное на обработку сообщений, разделяет конечные точки как в пространстве, так и во времени. Так как посылающая запрос система не замирает в ожидании ответа, эта форма связующего ПО не способна вызвать каскадный отказ.

Основным преимуществом синхронного (жестко связанного) связующего программного обеспечения является его логическая простота. Предположим, планируемая клиентом покупка по кредитной карте должна быть авторизована. Если авторизация реализована через вызов удаленной процедуры или XML-RPC, приложение может однозначно решить, стоит ему перейти к этапу оформления заказа или отправить пользователя обратно на страницу с выбором метода платежа. Для сравнения, если система отправляет сообщение с просьбой об авторизации кредитной карты и не ждет на него ответа, она должна каким-то образом решить, что делать, если запрос авторизации будет отклонен или, что еще хуже, останется без ответа.

Разработка асинхронных процессов по своей природе намного сложнее. Процессу приходится сталкиваться с очередями исключений, запоздалыми реакциями, обратными вызовами (как компьютера к компьютеру, так и человека к человеку) и допущениями. К принятию этих решений следует привлекать и спонсоров вызывающей системы, которые в конечном счете должны сказать, какой уровень финансовых рисков для них приемлем.

Большинство описанных в этой главе паттернов можно применять без особого влияния на стоимость реализации системы. Однако решения, связанные со связующим ПО, в эту категорию не попадают. Во-первых, это дорогая продукция. Во-вторых, различные стили реализации связующего программного обеспечения влекут за собой разные варианты проектов. В итоге попытка что-то поменять стоит очень дорого. Переход от RPC-модели запроса-ответа по отношению к пространству кортежей требует других представлений на всех уровнях.

Ну и наконец, связующее программное обеспечение часто закупается на уровне предприятия, то есть решение зачастую принимается за вас еще до того, как вы начали работу над архитектурой или структурой проекта.

ЗАПОМНИТЕ

Принимайте решение в последний ответственный момент

Другие паттерны стабильности допускают реализацию без масштабных изменений проекта и архитектуры программы. Решение о разделении связующего программного обеспечения связано с архитектурой. Оно влияет на все части системы. Это одно из тех необратимых решений, которые лучше применять на ранней стадии.

Избавляйтесь от многих режимов отказа путем тотального разделения

Чем более полно вы разделите отдельные серверы, слои и приложения, тем меньше проблем с точками интеграции, каскадными отказами, медленными ответами и блокированными потоками вас ждет. Вы обнаружите, что разделенные приложения легче поддаются адаптации, так как все члены системы настраиваются независимо друг от друга.

Изучите множество архитектур и выберите лучший вариант

Далеко не каждая система должна представлять собой трехуровневое приложение с базой данных Oracle. Изучите разные стили архитектуры и выберите наиболее подходящий под конкретную задачу.

6 Заключение по теме стабильности

Со временем возникают даже самые невероятные комбинации. Если вы когда-нибудь говорили: «Шансы на это исчезающе малы», рассмотрите такую ситуацию: десять миллионов просмотров страницы в день за три года (предполагаем, что на странице находится пятьдесят ресурсов) дают вам 547 500 000 000 шансов, что что-то пойдет не так. Более пяти сотен миллиардов возможностей отказа. Последние астрономические наблюдения показали, что галактика Млечный путь насчитывает четыре сотни миллиардов звезд. Астрономически невероятные совпадения случаются ежедневно.

Астрономически невероятные совпадения случаются ежедневно.

Отказы неизбежны. И в наших системах, и в системах, от которых они зависят, будут возникать как серьезные, так и несерьезные отказы. Антипаттерны стабильности усугубляют временные события. Они ускоряют распространение трещин в системе. Устранение антипаттернов не помешает возникновению плохих вещей, но минимизирует причиняемый ими вред.

Разумное применение паттернов стабильности оставляет программное обеспечение работоспособным, что бы ни происходило. Ключом к успешному их применению является здравый смысл. Внимательно изучайте требования, предъявляемые программным обеспечением. Смотрите на другие промышленные системы с подозрением и недоверием — любая из них может нанести вам удар в спину. Распознавайте

угрозы и применяйте соответствующие паттерны стабильности. Паранойя — это всего лишь правильное мышление.

Остаться на плаву — больше чем полдела. Рассмотрите все обстоятельства, мешающие вашей системе. Примерно половина всех проектов аннулируется на стадии разработки. Из выживших половина внедряется с опозданием, не укладывается в бюджет или не соответствует требованиям. Из оставшихся проектов (менее 25 % всех проектов), которые вводятся в эксплуатацию, большинство приносит убытки из-за простоев, потерянной выгоды и затрат на обслуживание.

Раз вы читаете эту книгу, значит, вы не входите в число тех, кому не повезло. Вам есть чем гордиться!

Как ни грустно признавать, но отсутствия проблем, как правило, не замечают. Вот если бы вы спасли неудачную реализацию, был бы шанс выглядеть героем. А если вы как следует потрудились и разработали систему, которая остается стабильной с самого начала, скорее всего, на отсутствие простоев никто не обратит внимания. Вот такие дела. Создайте неразрушимую систему, и пользователи гарантированно начнут жаловаться на что-то еще. Просто они так устроены.

Вообще говоря, если система никогда не останавливается, пользователи, как правило, жалуются на то, что она медленно работает. Поэтому далее мы поговорим о вычислительной мощности и производительности, а также о том, как извлечь из имеющихся ресурсов максимум.

Часть II

Вычислительная МОЩНОСТЬ

7 Затоптаны клиентами

7.1. Обратный отсчет и запуск

После нескольких лет работы наконец наступил день запуска. Я присоединился к этой огромной рабочей группе (всего в ней было более трех сотен человек) за девять месяцев до этого момента, чтобы помочь в разработке нового интернет-магазина, системы управления контентом, системы обслуживания клиентов и системы обработки заказов. Обреченный в течение следующих семи лет быть опорой компании, я присоединился к коллективу с опозданием более чем на год. В течение последующих девяти месяцев я работал на износ: ел прямо за рабочим столом и ложился спать только поздно ночью. Зима в Миннесоте даже в лучшие дни является суровым испытанием для нервов. Рассвет наступает поздно, а ночь приходит рано. Все мы месяцами не видели солнца. Часто казалось, что мы попали в ночной кошмар Оруэлла. Нам нужно было дотянуть до весны — единственного времени года, когда в этой местности стоит жить. Однажды зимней ночью я заснул, а проснувшись, обнаружил, что пришло лето.

Даже спустя девять месяцев я оставался одним из новеньких. Часть группы разработчиков вкалывала тут уже больше года. Каждый день они ели то, что приносили им клиенты. Даже сейчас некоторые из них дрожат, вспоминая блинчики тако с индейкой.

Но сегодня был день триумфа. Каторжный труд и разочарования, забытые друзья и разводы — все это должно было стать частью прошлого, как только мы запустим систему.

Отдел маркетинга, многих сотрудников которого последний раз видели пару лет назад на совещаниях, посвященных формулировке требований, собрался в большой

комнате для конференций, чтобы провести церемонию запуска с последующим распитием шампанского. Технологи, воплотившие свои смутные и расплывчатые мечты в реальность, толпились у стены с ноутбуками и мониторами, которые должны были показывать состояние сайта.

В 9 утра руководитель проекта нажал большую красную кнопку. (У него реально была большая красная кнопка, соединенная с лампочкой в соседней комнате, где техник щелкал на кнопке перезагрузки браузера, проецируемого на большой экран.) Как по волшебству на экране в большом конференц-зале появился новый сайт. Из своего укрытия в другом конце этажа мы слышали, как маркетологи разразились одобрительными криками. Послышались звуки открываемого шампанского. Сайт запустили в эксплуатацию.

Разумеется, реальные изменения были вызваны сетью доставки контента¹. Там было запланировано обновление метаданных, которые должны были распространиться по сети в 9 утра по центральному времени. Вступление изменений в силу во всей сети серверов занимало около восьми минут. Роста трафика на новых серверах мы ожидали примерно с 9:05 утра. Браузер в конференц-зале был настроен на обход CDN и непосредственное обращение к сайту, то есть прямо к тому, что в CDN называется «исходными серверами». Маркетологи — не единственные, кто знает, как пустить пыль в глаза. По сути, мы могли немедленно видеть новый трафик, поступающий на сайт.

Сеть доставки контента (Content Delivery Network, CDN) известна также как «пограничная сеть». Это ускоритель, кэширующий изображения и статический контент перед браузером, что сокращает до 80 % запросов к веб-серверам сайта.

К 9:05 на серверах было уже 10 000 активных сеансов.

К 9:10 количество сеансов на сайте перевалило за 50 000.

К 9:30 мы имели 250 000 сеансов, после чего сайт рухнул.

7.2. Цель — пройти тест качества

Чтобы понять, что стало причиной столь катастрофического и быстрого падения сайта, нужно вернуться на три года назад.

¹ На самом деле CDN позволила миру взглянуть на новый сайт еще в воскресенье, до нашего понедельничного запуска. Каким-то образом изменения метаданных были сделаны некорректно, и переключение сервера-источника произошло в воскресенье днем. С момента, когда человек, управляющий клиентом, заметил, что сайт видим (и принимает заказы!) с частично загруженным контентом, и до момента, когда мы установили, что корень проблемы лежит в CDN, прошел почти час. То есть нам потребовался еще час, чтобы откатить изменения назад и распространить их по CDN.

В наши дни по ряду веских причин проекты, начинаемые «с нуля», встречаются редко. Прежде всего, такой вещи, как проект веб-сайта, не существует. На самом деле в таких случаях речь идет о проектах интеграции предприятия с HTML-интерфейсом. В большинстве проектов присутствует по меньшей мере некая внутренняя часть, с которой мы должны интегрироваться.

Любой проект сайта на самом деле является проектом производственной интеграции.

Может показаться, что одновременная разработка внутренней и входной частей системы дает в результате более четкую, хорошую и плотную интеграцию. Разумеется, такое тоже может произойти, но не само по себе; тут все зависит от закона Конвея. Но куда чаще встречается ситуация, когда интегрируемые части ориентированы на меняющуюся цель.

ЗАКОН КОНВЕЯ

В статье 1968 года в журнале *Datamation* Мелвин Конвей описал социологический феномен: «Проектирующие системы организации вынуждены делать проекты, структура которых воспроизводит принятую в этих организациях структуру связей». На обычный язык это иногда переводится так: «Если над компилятором работают четыре группы, вы получите компилятор, делающий четыре прохода».

Хотя все это напоминает комиксы Дилберта¹, на самом деле это результат серьезного, обоснованного анализа конкретных движущих сил, возникающих во время проектирования программного обеспечения. Конвей утверждает, что для построения интерфейса внутри или между системами два человека должны каким-то образом договориться о характеристиках этого интерфейса. Без такой договоренности ничего не получится.

Обратите внимание, что Конвей говорит о «структуре связей» организации. Как правило, она не совпадает с формальной структурой. Если два разработчика из разных отделов могут общаться напрямую, это отразится на одном или нескольких интерфейсах внутри системы.

Я видел применение закона Конвея в нормирующей форме — создание структуры связей, которую хотелось видеть воплощенной в программе, — и в иллюстративной форме — фиксация на бумаге структуры программы как попытка понять реальную структуру связей в организации.

С исходной статьей Конвея можно познакомиться на сайте автора по адресу <http://www.melconway.com/research/committees.html>.

Кроме того, разовая замена всех компонентов, обеспечивающих коммерческую деятельность, связана со значительным техническим риском. Если для построения системы не использовались паттерны стабильности, скорее всего, вы обнаружите

¹ Серия комиксов об офисной жизни, менеджерах, инженерах, маркетологах, боссах, юристах, бытовиках, практикантах, бухгалтерях и прочих странных людях.

там типичную жестко связанную архитектуру. В этом случае общая вероятность отказа системы равна сумме вероятностей отказов каждого из компонентов.

Но даже если система строилась с учетом паттернов стабильности (рассматриваемая система таковой не была), совершенно новые компоненты означают, что предсказать ее поведение во время эксплуатации невозможно. Вычислительная мощность, стабильность, управляемость и эксплуатационная гибкость находятся под большим вопросом.

Практически сразу после моего присоединения к проекту я понял, что вся деятельность группы разработчиков направлена на прохождение тестирования, а не на последующую эксплуатацию. Все конфигурационные файлы пятнадцати приложений и более пяти сотен точек интеграции писались для среды интеграционного тестирования. Имена хостов, номера портов, пароли к базам данных были разбросаны по тысячам конфигурационных файлов. Что еще хуже, некоторые компоненты приложений задавали топологию тестов качества, которая, как мы знали, не совпадает со средой эксплуатации. К примеру, при эксплуатации появлялись дополнительные фаерволы, не представленные на этапе тестирования. Это распространенное решение из серии «сберечь копейку, потратить рубль», экономящее несколько тысяч долларов на построение сети, но приносящее большие убытки из-за простоев и неудачных внедрений. Более того, во время тестирования некоторые приложения были представлены всего в одном экземпляре, в то время как для работы требовалось несколько экземпляров, объединенных в кластер. Еще среда тестирования во многих отношениях отражала устаревшие представления об архитектуре системы, хотя все «точно знали», что при эксплуатации все будет по-другому. Однако поменять тестовую среду было настолько непросто, что большинство разработчиков предпочло не обращать на все эти моменты внимания, лишь бы не потерять еще несколько недель привычных циклов «построение-развертывание-тестирование».

Когда я решил выяснить, какая конфигурация предполагается на стадии эксплуатации, мне казалось, что главное — найти человека или людей, которые уже решили эти проблемы. Я спрашивал: «В каком хранилище исходного кода можно увидеть итоговый вариант конфигурации?» и «Кто может сказать, какие свойства потребуется переопределить при переходе к эксплуатации?»

Иногда отсутствие ответов на подобные вопросы означает, что никто их просто не знает. Но бывает и так, что люди просто не хотят отвечать. В этом проекте было и то и другое.

Я решил составить список свойств, которые, возможно, потребуется изменить после начала эксплуатации: имена хостов, номера портов, URL-адреса, параметры подключения к базе данных, положение файлов журналов и т. п. Затем я набросился на разработчиков. Свойство с именем «host» неоднозначно, особенно когда с хостом во время тестирования качества связывается пять приложений. Оно может означать «мое собственное имя хоста», «хост, который может ко мне обращаться» или даже «хост, который я использую для отмывания денег». Прежде

чем определять значение свойства на стадии эксплуатации, следовало понять, что это за свойство.

После того как у меня появился список свойств, которые требовалось поменять после ввода сайта в эксплуатацию, пришло время определить структуру производственного развертывания. Для запуска требовалось внести изменения в тысячи файлов. Все они должны были переопределяться с каждой новой версией программного обеспечения. Идея при выходе каждой следующей версии посреди ночи вручную редактировать тысячи файлов была заранее обречена на провал. Кроме того, некоторые свойства повторялись многократно. Для одного только изменения пароля для входа в базу данных требовалось отредактировать более сотни файлов на двадцати серверах, а с ростом сайта эта проблема только усугублялась.

Столкнувшись с неразрешимой проблемой, я поступил как любой хороший разработчик: добавил уровень косвенности. Идея состояла в создании структуры переопределений, которая была бы отделена от основного кода приложения. Переопределения следовало структурировать так, чтобы каждое меняющееся при переходе из одной среды в другую свойство существовало только в одном месте. После этого все развертывания новых версий можно было осуществлять без перезаписи производственной конфигурации. Кроме того, пароли от рабочей базы данных больше не хранились ни в среде тестирования (к которой имели доступ разработчики), ни в системе управления исходным кодом (к которой имели доступ все сотрудники фирмы), что защищало конфиденциальность клиентов.

А во время настройки производственной среды я неосмотрительно вызвался помочь с нагрузочным тестированием.

7.3. Нагрузочное тестирование

Наш заказчик знал, что для успешного запуска новой непроверенной системы нагрузочное тестирование имеет решающее значение. Поэтому на данный этап нам был выделен целый месяц — это больше, чем когда-либо в моей практике. Отдел маркетинга объявил, что сайт должен быть в состоянии справиться с 25 000 со-пользователей (параллельно работающих пользователей).

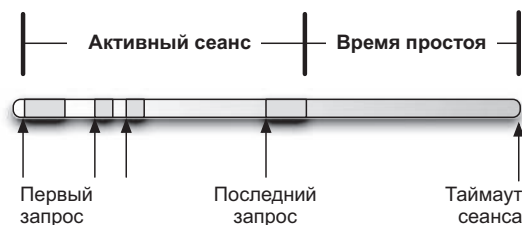
СОПОЛЬЗОВАТЕЛИ

Компании, занимающиеся нагрузочным тестированием, говоря про «сопользователей», часто подразумевают ботов. Некоторые инвесторы тоже начали использовать данный термин, подразумевая под ним параллельные сеансы. Но такой вещи, как «сопользователь», не существует. Если у вас нет двухуровневой клиент-серверной системы, в которой пользователи напрямую подключаются к базе данных, сопользователь — это фикция.

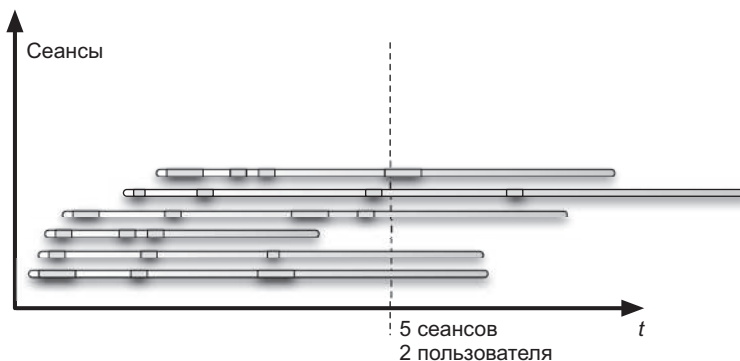
Подсчет сопользователей является некорректным способом оценки производительности системы. Если 100 % пользователей посмотрят на первую страницу и уйдут

восвояси, производительность получится намного выше, чем в случае, когда 100 % пользователей что-то купят.

Подсчитать сопользователей невозможно. Ведь вместо долговременного соединения есть лишь последовательность дискретных импульсов. Серверы получают такую последовательность запросов и связывают с каким-то идентификатором. Эта последовательность запросов идентифицируется с сеансом — абстракцией, облегчающей программирование приложений.



Обратите внимание, что на самом деле пользователь уходит в момент начала простоя. Сервер не видит разницы между пользователем, который больше не сделает ни одного щелчка, и пользователем, который пока не сделал ни одного щелчка. Поэтому он применяет таймаут. Сеансовая активность поддерживается некоторое время после последнего щелчка. Поэтому сеанс гарантированно продолжается дольше, чем пользователь находится на сайте. А значит, подсчитывая сеансы, мы переоцениваем число пользователей.



Если рассмотреть все активные сеансы, окажется, что часть из них неизбежно завершается без очередного запроса. Количество активных сеансов является одной из наиболее важных характеристик веб-системы, но не стоит путать его с количеством пользователей.

Нагрузочное тестирование обычно представляет собой в достаточной степени автоматический процесс. Вы определяете план тестирования, пишете сценарии (или берете сценарии от производителя), настраиваете генераторы нагрузки и диспетчер, который будет управлять процессом, а затем глубоко за полночь запускаете тест.

На следующий день, после завершения процесса, вы анализируете все собранные данные. После этого остается внести изменения в код или в конфигурацию и запланировать очередной прогон теста. Между прогонами обычно проходит от трех до четырех дней.

Но в нашем случае все требовалось сделать намного быстрее. Поэтому была устроена телефонная конференция с множеством участников, включая руководителя испытаний, инженера из службы нагрузочного тестирования, архитектора из группы разработчиков, администратора базы данных для наблюдения за тем, как будет использоваться база, и меня (я отвечал за приложения и серверы, осуществляющие мониторинг и анализ).

Нагрузочное тестирование одновременно является наукой и искусством. Сымитировать реальный трафик невозможно, поэтому приходится пользоваться анализом трафика, опытом и интуицией, чтобы имитация получилась по возможности близкой к реальности. Анализ трафика дает вам только переменные: паттерны просмотров, количество страниц на сеанс, коэффициенты обращаемости, распределение времени на обдумывание, скорости подключения, паттерны доступа к каталогу и т. п. Опыт и интуиция помогут определить важность каждой из переменных. Мы сочли самыми важными факторами время на обдумывание, коэффициент обращаемости, продолжительность сеанса и доступ к каталогу. Наши первые сценарии представляли смесь «любопытствующих», «ищущих» и «покупателей». Более 90 % сценариев просматривали только главную страницу и одну страницу с информацией о продукте. Они представляли любителей распродаж, заходящих на сайт практически каждый день. Мы оптимистично решили, что весь путь до оформления заказа будут проходить 4 % виртуальных пользователей. На этом сайте, как и на большинстве коммерческих сайтов, процедура оформления заказа требовала наибольшего количества ресурсов. Она включала в себя интеграцию с внешними службами (CCVS, нормализация адреса, проверка складских запасов и возможности заказа товара) и загрузку большего числа страниц, чем для любого другого сеанса. Пользователь, оформляющий заказ, как правило, посещает за сеанс двенадцать страниц, в то время как пользователь, просто путешествующий по сайту, как правило, заходит не более чем на семь страниц. Мы решили, что такой набор виртуальных пользователей нагрузит систему сильнее, чем реальный трафик.

Во время первого прогона сайт был полностью заблокирован, когда число со-пользователей достигло значения 1200. Пришлось перезагрузить все серверы приложений. Нам требовалось каким-то образом увеличить производительность системы в двадцать раз.

В течение следующих трех месяцев телефонные конференции устраивались каждый день и длились по двенадцать часов, при этом происходило много интересных вещей. В один памятный вечер инженер из группы нагрузочного тестирования обнаружил, что все машины в их нагрузочной ферме, работающие под управлением операционной системы Windows, одновременно начали загружать и устанавливать какую-то программу. Машины, которые мы использовали как генераторы нагрузки, кто-то взломал! В другой раз мы уперлись в потолок полосы пропускания.

Оказалось, что кто-то из инженеров фирмы AT&T обратил внимание, что одна из подсетей расходует «слишком много» полосы пропускания, и ограничил канал, генерировавший 80 % нашей нагрузки. Тем не менее, невзирая на все эти неприятности, мы изрядно усовершенствовали сайт. Каждый день мы обнаруживали новые узкие места и ограничители производительности. Мы смогли за один день поменять конфигурацию. Редактирование кода заняло немного больше времени, но все равно мы справились всего за два или три дня.

Мы даже внесли несколько значительных изменений в архитектуру меньше чем за неделю. В следующей главе вы найдете подробный рассказ о них.

Эта проведенная на ранней стадии предварительная оценка управляемости сайта в рабочем режиме дала нам возможность создать сценарии, инструменты и отчеты, которые вскоре оказались жизненно важными.

После трех месяцев тестирования и создания более шестидесяти новых приложений мы достигли десятикратного роста вычислительной мощности сайта. Он мог поддерживать 12 000 активных сеансов, что мы оценивали как примерно 10 000 посетителей за раз (с учетом всех подводных камней, связанных с подсчетом посетителей). Более того, под нагрузкой в 12 000 сеансов сайт не рухнул, хотя и стал вести себя несколько «капризно». За эти три месяца отдел маркетинга также пересмотрел намеченные характеристики. Там решили, что лучше иметь медленный сайт, чем не иметь его вовсе. Они больше не требовали 25 000 сопользователей, а считали, что 12 000 сеансов в это время года вполне достаточно. Хотя все ожидали, что к сезону отпусков потребуется внести значительные усовершенствования.

7.4. Растоптаны толпой

Что же случилось в день запуска? Почему сайт *так быстро* прекратил свою работу? Первым делом мы подумали, что отдел маркетинга просто ошибся в своих оценках спроса. Возможно, потенциальные покупатели слишком сильно ждали появления нового сайта. Эта теория быстро провалилась, так как мы выяснили, что дату запуска клиентам никто не сообщал. Возможно, имели место ошибки в конфигурации или несовпадение между средами эксплуатации и тестирования?

Подсчет сеансов сразу выявил источник проблемы. Сайт рухнул из-за количества сеансов. Именно сеансы являются ахиллесовой пятой любого сервера приложений. Каждый из них потребляет ресурсы, в основном RAM. При включенной репликации сеансов (а она была включена) каждый сеанс после каждого запроса страницы сериализуется и передается на резервный сервер. То есть каждый сеанс расходует ресурсы RAM, CPU и полосы пропускания. Откуда взялись все эти сеансы?

Сеансы являются ахиллесовой пятой любого сервера приложений.

В конечном счете мы поняли, что самой большой нашей проблемой были помехи. Нагрузочное тестирование проводилось при помощи сценариев, имитировавших реальных пользователей с реальными браузерами. Они переходили по ссылкам с одной страницы на другую. Все сценарии отслеживали сеансы через файлы cookie. Они *вежливо* вели себя с системой. При этом реальный мир может быть грубым, неаккуратным и жестоким.

В процессе эксплуатации происходят плохие вещи, предсказать которые заранее не всегда возможно. Так, одну из проблем нам создали поисковые системы. Именно они обеспечивали нам примерно 40 % всех посещений. Тем не менее, к сожалению, в день переключения они отправляли клиентов на устаревшие URL-адреса. Веб-серверы были настроены направлять все запросы `.html` на серверы приложений (из-за того, что последние могли отслеживать сеансы и формировать о них отчет). Фактически каждый клиент, пришедший из поисковой службы, гарантированно создавал на серверах приложений сеанс, чтобы в результате увидеть страницу 404.

Другой крупной проблемой, которую мы обнаружили, были пауки, присылаемые поисковыми службами. Некоторые из этих пауков (особенно присланных малоизвестными поисковыми службами) по разным причинам не сохраняли файлы cookie. Они не хотели повлиять на данные маркетинга или доход от рекламы. В общем случае пауки ожидают от сайта поддержки отслеживания сеансов через перезапись URL. Но без файлов cookie каждый запрос страницы приводит к открытию нового сеанса, который остается в памяти до истечения срока своей жизни (тридцать минут). Мы обнаружили поисковую службу, открывавшую до десяти сеансов в секунду.

Кроме того, были обнаружены анализаторы и боты, ищущие наиболее выгодные предложения. Мы нашли почти дюжину крупных анализаторов страниц. Некоторые из них крайне талантливо прятали свое происхождение. В частности, один отправлял запросы страниц из различных небольших подсетей, чтобы скрыть тот факт, что все они происходят из одного источника. Ведь даже последовательные запросы с одного IP-адреса могут использовать разные строки User-Agent для маскировки своего истинного происхождения.

Строка User-Agent — это HTTP-заголовок, посылаемый браузером для идентификации. Почти все браузеры пытаются представить себя в качестве варианта браузера Mozilla, и даже Internet Explorer от Microsoft.

Впрочем, интернет-регистратор ARIN¹ позволяет идентифицировать IP-адреса источника как принадлежащие одной организации. Подобные коммерческие анализаторы на самом деле продают услуги консультационного характера. Продавец,

¹ См. <http://www.arin.net>.

желающий быть в курсе цен конкурентов, может подписаться на рассылку таких фирм. И раз в неделю или раз в день получать отчет о том, что и по каким ценам продают конкуренты. Именно поэтому некоторые сайты не показывают цену, пока товар не окажется в вашей корзине. Разумеется, ни один из таких анализаторов не обрабатывает файлы cookie должным образом, поэтому каждый из них открывает дополнительные сеансы.

Также нам пришлось иметь дело с любительскими ботами, занимающимися поиском наиболее выгодных цен. С нескольких IP-адресов раз в минуту на старой версии сайта они заходили на одну и ту же страницу с описанием продукта. Для идентификации этого продукта потребовалось некоторое время. В конечном счете это оказалась игровая приставка PlayStation 2. Дефицитом эти приставки не являлись уже три года, а сценарии все еще занимались поиском доступных для заказа приставок, что приводило к открытию еще большего числа сеансов.

Наконец, были источники, которые мы называли «непонятные, странные штуки»¹. К примеру, один компьютер с базы военно-морского флота отображался как обычный сеанс с просмотром страниц, но через пятнадцать минут после последнего правомерного запроса страницы последний URL-адрес начинал запрашиваться снова и снова. Здравствуйтесь, дополнительные сеансы.

7.5. Недочеты тестирования

Вопреки всем нашим усилиям по нагрузочному тестированию сайт не выдержал столкновения с реальным миром. При тестировании мы упустили две вещи.

Во-первых, приложение тестировалось тем способом, которым *оно должно было использоваться*. Сценарии запрашивали один URL-адрес, ждали ответа, а затем запрашивали следующий URL-адрес, присутствовавший на присланной в ответ странице. Ни один из сценариев нагрузочного тестирования не требовал один и тот же URL-адрес без файлов cookie 100 раз в секунду. Если бы подобное произошло, мы, скорее всего, назвали бы тест «нереалистичным» и проигнорировали бы факт падения серверов. Так как для слежения за сеансами сайт использовал только файлы cookie, не прибегая к перезаписи URL-адреса, все наши сценарии нагрузочного тестирования пользовались этими файлами.

Короче говоря, все наши тестовые сценарии подчинялись правилам. Они напоминали тестера, который нажимает кнопки только в правильном порядке. Большинство моих знакомых тестеров столь добросовестны, что если им подсказать «корректный маршрут» через приложение, они никогда им не воспользуются. Аналогичный подход следует применять при нагрузочном тестировании. «Помехи» могут до некоторой степени уменьшить мощность системы, и порой этого вполне достаточно для прекращения работы сайта.

¹ Да... на самом деле вместо слова «штуки» мы использовали более грубый аналог.

Не ограничивайтесь прогулками по «корректным маршрутам».

Во-вторых, разработчики приложения не позаботились о способах защиты, которые могли бы противостоять плохим вещам. Когда что-то шло не так, приложение продолжало посылать потоки в опасную зону. И как в случае аварии на автомагистрали во время тумана, новые потоки, обрабатывающие запросы, наслаивались на уже заблокированные или повисшие.

7.6. Последствия

Решительные действия в дни и недели, последовавшие за запуском, дали впечатляющие результаты. Инженеры, отвечавшие за CDN, искупили свою вину за «небольшие недочеты» перед запуском. За один день они написали для пограничных серверов сценарии, защищающие от самых злобных атак. Они добавили шлюзовую страницу, обслуживавшую три основные вещи. Во-первых, если источник запроса некорректно работал с файлами cookie, его браузер направляли на специальную страницу, где объяснялось, как включить режим передачи файлов cookie. Во-вторых, мы настроили регулятор количества доступных новых сеансов. Если ему присваивалось значение 25 %, то только 25 % запросов к шлюзовой странице направлялись на главную страницу сайта. Остальные получали составленное в крайне вежливых выражениях сообщение с просьбой зайти позднее. В течение следующих трех недель за счетчиком сеансов постоянно наблюдал инженер, готовый перенастроить регулятор, как только создается впечатление, что объем запросов стал слишком большим. В случае полной перегрузки серверов восстановление обслуживания страниц занимает по меньшей мере час, поэтому регулятор, предотвращающий подобную ситуацию, был для нас жизненно важным. К третьей неделе мы смогли оставлять регулятор со значением 100 % в течение целого дня. В-третьих, мы получили возможность блокировать доступ на сайт с определенных IP-адресов. Как только мы обнаруживали одного из ботов, занимающихся сравнением цен, или огромный поток запросов, мы просто добавляли их в список блокировок.

Все эти вещи можно было реализовать средствами приложения, но в сумасшедшей спешке, последовавшей за запуском, было проще и быстрее использовать для этого CDN. К тому же нам следовало внести еще ряд глобальных изменений.

Главная страница генерировалась полностью динамически, начиная от JavaScript-кода для выпадающих меню и заканчивая ссылками на информацию о продуктах и даже на расположенные в нижней части страницы «условия использования». Одним из ключевых коммерческих доводов именно такой платформы приложения была персонализация. Отдел маркетинга испытывал по поводу этого программного

компонента особый восторг, но толком не решил, как его следует использовать. Поэтому пять миллионов раз в день генерировалась одна и та же главная страница. Хотя для каждого ее построения требовалось более 1000 транзакций базы данных. (Даже если данные уже были кэшированы, транзакция все равно происходила, потому что именно таким способом работала данная платформа.) Написанные на языке JavaScript выпадающие меню, меняющиеся при наведении на них указателя мыши, требовали перебора более восьмидесяти категорий. А анализ трафика показал, что значительный процент ежедневных посетителей не уходил дальше главной страницы. Большинство из них не предоставляло идентификационных файлов cookie, то есть персонализация была попросту невозможной. Но даже отправка сервером приложений главной страницы требует времени и приводит к открытию сеанса, который занимает память на следующие тридцать минут. Поэтому мы быстро написали сценарии, создающие статическую страницу, которая и предоставлялась всем неустановленным пользователям.

Вы когда-либо обращали внимание на юридические условия, опубликованные на большинстве коммерческих сайтов? Там пишутся, например, такие вещи: «Просматривая эту страницу, вы соглашаетесь на следующие условия...» Оказывается, они существуют по единственной причине. Обнаружив программу-анализатор или бота, занимающегося сравнением цен, владелец сайта может натравить на виновную сторону адвоката. Первые несколько дней наш юридический отдел работал не покладая рук. Обнаружив на сайте очередную шайку запрещенных ботов, занимающихся анализом контента или сбором цен, адвокаты отправляли требование прекратить данную деятельность; в большинстве случаев активность ботов прекращалась. (Но как и в случае, когда гонишь собаку от обеденного стола, боты всегда возвращаются — иногда замаскировавшись.)

Одно из самых героических усилий в этот период хаоса было предпринято в неделю запуска. Руководитель ИТ-отдела обнаружил шесть дополнительных серверов, конфигурация которых совпадала с нашей. Они были зарезервированы другим отделом, но пока не использовались. Руководитель забрал их для коммерческого сайта (и предположительно для плановых замен) в качестве дополнительных серверов приложений. Один из наших системных администраторов провел 36-часовой марафон для их подготовки: устанавливал операционную систему, задавал конфигурацию сети и файловой системы, обеспечивал доступ к SAN и мониторинг. После завершения этой работы кто-то отвез его в гостиницу, где он упал на кровать и сразу уснул. А я смог получить серверы приложений и приложения, установленные и настроенные в один день. За два дня мы удвоили вычислительную мощность слоя серверов приложений.

Механизм переключения сеансов этого конкретного сервера приложений основан на сериализации. Сеанс пользователя остается привязанным к экземпляру исходного сервера, поэтому все новые запросы возвращаются экземпляру, уже имеющему в памяти этот сеанс. После каждого запроса страницы сеанс сериализуется и отправляется на «сервер резервирования сеансов». Именно последний хранит в памяти все сеансы. Если экземпляр сервера, с которым общался пользователь, прекратит свою

работу — намеренно или по другой причине, — следующий запрос будет передан другому экземпляру, выбранному распределителем нагрузки. Этот новый экземпляр попытается загрузить сеанс пользователя с сервера резервирования сеансов. Этот механизм хорошо работает (и на удивление хорошо масштабируется), учитывая, что все сеансы хранятся в памяти, а не в базе данных и не на диске; соответственно, он хорошо масштабируется, пока данные сеанса имеют небольшой объем. Как правило, к этим данным относятся: идентификатор пользователя, идентификатор его корзины и, возможно, поисковая информация, например ключевые слова и индекс страницы результатов. Обычно никто не добавляет в сеанс всю корзину в сериализованном виде или все результаты поиска (количеством более 2000). Но, как ни грустно, именно это мы обнаружили в сеансах. Нам пришлось отключить механизм переключения.

Все эти действия по обеспечению быстрой ответной реакции имеют общие черты. Во-первых, ничто не является столь постоянным, как временное исправление. Большинство из них никуда не делось даже за пару лет. Во-вторых, все это стоит больших денег, в основном в плане неполученного дохода. Очевидно, что пользователи, которые из-за ограничений не смогли попасть на сайт, вряд ли сделают там заказ. (По крайней мере, снижается вероятность их покупок именно *на этом* сайте.) Без переключения сеансов пользователь не сможет закончить оформление заказа, если что-то произойдет с экземпляром сервера, обслуживающим его запрос. Например, вместо страницы подтверждения заказа он будет отправлен в собственную корзину. Большинство пользователей, снова увидевшие перечень заказанных товаров после того, как они практически закончили оформление, просто уходят. Статическая главная страница затрудняет персонализацию, хотя именно эта функция была основной причиной изменения архитектуры проекта. То, что нам пришлось заплатить за дополнительное аппаратное обеспечение для сервера приложений, это очевидно, но заодно появились и дополнительные затраты на обслуживание. Наконец, нельзя забывать про альтернативные издержки из-за того, что следующий год нам предстояло потратить на исправление ошибок в данном проекте вместо работы над новыми коммерчески выгодными программными компонентами.

А хуже всего то, что эти потери оказались совершенно напрасными. На момент написания этих строк прошло два года со дня запуска сайта. Сейчас сайт даже без обновления аппаратного обеспечения справляется с нагрузкой, ставшей в четыре раза больше. Настолько улучшилось программное обеспечение. Если бы сайт изначально был построен таким образом, инженеры могли бы присоединиться к торжеству отдела маркетинга и выпить шампанского, а не пытаться всеми силами спасти ситуацию.

8

Понятие вычислительной мощности

Еще до того, как лопнул пузырь доткомов, одна из реклам IBM точно подметила парадокс развертывания систем. Группа озабоченных предпринимателей собралась вокруг компьютера, чтобы наблюдать за запуском нового сайта. Счетчик «полученные заказы» начал вертеться, и группа зааплодировала. Аплодисменты сменились неловким молчанием, когда счетчик устремился в стратосферу.

Мне знакомо это чувство. Именно это чувство беспомощности испытала наша группа при запуске коммерческого сайта, о котором я рассказал в предыдущей главе. В то же время я видел, как за восемнадцать месяцев система была усовершенствована настолько, что смогла справиться с четырехкратной нагрузкой, даже когда от исходного аппаратного обеспечения остались только две трети. Причиной возросшей вычислительной мощности были изменения, внесенные в программный проект.

В этой главе вы познакомитесь с таким понятием, как вычислительная мощность, а также с определенными паттернами и антипаттернами, влияющими на эту характеристику. Следующая глава будет посвящена финансовым аспектам планирования мощности.

8.1. Определение вычислительной мощности

Маркетологи и менеджеры считают понятия *производительность* и *вычислительная мощность* взаимозаменяемыми. Однако в области архитектуры и проектирования

нужна бóльшая точность. Рискаю показаться чрезмерно педантичным, я хотел бы дать этим терминам определение.

Производительность (performance) показывает, насколько быстро система обрабатывает одну транзакцию. Этот параметр может измеряться как в изоляции, так и под нагрузкой. Производительность системы весьма важна для оценки ее пропускной способности. Однако клиента, использующего данный термин, на самом деле интересует совсем не производительность. Его интересует либо пропускная способность, либо вычислительная мощность. В то же время конечным пользователям вычислительная мощность неинтересна; их волнует только производительность выполнения их собственных транзакций. Они не могут зарегистрироваться на серверах и посмотреть, работают ли приложения. И если время ожидания ответа превышает некий предел, они считают, что система вышла из строя.

Пропускная способность (throughput) описывает количество транзакций, которые система может провести за определенный промежуток времени. Производительность системы однозначно влияет на ее пропускную способность, но не всегда линейно. Пропускная способность всегда ограничена характеристиками системы — нехваткой ресурсов. Оптимизация производительности фрагментов системы, не являющихся слабым местом, не увеличит полосу пропускания.

Термин *масштабируемость* (scalability) используется в двух ситуациях. Во-первых, он описывает изменение пропускной способности при варьировании нагрузки. Мерой масштабируемости в этом случае является количество запросов в секунду как функция от времени отклика. Во-вторых, этот термин относится к поддерживаемым системой режимам масштабирования. Мы будем его использовать в смысле добавления в систему вычислительной мощности.

Ну и наконец, максимальная пропускная способность, которую при данной нагрузке может поддерживать система, сохраняя для каждой отдельной транзакции приемлемое время отклика, называется ее *вычислительной мощностью* (capacity).

Обратите внимание, что это определение включает в себя несколько важных переменных. Не существует одного фиксированного числа, которое можно рассматривать как вычислительную мощность. Если нагрузка изменится — например, потому, что во время праздников пользователей начинают интересовать другие сервисы, — может резко измениться и мощность системы.

Кроме того, это определение требует оценки. Что именно считать «приемлемым временем реакции»? В случае интернет-магазина любое время реакции, превышающее две секунды, приводит к потере клиентов. Для финансовой биржи оно может быть еще короче — порядка миллисекунд. В то же время в системе бронирования путешествий вполне допустимы пять сотен миллисекунд на поиск доступных вариантов и тридцать секунд на подтверждение бронирования.

8.2. Ограничители

Имея дело с вычислительной мощностью, сложнее всего работать с нелинейными эффектами. Когда требуется вести машину или поймать бейсбольный мяч, наш мозг способен достаточно быстро интегрировать дифференциальные уравнения. Но как только речь заходит о вычислительной мощности, все почему-то хотят вернуться к линейной зависимости. Вам когда-нибудь задавали вопрос: «Если на обработку 10 000 пользователей уходит 50 % ресурсов процессора, значит ли это, что всего мы можем обслужить 20 000 пользователей?»

Вычислительная мощность каждой системы определяет ровно один ограничитель¹. Это фактор, который первым достигнет предельного значения. После достижения этого значения все остальные части системы начнут ставить свои задания в очередь или пропускать их. Предположим, что в роли ограничителя выступает ваш сервер базы данных. Сервер Oracle с включенным режимом многопоточности (MTS) может обрабатывать столько параллельно поступающих запросов, сколько допускает конфигурация процесса-демона. Пусть у нас будет небольшой сервер базы данных со всего пятьюдесятью обрабатывающими все входящие запросы процессами. Пятьдесят первому запросу уже придется ждать своей очереди. Приславший этот запрос сервер приложений мог бы обслужить дополнительные страницы, если бы получил назад свои данные. Соответственно, в ожидании ответа сервера приложений будет находиться и исполняемый модуль веб-сервера.

Теперь предположим, что ограничителем является оперативная память серверов приложений. Каждый пользовательский сеанс потребляет определенное количество RAM. Как только свободной памяти не останется, новые сеансы заставят сервер приложений начать подкачку страниц. В этом случае веб-сервер, вероятно, будет просто ждать ответа от сервера приложений. В момент, когда сервер приложений начнет пробуксовку, серверу базы данных повезет больше — он получит отказ и расслабится.

Любые не связанные с ограничителем показатели при проектировании или увеличении вычислительной мощности бесполезны. Если ваша база данных начала дымиться, не имеет смысла рассматривать коэффициент загрузки процессора вашего веб-сервера, чтобы учесть в проекте количество сеансов, с которым может справиться система. Одновременно это означает, что, если вы обнаружите ограничитель, увеличение мощности можно будет надежно предсказать с учетом вносимых в этот ограничитель изменений. Если ограничителем является память, увеличение ее объема будет увеличивать мощность, естественно, пока не появится какой-то другой вид ограничителя.

Для понимания природы мощности любой системы требуется *системное мышление*, описанное Питером Сенге в книге *Fifth Discipline*, — способность думать с точки

¹ См. книгу *The Goal* Элияху Голдратта, в которой рассказывается о теории ограничений.

зрения динамических переменных, изменений во времени и взаимосвязанных соединений. Простой формулы, позволяющей рассчитать всеобъемлющее «значение вычислительной мощности», не существует.

Начните с рассмотрения системы как целого. Найдите задающие переменные. К этим переменным относятся такие параметры, как «количество запросов страницы в секунду». На «общесистемном уровне» для поиска таких переменных нужно обращать внимание на параметры, не поддающиеся вашему контролю, например пользовательский спрос, время, дату и т. п.

Задающие переменные связаны причинно-следственной связью с зависимыми переменными. Значение последних меняется в ответ на изменение задающих переменных. К зависимым переменным относится вся измеряемая напрямую статистика производительности: загрузка процессора, свободная память, скорость ввода-вывода, скорость подкачки страниц, пропускная способность сети и т. п. В поиске зависимых переменных для каждой задающей переменной помогают нагрузочное тестирование, тестирование в предельных режимах, мониторинг работающей системы и анализ данных¹. Имейте в виду, что каждая зависимая переменная может иметь значительную корреляцию с более чем одной задающей переменной.

От системы в целом можно перейти к рассмотрению слоев или подсистем, продолжая поиск задающих и зависимых переменных. При этом их природа может меняться от слоя к слою. К примеру, ввод-вывод базы данных определяет время реакции сервера, которое, в свою очередь, определяет потребление памяти веб-сервера. То есть переменная, которая является зависимой в одном слое, в другом может превратиться в задающую. Эта сеть взаимосвязей показывает, как изменения в главных задающих переменных расходятся по всей системе.

Ограничителем в системе будет предельное значение одной из зависимых переменных. После достижения этого лимита она больше не сможет удовлетворять спрос задающей переменной. Корреляционный анализ с окнами переменных покажет, что между ограничивающей переменной и основными задающими переменными до момента достижения предела существует сильная корреляция. Но после достижения предела корреляция исчезает, так как число запросов продолжает расти, а возможность их удовлетворения должна уменьшиться. Это дает хорошо известный изгиб на графиках нагрузочного тестирования. Быстрый спад в точке изгиба показывает, что ограничение уже достигнуто.

После обнаружения ограничителя остается самое простое. Для увеличения вычислительной мощности нужно увеличить пороговое значение, добавив ресурс, влияющий на ограничивающую переменную, или уменьшив использование этого ресурса.

¹ Нас интересует высокий коэффициент корреляции, где-то между 0,8 (высокая корреляция) и 1,0 (абсолютная корреляция).

8.3. Взаимное влияние

Часто можно видеть, как эффекты одного слоя в другом слое превращаются в факторы, влияющие на ситуацию. К примеру, если спрос в одном из слоев превышает его вычислительную мощность, производительность слоя уменьшается. Слой начинает медленно реагировать или перестает реагировать вообще. При этом медленная реакция хуже ее отсутствия. Ее появление может стать триггером для каскадных отказов в другом слое. Поэтому разделить проблемы с мощностью и проблемы со стабильностью в подобных ситуациях крайне сложно. Во время запуска интернет-магазина, описанного в главе 7, рабочей группе пришлось иметь дело с серьезными проблемами с мощностью, которые напрямую привели к проблемам со стабильностью.

8.4. Масштабируемость

Работоспособные системы со временем начинают пользоваться все большим спросом. В какой-то момент возникает необходимость в дополнительной мощности, что зачастую означает дополнительное аппаратное обеспечение. Горизонтально масштабируемая система может расти путем добавления серверов. Вертикально масштабируемая система требует обновления существующих серверов.

Любой сервер, который можно поместить в однородный пул ресурсов за распределителем нагрузки или виртуальным IP-адресом, допускает горизонтальное масштабирование, как показано на рис. 20. Идеальной для горизонтального масштабирования является ситуация, когда каждый сервер может работать, ничего не зная о своих соседях. Такие «неразделяемые» варианты архитектуры обеспечивают практически линейный рост мощности. Удвоение числа серверов должно фактически удваивать мощность (если добавочная нагрузка не поставит какую-то службу в подчиненное положение). Кластерная архитектура также допускает горизонтальное масштабирование, хотя в данном случае соотношение уже не является строго линейным, так как приходится тратить ресурсы на управление кластером.

Веб-серверы идеально масштабируются по горизонтали. Как и серверы Ruby on Rails. Горизонтальное масштабирование таких серверов J2EE-приложений, как WebSphere, WebLogic и JBoss, достигается за счет кластеризации.

Однако иногда добавление параллельных серверов непрактично или невозможно. В таких случаях каждый отдельный сервер должен иметь максимальные возможности. Например, серверы баз данных при попытке объединить в кластер три и больше резервных сервера становятся неудобными в управлении. Так что лучше взять пару сильных серверов с автоматическим переключением в случае сбоя.

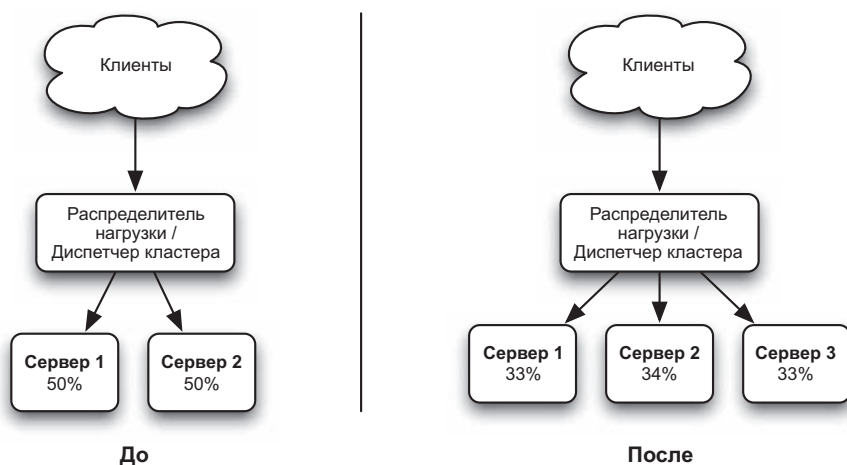


Рис. 20. Горизонтальное масштабирование

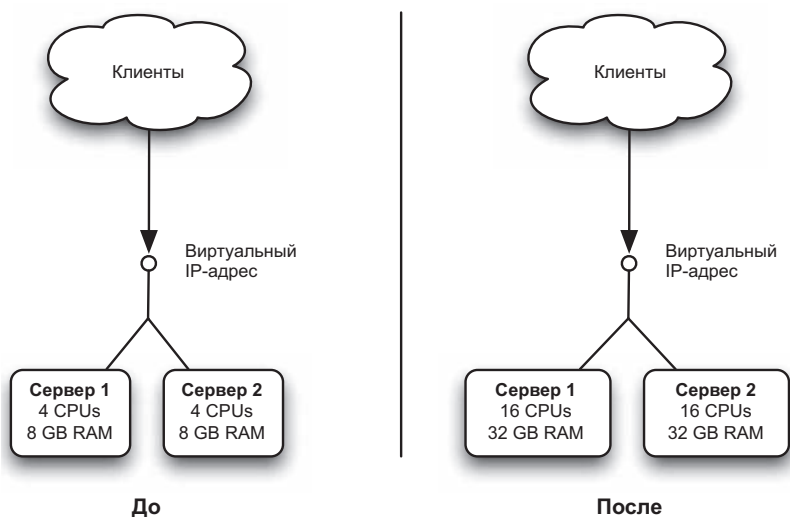


Рис. 21. Вертикальное масштабирование

Полное обновление оборудования (forklift upgrade) — замена корпуса сервера со всем его содержимым. Серверы больших баз данных могут весить сотни килограммов.

Вертикальное масштабирование требует достаточного пространства в корпусе сервера, чтобы туда можно было вставлять дополнительные процессоры и оперативную память, как показано на рис. 21. Как только место кончается, добавить

дополнительные «лошадиные силы» отдельным машинам можно только путем *полного обновления оборудования*. Так как корпуса, в которые можно добавлять большое количество CPU, как правило, являются самыми дорогостоящими из всех: начальная стоимость вертикально масштабируемой архитектуры больше, чем горизонтально масштабируемой. Последняя позволяет более гибко тратить средства, выделенные на инфраструктуру; вам не придется связывать крупные суммы с несколькими огромными корпусами. Ничто не мешает начать с минимального количества необходимых серверов и постепенно их увеличивать. Это позволяет не идти одновременно на большие расходы.

8.5. Мифы о вычислительной мощности

Порой люди верят в странные вещи. Например, в Средние века верили, что ведьмы не тонут, а сейчас верят, что степень загрузки процессора помогает определить вычислительную мощность системы. Некоторые из этих верований безвредны. Если руководитель проекта считает, что система рухнет каждый раз, когда он надевает свою гавайскую рубашку, возможно, не имеет смысла его разубеждать. В то же время некоторые ошибочные суждения могут стоить компании миллионы долларов.

Дешевизна процессоров

Мне встречались страшные нарушения, совершенные неквалифицированными разнорабочими, которые без тени сомнения оправдывались тем, что процессоры стоят дешево. Это все равно что заявить: «Арахисовое масло стоит дешево, поэтому для бутерброда я возьму его в три раза больше, чем обычно». В 1960-х годах стоимость компьютера равнялась зарплате программиста за несколько лет. И тратить часы или даже дни рабочего времени программиста на попытки уменьшить количество используемых циклов процессора было вполне естественно. В данном случае использовался дешевый ресурс, чтобы снизить потребление дорогого ресурса. Но в наши дни процессор, как правило, стоит меньше, чем полдня работы программиста. Так зачем же тратить рабочее время на оптимизацию загрузки процессора?

Сами по себе кремниевые микросхемы могут быть дешевыми (особенно по сравнению с ценами в прошлом), но циклы процессора стоят дорого. Каждый цикл процессора потребляет тактовое время. А тактовое время — это период ожидания. Неэкономичные приложения заставляют пользователей ждать дольше, чем это требуется, а пользователи больше всего ненавидят ждать. В случае веб-систем задержка в приложении производит двойственный эффект. Дополнительная обработка увеличивает нагрузку непосредственно на серверы приложений. Предположим, что приложению на каждую транзакцию требуются дополнительные 250 миллисекунд. Если в день система обрабатывает миллион транзакций, эти секунды превращаются в дополнительные 69,4 часа компьютерного времени. Исходя из предположения, что коэффициент загрузки сервера составляет 80 %, на обработку дополнительной нагрузки нам потребуется четыре дополнительных сервера.

В реальности 250 миллисекунд на транзакцию каждый день добавляют к процессорному времени 69,4 часа.

Так как большинство серверов приложений обрабатывает транзакции потоками из пула, возникает нелинейное влияние на пропускную способность. Чем дольше поток, обрабатывающий запрос, выгружается из пула, тем выше вероятность, что следующий входящий запрос не будет обработан немедленно, а окажется помещенным в очередь.

Задержка в работе веб-приложений вредит и веб-серверам. В процессе ожидания ответа от сервера приложений веб-сервер удерживает некоторые свободные ресурсы: по меньшей мере два сокета (один для входящего HTTP-запроса и один для сервера приложений), память для хранения состояния запроса и память для частично буферизованного ответа. Даже если веб-сервер ничего не делает для обслуживания этого подключения, указанные ресурсы ограничены. Загрузка CPU сервера приложений приводит к использованию памяти веб-сервера.

Наконец, процессоры имеют разную стоимость. И дело не только в том, что PA-RISC дороже, чем Intel x86. В одну машину помещается только определенное количество процессоров. И если в корпусе есть место только под четыре процессора, добавление пятого будет стоить непропорционально дорого, потому что придется покупать новый корпус. После этого вам потребуются новая оперативная память, возможно, какой-то локальный жесткий диск, сетевые карты, может быть, адаптер Fibre Channel, кулеры и т. п. Потребуется место в стойке или на полу, если корпус слишком велик. Лицензия на программное обеспечение может быть привязана к хосту, а не к процессору. Появятся дополнительные расходы на мониторинг/обслуживание нового компьютера. Кроме того, возникнет дополнительная нагрузка на систему охлаждения центра сбора и хранения данных.

Следующие диаграммы показывают приращение стоимости при добавлении каждого следующего процессора. При этом не учитывается ни стоимость места на жестком диске, ни стоимость программного обеспечения, ни административные расходы. На диаграмме представлены только цены микросхемы, корпуса и оперативной памяти. Для серверов начального уровня на рис. 22 приращение стоимости невелико, потому что корпуса стоят недорого. Разница возникает в основном из-за необходимости дополнительной памяти и места на жестком диске. В любом случае разница в стоимости добавления процессора, не требующего и требующего покупки нового корпуса, составляет примерно 1,2. То есть стоимость добавления третьего процессора примерно в 1,2 раза выше стоимости добавления второго. При повышении уровня картина радикально меняется. Переход в другой корпус в случае более совершенных систем обойдется вам намного дороже. Корпуса для серверов Sun Fire 6900, представленные на рис. 23, стоят более 200 000 долларов. Новый корпус для сервера E25K в минимальной конфигурации стоит более миллиона долларов. Разумеется, переход к другому корпусу происходит при куда больших количествах процессоров: например, корпус для E25K может вместить 72 процессора, но все равно потребуется заплатить миллионы долларов за отладку и оптимизацию

в стеке приложений. Перед тем как вкладывать миллион долларов, хотелось бы быть уверенным, что процессоры не будут работать впустую. Вы явно захотите знать, действительно ли вам требуется добавлять 73-й процессор!



Рис. 22. Рост стоимости процессоров для серверов Sun 440



Рис. 23. Рост стоимости процессоров для серверов Sun Fire 6900

Место на диске стоит дешево

Зачастую люди считают, что дисковое пространство стоит дешево. Можно ли с этим поспорить? В конце концов, десять лет назад стоимость накопителей составляла около доллара за мегабайт; сейчас же гигабайт стоит меньше пятидесяти центов¹. Конечно, существуют и дисковые массивы SCSI premium, но в основном накопители столь дешевы, что вы можете хранить на ноутбуке все когда-либо сделанные вами фотографии. (Правда, к профессиональным фотографам последнее утверждение не относится.)

Проблема в том, что современный термин *хранилище данных* (storage) означает не только отдельный накопитель. Это целая управляемая система дисков, взаимосвязей, выделения, избыточности и резервирования, обеспечивающая высокий уровень обслуживания по выгодной цене. На современных крупных предприятиях хранилище данных — это скорее служба, а не элемент недорогого стандартного оборудования. Все начинается с жестких дисков, но ими не заканчивается.

Хранилище данных — это служба, а не устройство.

В случае, когда жесткий диск вставлен в сервер, на нем должно быть достаточно места для:

- ☐ операционной системы;
- ☐ приложений;
- ☐ локальных конфигураций или данных;
- ☐ файлов журналов регистрации;
- ☐ временного рабочего пространства приложений.

Именно столько места требуется каждому серверу. В итоге мы снова сталкиваемся с эффектом умножения: цена диска емкостью 1 Гбайт невелика, но эту покупку нужно сделать n раз, по разу для каждого сервера. То есть для двадцати серверов вам потребуется 20 Гбайт, а не 1 Гбайт.

Кроме того, не забудьте про технологию RAID. Почти все серверы, используемые для центров хранения и обработки данных, загружаются с томов RAID 1. RAID-массив 1 делает зеркальную копию дисков, что означает 100 % непроизводительных расходов, то есть двойного количества дисков. Данные, как правило, хранятся на томах RAID 5, означающих 20 % непроизводительных расходов. В любом случае вам все равно придется иметь дело с эффектом умножения. При работе с зеркальным

¹ На 13 января 2007 года стоимость SATA-диска на 500 Гбайт составляла 147 долларов (источник: <http://www.pricewatch.com>).

массивом нужно двойное пространство. В итоге 1 Гбайт превращается в 40 Гбайт (1 Гбайт, зеркально копируемый для каждого из двадцати серверов).

Если принять во внимание резервное копирование, ситуация становится еще хуже. Дополнительный гигабайт данных, скопированный на двадцать серверов, существенно повлияет на резервные копии. Чтобы вовремя завершить резервное копирование, могут потребоваться дополнительные ленточные накопители, работающие параллельно. А также дополнительные магнитные ленты.

Гигабайт памяти на местном запоминающем устройстве стоит меньше доллара. Но в случае *управляемого хранилища данных* на предприятии эта сумма может доходить до 7 долларов за гигабайт. Поэтому важно обсудить проблемы хранилища данных и управления им с ИТ-отделом предприятия.

Управляемое хранилище данных (managed storage) — это дисковая память плюс такие системы обеспечения надежности, как RAID-чередование или зеркалирование, плюс резервные копии.

ДЖО СПРАШИВАЕТ: ЧТО ТАКОЕ SAN И NAS?

Как вам нравится ситуация, когда пара производителей так старательно работает над созданием собственной торговой марки и так старательно пытается выделить ее среди остальной продукции, что в результате названия звучат практически одинаково? Технологии SAN и NAS радикально отличаются друг от друга, но из-за схожести названий их часто путают.

Сетевое хранилище данных (Network-Attached Storage, NAS) представляет собой устройство хранения, которое подключается к вашей IP-сети и доступ к которому реализуется через протокол NFS или CIFS (папки общего доступа в Windows). NAS-устройства часто поддерживают несколько вспомогательных протоколов, например HTTP и FTP, все из одного набора дисков и их файловых систем. Как правило, они используются в операционной системе Linux или внедренной операционной системе Windows, хотя бывают и варианты устройств, с которыми можно работать в некоторых системах семейства UNIX. В большинстве случаев NAS-диски объединяются в RAID-массив 0, 1 или 5.

Сеть хранения данных (Storage Area Network, SAN) — совсем другое дело. Это сеть, полностью отделенная от вашей IP-сети. Сети хранения данных используют протокол Fibre Channel, способный передавать по 2 Гбайт данных в секунду. Если у вас есть SAN, значит, где-то находится большая полка, в которую установлены многочисленные жесткие диски. Управляющее программное обеспечение сетей хранения данных позволяет системным администраторам разбивать физические диски на логические — с RAID-массивами любого уровня. Благодаря хост-адаптеру шины (Host Bus Adapter, HBA) для клиентской операционной системы удаленные диски выглядят так же, как обычные локальные физические диски.

Поскольку SAN — это совершенно другая отдельная система, эта сеть требует дорогостоящей инфраструктуры. Она немедленно становится критически важной для бизнеса, а потому должна быть быстрой, надежной и избыточной. Надежность SAN обеспечивается многоканальностью; каждый сервер должен иметь несколько каналов доступа ко всем своим томам. Это означает использование двухканальных карт на серверах, двойных коммутаторов и двойных контроллеров на дисковых полках.

Первоначальные затраты на NAS невысоки; сетевое хранилище данных на 500 Гбайт сейчас доступно меньше чем за 700 долларов. NAS-устройство объемом в терабайт можно приобрести меньше чем за 1000 долларов. В то же время SAN для предприятия стоит по меньшей мере миллион долларов. К тому же для SAN требуется непрерывное управление, а NAS, как правило, превращается в место для сохранения данных, не требующее структурирования или непрерывного управления. Принять решение о построении NAS можно на уровне отдела или даже работающей над проектом группы, в то время как решение о создании SAN принимается директором по инвестициям.

Полоса пропускания стоит дешево

Этот миф распространен чуть меньше предыдущих, но тем не менее его можно услышать достаточно часто. На современном рынке плата за подключение к магистральному провайдеру составляет от 7500 до 12 000 долларов в месяц. Серьезная фирма нуждается хотя бы в паре подключений, желательно от разных операторов. Распределение нагрузки обоих соединений даст вам теоретический максимум полосы пропускания в 310 Мбайт в секунду за 15 000–24 000 долларов в месяц.

В зависимости от формы вашего трафика может оказаться целесообразным платить за фактически потребленную, а не за выделенную полосу. Цена выделенного подключения высчитывается легко. У вас есть одинаковая полоса пропускания, доступная в любую секунду по фиксированной цене. Во втором же случае вы платите по фиксированной ставке за «потребленную» полосу пропускания, что обычно намного меньше, чем за выделенную линию. Если ваш расход трафика не превышает некоего заданного уровня, дополнительная плата не взимается.

Если скорость передачи данных превышает выделенную полосу пропускания, вы начинаете платить за мегабайт в минуту¹. Оплата фактического потребления может оказаться ниже, чем плата за выделенную линию. Но если ежедневный пик трафика всегда превышает выделенную величину, цена быстро начинает расти. В случае резкого всплеска посещаемости сайта остается только шире открыть кошелек!

Итак, сколько же параллельных запросов может одновременно пройти через пару магистральных соединений? Давайте посмотрим на некоторые переменные. Как ни странно, чем больше пользователей обслуживается вашим соединением, тем хуже будет выглядеть полоса пропускания. Квотирование TCP/IP гарантирует, что данные будут извлекаться с той скоростью, с какой они могут быть получены. Так как в данном случае пользователи могут получать данные быстрее, чем в случае доступа по телефонной линии, для каждого из них задействуется большая часть доступной полосы пропускания. Скорость доступа по телефонной линии, как правило, составляет 44 Кбит в секунду. Если принять в расчет служебные PPP-сигналы, извлечение данных с вашего сайта будет, скорее всего, происходить со скоростью

¹ Считается как киловатт-часы при оплате электричества: избыточная скорость передачи данных умножается на количество минут, в течение которых наблюдается данная скорость.

всего 38 или 39 Кбит в секунду. У некоторых пользователей кабельных модемов скорость загрузки может достигать 6 Мбит в секунду. Простой расчет показывает, что вы можете обслужить в тринадцать раз больше пользователей с доступом по коммутируемой телефонной линии, чем пользователей кабельных модемов.

Когда большинство ваших пользователей посещает ваш сайт, находясь на работе, картина получается примерно такой же, как в случае домашних пользователей с широкополосным Интернетом. Корпоративные сети, как правило, ограничивают компьютеры своих сотрудников полосой пропускания в 10 Мбит в секунду (хотя в последние годы этот показатель постепенно растет до 100 Мбит в секунду). Даже при наличии в фирме магистрального канала Ethernet, как правило, устанавливаются ограничения, мешающие слишком интенсивно загружающим информацию пользователям монополизировать сеть. Кроме того, корпоративные фаерволы и прокси-серверы в самое удобное время обычно всегда загружены. Поэтому в момент, когда вероятность попадания пользователя на ваш сайт максимальна (например, во время обеденного перерыва), скорость передачи данных замедлена из-за перегруженного прокси-сервера. И в результате корпоративные пользователи получают примерно такую же полосу пропускания, как и пользователи DSL или кабельных модемов.

Оценивая полосу пропускания, мы, как и в случае с процессором и хранилищем данных, столкнемся с эффектом умножения. Динамически генерируемые страницы, как правило, несут в себе много лишнего. Предположим, что каждая страница содержит хотя бы 1024 байт мусорной информации. Просмотр миллиона страниц в день превращается в пересылку 1 024 000 000 избыточных байтов. Это чуть менее одного *гигабайта* ненужных пересылок. А большинство страниц содержит куда больше, чем 1024 ненужных байтов. Технику создания страниц, позволяющую избежать таких потерь, мы обсудим в следующей главе.

8.6. Заключение

Управление вычислительной мощностью представляет собой непрерывный процесс мониторинга и оптимизации. Приходится учитывать различные аспекты и движущие силы, направленные порой в противоположные стороны. Меняется программное обеспечение, меняется трафик, и даже маркетинговые кампании иногда способны привести к тому, что на мощность вашей системы начнет влиять некая сторонняя сила. Для регулирования мощности требуется видеть работу системы в целом. Чрезмерно упрощенные или линейные модели ее оценки могут увести вас в неверном направлении, что станет причиной избыточных расходов или упущенной выгоды фирмы.

По сути, вычислительная мощность является мерой того, какой доход система может принести за определенный период времени. Поэтому принятые на этапе проектирования неверные решения, снижающие мощность, непосредственно уменьшают прибыли компании. Для устранения последствий этих неверных решений требуются

дополнительные капитальные вложения и постоянные операционные издержки. Чтобы извлечь из инвестиций максимум, всегда придерживайтесь перечисленных далее правил.

- ☐ Обращайте внимание на эффект умножения. Именно он главным образом сказывается на стоимости.
- ☐ Понимайте, какое влияние один слой оказывает на другой.
- ☐ Помните, что улучшение показателей, не связанных с ограничителями, никак не влияет на вычислительную мощность.
- ☐ Пытайтесь сделать основную часть работы заблаговременно, пока в ней не возникла острая необходимость.
- ☐ Устанавливайте максимальные защитные границы для всего: таймаутов, потребления памяти и т. п.
- ☐ Защищайте программные потоки, отвечающие за обработку запросов.
- ☐ Постоянно контролируйте вычислительную мощность. Каждая новая версия приложения может повлиять на масштабируемость и производительность. Изменения пользовательского спроса или моделей трафика меняют нагрузку на систему.

9 Антипаттерны вычислительной мощности

Надеюсь, вы уже убедились, что вычислительная мощность процессора, дисковое пространство и полоса пропускания стоят довольно дорого, а значит, нужно извлекать из этих ресурсов максимум. В этой главе мы рассмотрим антипаттерны, негативно влияющие на вычислительную мощность вашей системы.

Эти антипаттерны заставляют ваши приложения выполнять ненужную работу, превращая электричество в тепло, а не в деньги. Аппаратное обеспечение стоит дорого, поэтому нужно постараться получить от него все, что только возможно.

9.1. Конкуренция в пуле ресурсов



К пулам соединений с базами данных я отношусь одновременно с любовью и ненавистью. Я смотрю на них как на необходимое зло. Так как установка нового соединения с базой данных занимает до 250 миллисекунд, имеет смысл по возможности использовать соединения многократно. При правильном подходе к ним пулы соединений, как и все пулы ресурсов, позволяют увеличить вычислительную мощность за счет повышения пропускной способности. В то же самое время оставленные без присмотра пулы ресурсов могут быстро стать одним из самых проблемных мест приложения.

Узкое место возникает при наличии конкуренции за ресурсы, когда некоего ресурса не хватает для всех требующих его потоков. В этом случае большинство пулов

соединений просто блокирует запрашивающий поток на неопределенное время, до момента доступности ресурса. Такая ситуация явно не повышает пропускную способность. Рисунок 24 демонстрирует, сколько процентов от процессорного времени потоки, использующие пулы разного размера, проводят в конкуренции за пулы. В каждом случае конкуренция отсутствует, пока количество потоков не превышает количества доступных ресурсов. Самая верхняя кривая демонстрирует, что происходит, когда потоки борются всего за четыре подключения к базе данных. К моменту, когда количество потоков достигает тридцати, более 80 % процессорного времени тратится на бесполезное ожидание освободившегося соединения.

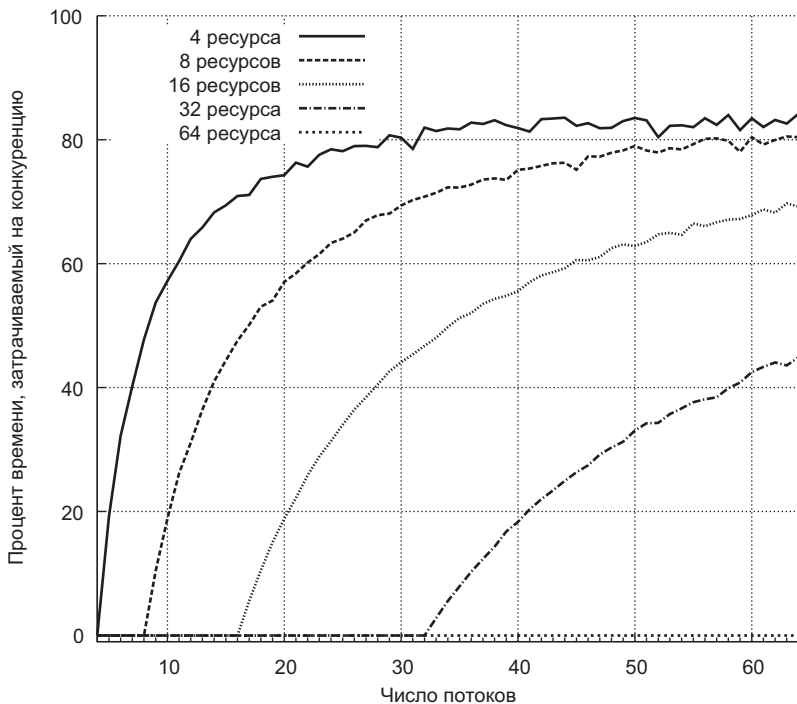


Рис. 24. Конкуренция среди потоков

Если большую часть своего времени потоки, занимающиеся обработкой запросов, проводят в заблокированном состоянии, очевидно, что свою работу они не выполняют. И в самом деле, рис. 25 демонстрирует пропускную способность (в «заданиях в минуту») разного количества потоков, обрабатывающих запросы, с пулами ресурсов разного размера. Линии представляют эти самые пулы. В левой части графика все они являются более-менее линейными и имеют почти одинаковый наклон. Но когда количество потоков превышает количество ресурсов, линия начинает приближаться к горизонтальной. Это так называемое плато, знакомое любому, кто

когда-либо проводил нагрузочные испытания. До известной степени после начала конкуренции за ресурсы увеличение числа потоков, занимающихся обработкой запросов, перестает влиять на пропускную способность.

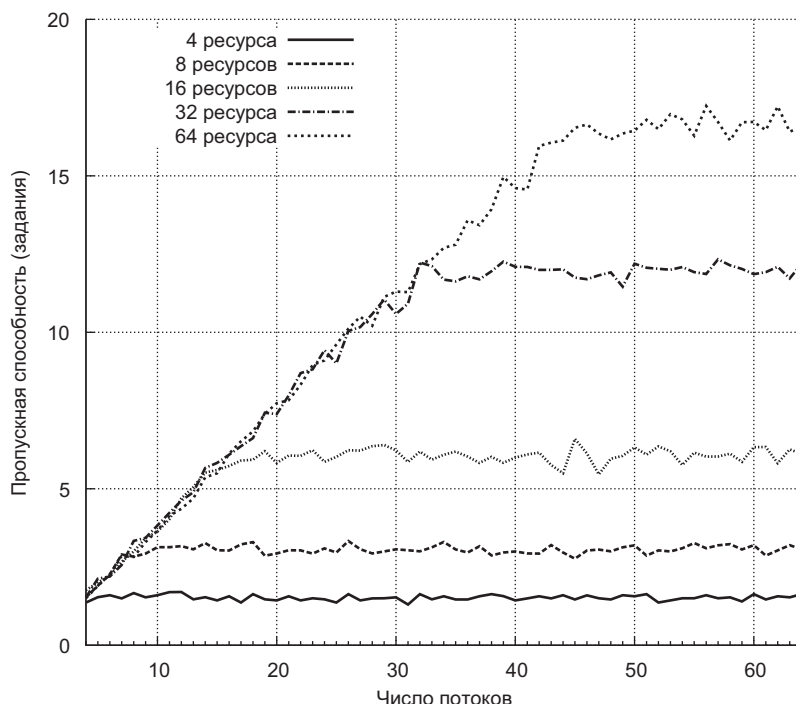


Рис. 25. Влияние конкуренции на пропускную способность

В идеале каждый поток немедленно получает необходимый ему ресурс. Чтобы это гарантировать, сделайте размер пула ресурсов равным числу потоков. Это ослабит конкуренцию на сервере приложений, но может привести к возникновению узкого места на сервере базы данных. К примеру, в Oracle каждое подключение порождает процесс на сервере базы данных. В зависимости от конфигурации этот процесс может использовать несколько мегабайтов оперативной памяти. Если серверов приложения несколько, сервер базы данных рискует стать жертвой очередного эффекта умножения.

К примеру, в случае фермы серверов на двадцать машин, на каждой из которых запускается пять экземпляров сервера приложений, а каждый экземпляр пользуется пулом из пятидесяти соединений с базой данных, сервер этой базы должен поддерживать 5000 соединений. Даже если каждое из них задействует всего 1 Мбайт оперативной памяти, у сервера базы данных должно быть 5 Гбайт RAM только для обслуживания этих соединений. Хуже всего то, что некоторые из соединений

большую часть времени будут находиться в состоянии простоя. (Будь они все заняты 100% времени, база данных превратилась бы в ограничитель.) Другие базы данных не порождают процессы для каждого соединения¹, но все равно обязаны выделять для него некоторые ресурсы.

Блокировка на неопределенный срок при отсутствии доступных ресурсов гарантирует проблему стабильности (см. раздел 4.5). Система будет лучше обслуживаться, если настроить пул ресурсов только на временную блокировку. Каждый сервер приложений конфигурируется по-своему, поэтому имеет смысл досконально изучить поведение вашего конкретного пула ресурсов². Если после истечения времени ожидания доступный ресурс не появится, пул должен вернуть значение `null` или выбросить исключение. Код приложения должен быть готов к такому повороту событий.

Все классы общего пула ресурсов в той или иной степени страдают от недостатка прозрачности. Во время выполнения хорошо бы знать, насколько часто блокируются вызывающие потоки, какое максимальное количество ресурсов было выгружено с момента начала работы, сколько ресурсов было создано и уничтожено. Все эти показатели могут выдвинуть на передний план проблемы вычислительной мощности. Серверы приложений JBoss и WebLogic предоставляют доступ к части этой информации через JMX-расширения, а дальше вы самостоятельно можете периодически собирать указанные показатели для автономного анализа.

ЗАПОМНИТЕ

Исключите конкуренцию при нормальной нагрузке

Во время регулярных нагрузочных всплесков не должно возникать конкуренции за ресурсы. Регулярные нагрузочные всплески — это явление, не привязанное к пиковому для вашей компании сезону.

По возможности выбирайте размер пула ресурсов в соответствии с размером пула потоков, обрабатывающих запросы

Если у вас доступный ресурс есть всегда, когда он требуется обрабатывающему запросу потоку, вы избежите снижения производительности из-за непроизводительных издержек. В случае подключений к базе данных дополнительные подключения в основном потребляют оперативную память на сервере базы данных, что, конечно, обходится дорого, но куда дешевле упущенной выгоды. При этом имеет смысл следить за тем, чтобы один сервер базы данных мог справиться с максимальным числом подключений.

¹ Oracle также поддерживает более сложные конфигурации многопоточного сервера (MTS) с мультиплексными подключениями. Администраторы баз данных, способные корректно настроить MTS, находятся на вершине пирамиды жрецов Oracle.

² Класс `BasicDataSource` из проекта Jakarta Commons поддерживает этот режим через свойство `maxWait`. Сервер JBoss использует элемент `<blocking-timeout-millis>` в конфигурационных файлах своего источника данных. Более подробную информацию вы найдете в сопроводительной документации.

В случае аварийного переключения один из узлов кластера баз данных должен будет обслуживать все запросы — и все подключения.

Разорвите порочный круг

Конкуренция за ресурсы увеличивает время транзакций. Более медленные транзакции усиливают конкуренцию за ресурсы. При наличии нескольких пулов ресурсов этот цикл может вызвать экспоненциальное падение пропускной способности, когда время отклика начнет возрастать.

Обращайте внимание на паттерн блокированных потоков

Проблема вычислительной мощности, вызванная конкуренцией в пуле ресурсов, может быстро превратиться в проблему стабильности, если потоки при отсутствии доступных ресурсов блокируются навсегда.

9.2. Избыточные JSP-фрагменты



В царстве Java язык JSP (Java Server Pages — серверные Java-страницы) представляет собой стандартный язык создания шаблонов страниц. JSP-файлы компилируются в два прохода. Сначала сервер приложений генерирует файл `.java` с характерным именно для него кодом сервлета. После чего этот файл с текстом программы сервер приложений компилирует в машинно независимый код. Далее новый файл класса загружается в виртуальную Java-машину. Как все Java-классы, скомпилированные JSP-файлы загружаются в область памяти виртуальной Java-машины, которая называется областью несменяемого поколения (permanent generation). Назначение этой области понятно из ее названия — объектами несменяемого поколения являются определения классов и методов.

Серверы J2EE-приложений почти всегда используют загрузочные сценарии с JVM-аргументом `-noclassgc`. Именно он указывает, что не следует выгружать классы из несменяемого поколения. Считается, что это повышает производительность, и порой это действительно так. Проблемы начинаются, когда во время одного рабочего цикла сервера приложений загружается большое число JSP-классов. В такой ситуации заданного по умолчанию размера области несменяемого поколения может не хватить. Если виртуальная Java-машина может загружать произвольное количество JSP-файлов, значит, размер несменяемого поколения, которое может потребоваться во время работы, ничем не ограничен. Так как все JSP-файлы компилируются и загружаются в область несменяемого поколения при условии, что сборщик мусора игнорирует классы, в какой-то момент эта область памяти окажется переполненной. Для спасения ситуации в этом случае нужно убрать аргумент `-noclassgc`, предоставив сборщику мусора возможность удалять не используемые более классы. По сути, вместо значительного падения производительности, почти неотличимого от отказа, вы получите сниженную, но не фатальную производительность.

JSP-ФАЙЛЫ КАК КОНТЕНТ

Вот ситуация, которая возникает, когда JSP-файлы рассматриваются как контент, а не как код. Мне пришлось работать с сайтом, содержащим более 25 000 JSP-фрагментов. Подавляющее большинство из них представляли собой фрагменты контента, относящиеся к раскрутке товара или страницам с перечислением категорий. К сожалению, ни один из этих файлов не устаревал. И каждый день в область несменяемого поколения загружались все новые и новые JSP-файлы в виде классов. Это привело к серьезным проблемам со сборкой мусора, потому что виртуальной машине становилось все сложнее искать место под новые классы в области несменяемого поколения. (Такая ситуация проявляется особенно остро, если вы аргументом `-XX:MaxPermSize` ограничили размер кучи). В данном случае JSP-файлы даже не относились к исполняемому коду. Они содержали только статический контент. И для управления этими 25 000 фрагментами куда лучше было воспользоваться обычным HTML-кодом с хранилищем для кэшированного контента. (Другие предостережения, касающиеся корректного кэширования, вы найдете в разделе 10.2.)

ЗАПОМНИТЕ

Не используйте код в качестве контента

Загрузка JSP-классов в память представляет собой своего рода кэширование. Большое количество JSP-файлов вполне может переполнить область несменяемого поколения. И даже если переполнения не произойдет, зачем напрасно тратить память, которую можно было бы использовать для решения полезных задач, а вместо этого вы храните там классы, обращение к которым происходит только в момент перезапуска сервера приложений.

9.3. Перебор с AJAX



Технология AJAX (Asynchronous JavaScript plus XML — асинхронный JavaScript-код плюс XML) стала синонимом сайтов «Web 2.0», таких как Google Maps и del.icio.us. AJAX-приложение встраивает в свои веб-страницы JavaScript-код, отправляющий в фоновом режиме запросы к сайту, пока пользователь занят другими делами. Ответ сервера позволяет обновлять части текущей страницы, в то время как в «Web 1.0» увидеть любое обновление можно было только после повторной загрузки.

Хотя технология AJAX известна уже несколько лет, в Google выдвинули ее на первый план только в проектах Gmail и Google Maps. Эти приложения привлекли всеобщее внимание тем, что в них отпала необходимость в некоторых вещах. Почтовая служба Gmail умеет без перезагрузки страницы автоматически подставлять адреса, а также разворачивать и сворачивать отдельные ветви переписки. Служба Google Maps идет еще дальше. Без перезагрузки страницы можно выполнять панорамирование и изменение масштаба страницы. Новые фрагменты карты подгружаются автоматически и появляются на экране сразу же, как только становятся доступными.

Однако, к сожалению, далеко не все обладают серверными фермами такого же размера, как Google. Для многих сайтов AJAX-приложения означают намного большее

количество запросов от браузера к веб-серверам. К тому же они еще и появляются куда чаще. Время между щелчками среднего пользователя на обычном сайте составляет от пяти до десяти секунд. Часть этого периода занимает время ожидания пользователем окончания загрузки и визуализации страницы. В случае же сайта на базе технологии AJAX время между HTTP-запросами в большинстве случаев составляет от одной до трех секунд. Индивидуальные запросы, как правило, имеют меньший размер, а так как в ответ обычно посылается только фрагмент страницы, размер ответа также меньше. Совокупный эффект во многом зависит от того, каким образом вы реализуете технологию AJAX. При корректном использовании вы можете снизить издержки на полосу пропускания. В противном случае AJAX дополнительно нагружает уровни веб-сервера и сервера приложений.

Более полное описание технологии AJAX выходит за рамки темы данной книги (интересующимся я рекомендую книгу *Pragmatic Ajax* Джастина Гетланда), но так как эта технология представляет собой обоюдоострый меч, хотелось бы рассказать о том, как ее применять, чтобы не порезаться.

Проектирование взаимодействия

AJAX — это технология достижения цели, а не сама цель. На первое место должно выходить удобство пользователя, поэтому применяйте AJAX только там, где эта технология действительно облегчает взаимодействие с пользователем. Например, я видел сайт, построенный на базе «главной AJAX-страницы». Это была единственная страница, содержавшая весь HTML-код. Все взаимодействия были реализованы через AJAX-запросы, а их результаты отображались на этой же странице. Это было ужасно. Казалось, что сайт сломал браузер! Кнопка **Назад (Back)** не возвращала вас на предыдущую страницу, а заставляла уйти с сайта. А хуже всего то, что обновлялась страница лишь через некоторое время после действий пользователя, причем это время зависело от сложности запроса. В результате создавалось впечатление, что страница меняется случайным образом без какой-либо обратной связи с браузером.

Применять AJAX лучше всего для реализации взаимодействий, которые мозг пользователя представляет как единую задачу. В случае почтовой службы Gmail такой задачей является, например, «отправка почты». Это может быть любое многоступенчатое взаимодействие, для завершения которого в обычном случае потребовалось бы несколько страниц. Если у вас есть «макет» сайта, обратите внимание на линейные цепочки страниц, которые в конечном счете возвращаются на главную или на любую другую страницу.

Время выполнения запроса

В ряде ранних руководств по AJAX-библиотекам демонстрировались запросы автоматического заполнения, посылаемые каждую четверть секунды, что, несомненно, увеличивало загрузку серверов сайта. Более новые библиотеки поддерживают

настраиваемую задержку, позволяющую отправлять запросы на автоматическое заполнение через определенные промежутки времени (обычно 500-миллисекундные) после прекращения пользовательского ввода. Разумеется, это задержка перед *отправкой* запроса. Время ожидания пользователя складывается из нее, а также из задержки, связанной с отправкой запроса, обработкой его сервером и получением ответа.

Смена сеанса

Привязка сеанса должна быть настроена таким образом, чтобы AJAX-запросы уходили на тот же сервер приложений, на котором находится пользовательский сеанс. Избегайте ненужных переключений сеансов.

Форматирование ответа

Не стоит возвращать ни HTML-страницы, ни их фрагменты. HTML-код является слишком подробным; передавая его, вы впустую занимаете полосу пропускания. Вместо этого возвращайте только данные без форматирования, чтобы клиент смог динамически обновлять элементы страницы.

Пользуйтесь для данных объектной JavaScript-нотацией (JavaScript Object Notation, JSON)¹, а не XML-кодом. Да, я помню, что в самом названии технологии есть упоминание XML, но это сделано только потому, что изначальный вариант «AJAS» показался менее звучным. Формат JSON намного проще анализируется в браузере и более лаконичен, что уменьшает потребление полосы пропускания. Примеры того, насколько кратким является JSON в сравнении с XML, можно найти на странице <http://www.json.org/example.html>.

ОБРАТНАЯ СТОРОНА ФОРМАТА JSON

Следует помнить об одной особенности формата JSON: в некоторых примерах для обработки JSON-строк используется JavaScript-функция `eval()`, в результате чего объекты оказываются непосредственно в сфере действия интерпретатора. Это крайне плохая идея! Даже если для проверки сайта в браузере вы пользуетесь протоколом SSL и надежными сертификатами, как вы сможете проверить код приложения на сайте? Достаточно всего одного злонамеренного программиста, который напишет вредоносный сценарий в формате JSON для запуска на компьютере пользователя. Такой сценарий может сделать немало плохого, от пересылки третьим лицам конфиденциальной информации до отправки фальшивых заказов. Разумеется, «хакер»-разработчик в состоянии причинить ущерб и другими способами, но зачем самостоятельно давать ему в руки такую простую возможность навредить пользователям?

¹ См. <http://json.org/json-ru.html>.

ЗАПОМНИТЕ**Избегайте ненужных запросов**

Не нужно посылать запросы в режиме опроса для таких функциональных возможностей, как, к примеру, автоматическое заполнение. Если вам необходимо ею воспользоваться, отправляйте запрос только после изменения поля ввода. (Мне встречались учебники, согласно которым запросы отправлялись каждые четверть секунды!)

Учитывайте архитектуру сеанса

Убедитесь, что ваши AJAX-запросы содержат cookie-файл с идентификатором сеанса или параметром запроса. (Первое реализуется намного проще второго.) В противном случае ваш сервер приложений будет открывать новый (совершенно ненужный) сеанс для каждого AJAX-запроса.

Минимизируйте размер ответов

Возвращайте минимально необходимое количество данных. Отправляйте ответ в формате XML или JSON, но никак не в виде HTML-кода.

Увеличьте размер вашего веб-звена

Вашим веб-серверам придется иметь дело с возрастающим количеством запросов. Обязательно увеличьте максимальное число соединений, с которыми может справиться ваше звено веб-узлов, добавив серверы или увеличив количество клиентов на каждом сервере¹.

9.4. Затянувшиеся сеансы



В незапамятные времена, когда появилась первая Java-спецификация Servlet, было решено, что по умолчанию время ожидания сеанса должно составлять 30 минут. Именно столько сеанс остается в памяти *после* последнего запроса пользователя. Как отмечалось в главе 7, пользователи на самом деле уходят в начале периода ожидания. Проблема в том, что предсказать, какой из запросов пользователя окажется последним, нельзя. (А это не предвещает ничего хорошего, не так ли?) Наверняка мы знаем только то, что сеанс остается в памяти после ухода пользователя.

Активный сеанс потребляет ресурсы, главным образом память. Серверы Java-приложений всегда имеют ограниченную память. Избыток свободной памяти оказывает критическое влияние на стабильность и производительность Java-приложений. Поэтому застрявшие в памяти сеансы представляют прямую угрозу здоровью и благополучию системы. Степень их угрозы прямо пропорциональна времени пребывания в памяти.

Ваша система должна стремиться избавиться от этих сеансов при первой же возможности. Общепринятый таймаут, по умолчанию равный 30 минутам, — явный

¹ В Apache за это отвечает конфигурационный параметр `MaxClients`.

перебор. Проанализируйте образцы трафика вашего сайта, чтобы определить среднее и стандартное отклонение от времени между запросами страниц, которое все еще можно отнести к сеансу. Другими словами, два посещения в один день с промежутком в несколько часов не относятся к одной и той же последовательности действий. Аналогично, два визита одного пользователя, начинающиеся с главной страницы, совершенные с промежутком в час, *скорее всего*, не входят в один сеанс. В то же время два запроса, совершенные с промежутком в 29 минут, с одной внутренней страницы на другую внутреннюю страницу, скорее всего, являются проявлением активности одного и того же пользователя. Хорошим вариантом является взятие в качестве таймаута для сеанса одного стандартного отклонения после среднего времени задержки. На практике для интернет-магазина это будет примерно 10 минут, 5 минут — для медиашлюза и до 20 минут — для сайтов, связанных с туристическим бизнесом.

Задайте для сеанса таймаут как одно стандартное отклонение, следующее за средне-статистическим временем обдумывания.

Еще лучше сделать так, чтобы необходимость в сеансе отпала. Если сеанс включает в себя набор временных состояний, которые уничтожаются после ухода пользователя, имеет смысл хранить его некоторое время. Если же все элементы сеанса представляют собой копию в памяти устойчивого состояния, ничто не мешает в любой момент уничтожить и заново создать сеанс. При таком подходе сеанс — это не более чем сохраняемый в памяти кэш. Я настоятельно рекомендую именно эту модель, так как она выгодна с точки зрения как вычислительной мощности и стабильности, так и пользователя. Только разработчики понимают, зачем нужны сеансы. Пользователи не любят навязываемых сеансами ограничений. Возвращение в исходную точку только потому, что начался показ очередной серии сериала, не может не раздражать. Пользователи хотят уходить и возвращаться по своему желанию, попадая в ту самую точку, на которой они закончили работу.

Это означает, что таких понятий, как, к примеру, временная корзина, быть не должно, а результаты выполненного пользователем поиска должны быть доступны и в будущем, каким бы способом они ни были получены — при помощи поисковой службы или обычным запросом к базе данных. Идентификатор и предпочтения пользователя должны оставаться активными. (Однако пользователю нужна возможность как-то отделять себя от используемого *устройства*, например от терминала в библиотеке или аэропорту.)

Единственным исключением являются связанные с финансами конфиденциальные данные, например номер кредитной карты или номер социального страхования¹.

¹ Номер социального страхования в США является эквивалентом нашего номера паспорта. — *Примеч. пер.*

ЗАПОМНИТЕ**Сокращайте время хранения сеанса**

Храните сеансы в памяти минимально необходимое время. В разных доменах оно будет различаться.

Помните, что пользователи не понимают, зачем нужен сеанс

Пользователи понимают, что автоматический выход связан с безопасностью. Но они никогда не смогут понять, почему их корзина исчезла после того, как они провели двадцать минут, разглядывая товары на другом сайте. Вещи не должны исчезать только потому, что пользователь на некоторое время отвлекся.

Сохраняйте ключи, а не целые объекты

Сохраняя в сеансе целые объекты, используйте мягкие ссылки. Но лучше вместо этого храните ключи к объектам длительного хранения. Вычислительная мощность такой системы выше, а для пользователя она удобнее: пользователь может вернуться через час или через неделю и оказаться в том же месте, где он закончил работу с сайтом.

9.5. Напрасный расход пространства HTML-страницами



Я видел много плохих вещей, в оправдание которых выдвигался аргумент «полоса пропускания стоит дешево» и «процессоры стоят дешево». Как вы могли убедиться в разделе 8.5, ни одно из этих утверждений не является правдой. Неэффективность нельзя оправдывать ничем.

Веб-приложения скрывают в себе массу неэффективных решений, от которых каждый день страдают пользователи. Представьте, что размер страницы можно уменьшить с 200 до 150 Кбайт, — кстати, не такая уж редкая ситуация. А это на 25 % меньше байтов, которые придется загружать пользователю. При подключении по среднестатистической телефонной линии загрузка этих 50 Кбайт занимает до десяти секунд. Влияет оно и на пользователей широкополосных соединений. И вот каким образом.

На генерацию этих 50 Кбайт ненужной информации сервер приложений тратит циклы процессора. Проходя через сетевую карту сервера, они занимают полосу пропускания. Они идут в сетевой коммутатор, для них выполняется вычисление контрольной суммы, затем буферизация, маршрутизация, повторное вычисление контрольной суммы и десериализация на каком-то другом порту коммутатора. Этот порт может быть соединен с фаерволом — специальным маршрутизатором, который повторяет перечисленные сетевые операции, добавляя к ним проверку безопасности. Эти 50 Кбайт возвращаются в коммутатор — может быть, тот же самый, а может быть, какой-то другой, проходят через все ту же сетевую низкоуровневую канитель и направляются на веб-сервер. Последний буферизует эти 50 Кбайт в памяти вместе

со значимой частью страницы. Если каждый запрос будет использовать 200 Кбайт памяти веб-сервера вместо 150 Кбайт, серверу для обработки запросов потребуется на 33 % больше оперативной памяти. После того как генерация страницы завершится, веб-сервер отправит ее в браузер... через еще один фаервол, коммутатор и по меньшей мере один маршрутизатор.

Эти 50 Кбайт требуют много внимания! На каждом шаге они используют либо дополнительную память, либо более широкую, чем нужно, полосу пропускания.

Есть и еще один эффект. Чем больше страница, тем дольше браузер и веб-сервер поддерживают соединение. А пока соединение открыто, ни один другой запрос не может им воспользоваться. Но веб-сервер может поддерживать только конечное число соединений! Аналогично тому, как потоки конкурируют за подключения к базе данных (см. раздел 9.1), конечные пользователи могут конкурировать за подключения к веб-серверу. А без этого пользователь не может попасть на сайт.

Поэтому в ваших интересах, чтобы все соединения обслуживались и освобождались с максимально возможной скоростью.

Но если влияние «раздутых» HTML-страниц так пагубно, кто и зачем их делает? Разумеется, намеренно их не делает никто. Все происходит автоматически. Дизайнеры создают стильные макеты страниц в приложении Dreamweaver. Программисты вставляют их в ASP-, JSP- или RHTML-шаблоны. А сервер приложений генерирует из этих шаблонов страницы. Неэффективность может вкрасься на каждом шаге. Рассмотрим наиболее частые случаи.

Пустое пространство

Веб-приложения строятся из фрагментов страниц, будь это JSP- и JHTML-страницы, плитки или файлы `.rhtml`. Эти фрагменты облегчают жизнь программистам, позволяя многократно использовать разные HTML-элементы и отделяя их от структуры страниц. Вот только объединение этих фрагментов друг с другом порой дает в результате чрезмерно большие страницы.

Проще всего занять место пустым пространством. Все нестандартные теги, используемые в языках написания шаблонов, заменяются сгенерированным контентом, а вот пустое пространство, применявшееся при форматировании файла, сохраняется. Сюда входят символы новой строки после тегов шаблона, даже когда тег «раскрывается» в никуда, как, к примеру, условный тег при невыполненном условии.

Я видел совершенно вопиющий пример сайта, главная страница которого состояла из более чем сотни JSP-фрагментов. Сгенерированный HTML-код занимал более 600 Кбайт, треть из которых состояла целиком из символов новой строки, полной пробелов.

Это звучит банально, но представьте себе пользователя с доступом по телефонной линии, который дополнительно пять секунд ожидает загрузки набора пробелов! На систему это тоже сильно влияет. Пробелы при каждом запросе страницы «съедают» дополнительно 200 Кбайт на веб-сервере, так как ответная страница буферизуется перед отправкой ее в браузер. Поскольку не процессоры, а веб-серверы, как правило, сталкиваются с ограничениями памяти, это уменьшает общую вычислительную мощность сайта. Избыточное пустое место может стоить компании более 15 000 долларов в год только за счет оплаты полосы пропускания и без учета стоимости дополнительных веб-серверов.

Пустое пространство может обойтись в 15 000 долларов в год.

К серверу приложений можно добавить перехватчик, который будет отфильтровывать избыточное пустое пространство. Я часто видел, как затраты процессорного времени на фильтрацию оказывались меньше затрат на оперативную память и полосу пропускания при отсутствии фильтрации.

Дорогие изображения-разделители

Практически любая динамически генерируемая HTML-страница имеет одно или несколько изображений-разделителей. Эти маленькие прозрачные GIF-файлы, или однопиксельные изображения (иногда называемые *прокладками*) различных цветов, вездесущи. И как оказалось, они плохо влияют на полосу пропускания и удобство пользователя.

Предположим, у вас есть ячейка таблицы, которая для реализации задуманной компоновки должна иметь определенный размер. Вы можете задать ширину и высоту самой ячейки, но зачастую в нее в качестве контента вставляется изображение-разделитель. Например:

```

```

Это 53 байта ссылки на изображение. Их можно заменить вот этим:

```
&nbsp;
```

Эти 5 байт, вставленные вместо 53, экономят 48 байт. Это число не кажется большим, но следует помнить, что стандартная страница содержит от 12 до 40 разделителей в разных местах.

Не забудьте учесть эффект умножения. Сорок восемь байтов, сохраненные в 12 местах страницы, умноженные на миллион запросов страницы в день, превращаются в 576 000 000 байт. Заметят ли ваши коммутаторы и маршрутизаторы дополнительные 576 000 000 байт ежедневного трафика?

Для пользователя каждое изображение-разделитель (не ссылка со страницы, а реальный URL-адрес) означает дополнительный запрос к серверу. Даже если браузер пользователя уже имеет в кэше это изображение, он зачастую интересуется: «Изображение, которое у меня есть, еще актуально?» Эта ситуация особенно распространена в случае динамически генерируемых страниц, когда браузер получает указание не кэшировать контент страницы.

Каждый из этих запросов изображения-разделителя или проверки его статуса в кэше на некоторое время требуют подключения к веб-серверу. И сервер в тысячный раз отправляет изображение-разделитель, хотя в это время мог бы выполнять приносящую доход транзакцию другого пользователя.

Избыток HTML-таблиц

Раз уж мы занялись уменьшением «веса» HTML-страниц, посмотрим, каким образом осуществляется форматирование при помощи таблиц. Все современные версии браузеров поддерживают каскадные таблицы стилей (Cascading Style Sheets, CSS). Разумное применение CSS и стилей обеспечивает такой же полный контроль над форматированием, как и HTML-таблицы. Но разница в вычислительной мощности при этом получается весьма впечатляющей. Табличные структуры необходимо загружать при каждой обработке страницы, а таблицу стилей достаточно загрузить один раз.

Даже если таблица стилей размером превышает обычную HTML-страницу, куда важнее чистая экономия. Я видел результаты сравнения страниц с CSS- и HTML-форматированием в случае, когда CSS-версия занимала менее половины HTML-версии. Фантастические примеры дизайна, достижимого исключительно средствами CSS и HTML, можно увидеть на сайте <http://www.csszengarden.com/>. Каждый раз, заходя на этот сайт, я нахожу по меньшей мере один вариант дизайна, заставляющий меня воскликнуть: «Я не понимаю, как такое удалось!»

ЗАПОМНИТЕ

Пропускайте ненужные символы

Убирайте лишние символы из HTML-кода. Их генерация занимает циклы процессора на сервере приложений. Они занимают полосу пропускания на сетевых картах серверов приложений, коммутаторах, веб-серверах и в пользовательских соединениях. Их загрузка требует времени, особенно в случае медленного соединения. Удаляйте пустое пространство, чтобы сэкономить время пользователей и деньги вашей фирмы.

Убирайте пустое пространство

Пустое пространство проникает на сгенерированные страницы в местах, где есть циклы, условия и включения. Директива порой заменяется строкой нулевой длины, а прекрасное форматирование, созданное кодером, остается. Мне доводилось видеть страницы, на которых знаки табуляции, пробелов и переносов строк занимали до 400 Кбайт.

Заменяйте изображения-разделители неразрывными пробелами или CSS-стилями

Изображения-разделители, оставляемые дизайнерскими программами на странице, напоминают мышинный помет. Они маленькие, но вредоносные. Превращая дизайн страницы в код, убирайте все такие изображения.

Заменяйте HTML-таблицы CSS-макетами

CSS-стили не только больше соответствуют «Web 2.0», но и не требуют загрузки для каждой страницы в отличие от табличных структур. HTML-файлы со стилями и классами зачастую оказываются намного меньше, чем их аналоги на основе HTML-таблиц.

9.6. Кнопка обновления страницы



Клянусь, иногда действия пользователя — это худшее, что может случиться с системой. Эта фраза оказывается как никогда актуальной, если система уже находится под нагрузкой. Представьте, что сайт, которым вы пользуетесь, начинает «тормозить». Если это J2EE-сайт, то какой-то пользователь мог инициировать запуск полного цикла сборки мусора, получив пятнадцатисекундное время отклика. (Разумеется, я пишу это в ироническом смысле, потому что в большинстве случаев сборщик мусора запускается автоматически после неудачного выделения памяти, но именно это практически гарантированно происходит во время пользовательского запроса.) Сидящий перед браузером пользователь понятия не имеет, сколько работы приходится проделывать на другом конце сети. И если в течение десяти секунд нужная страница не появится, он, скорее всего, щелкнет на кнопке обновления страницы. После этого браузер бросит уже имеющееся соединение, откроет новый сокет и пошлет новый HTTP-запрос.

Серверу приложений никто не говорит, что обработку предыдущего запроса следует прекратить. Более того, если линия связи между веб-сервером и сервером приложений буферизует ответы, о том, что предыдущее соединение больше не нужно, можно будет узнать только после того, как в памяти веб-сервера окажется сформирована и буферизована целая страница. В лучшем случае сервер приложений при попытке отправить все еще далекий от идеала ответ получит исключение `IOException`.

Если запрос страницы вызывает выполнение некой транзакции, например обновление пользовательского профиля или фиксацию каких-то данных мониторинга, второй запрос может оказаться заблокированным в ожидании завершения первого. В самом плохом случае оба запроса окажутся в состоянии взаимной блокировки.

Корректного ответа на вопрос, что делать с кнопкой обновления страницы, не существует. Поэтому просто постарайтесь гарантировать достаточно быструю реакцию сайта, чтобы у пользователей не возникало соблазна ее нажимать.

СЕРИАЛИЗАЦИЯ НА ИСХОДНОМ АДРЕСЕ

Я встречал серверы приложений, пытающиеся гарантировать обработку всего одного запроса за раз с одного IP-адреса отправителя. Это не получается по двум причинам. Во-первых, если вы пользуетесь сетью кэширования, например Akamai, все запросы будут приходить с одного IP-адреса. Такая ситуация возникает и у пользователей корпоративных сетей с единым «шлюзом» прокси-сервера. Во-вторых, сразу же после щелчка на кнопке обновления страницы браузер пользователя прекращает ждать ответ на первый запрос. Сериализация запросов только усугубит проблему, так как в этом случае второй запрос пользователя не начнет обрабатываться, пока не завершится первый. Это означает, что раздраженный пользователь будет смотреть на значок загрузки, а не на загрузившуюся страницу, после чего перед тем, как уйти на сайт конкурента, еще два или три раза щелкнет на кнопке обновления страницы.

В зависимости от выбранного вами способа балансировки нагрузки и параметров сеанса первый и второй запросы могут вообще отправиться на разные серверы приложений. Соответственно, код вашего приложения должен быть готов к ситуации, когда один и тот же пользователь выполняет одну транзакцию несколько раз, и не допускать состояния взаимной блокировки.

ЗАПОМНИТЕ

Сделайте ненужной кнопку обновления страницы

На быстро реагирующем сайте у пользователя не возникает желания воспользоваться кнопкой обновления страницы. Нужно обеспечить такую скорость работы сайта, чтобы пользователи даже не вспоминали про эту кнопку. Запросы на перезагрузку перегружают и без того перегруженный сайт.

9.7. Кустарный SQL-код



По сути, любое современное приложение должно работать с реляционной базой данных. Для заполнения этого пробела многие приложения используют объектно-реляционное проецирование (Object-Relational Mapping, ORM). Неважно, какой технологией вы пользуетесь — ActiveRecord в Ruby on Rails, Hibernate или любой из трех EJB-моделей сохранения, — там всегда в той или иной форме будет задействовано это проецирование. ORM-пакеты генерируют очень предсказуемый, повторяющийся SQL-код. Предсказуемые паттерны SQL-кода запросов доступа благотворно влияют на вычислительную мощность, так как квалифицированный администратор баз данных может настроить свою базу таким образом, что эти запросы начнут работать хорошо¹.

¹ Однако работая с ORM-инструментами, будьте осторожны с проблемой $N+1$ запроса. Она возникает при доступе к членам коллекции. Некоторые фреймворки ORM-разработки посылают один запрос для определения принадлежности к коллекции, а затем по одному запросу к каждому члену для заполнения отдельных объектов.

Каждый из упомянутых ORM-уровней тем или иным способом позволяет разработчику получить доступ непосредственно к SQL-коду. Например, Hibernate дает возможность встраивать именованные SQL-запросы непосредственно в файл проекции, а затем из вашего приложения ссылаться на них по имени. Даже если речь идет о получении доступа к соединению с базой данных и непосредственной отправке запроса, вы всегда можете отправить кустарный SQL-код прямо в базу данных. Обычно эта возможность описывается как способ повышения производительности, почему же я называю ее убийцей вычислительной мощности? Увы, зачастую разработчики, занимающиеся объектно-ориентированным программированием, вытворяют с несчастной базой данной странные, удивительные и даже садистские вещи.

Проблема в том, что, как правило, самостоятельно написанный SQL-код оказывается своеобразным и непредсказуемым. Настройка базы данных для остальной части приложения не срабатывает при наличии этих странных, уникальных фрагментов кода и может даже повредить их. Аналогичным образом попытки подстроиться под эти фрагменты не принесут ничего хорошего остальной части приложения и могут повредить его.

Чем же так плохи эти кустарные SQL-запросы? Как правило, в них встречается масса распространенных ошибок. Во-первых, зачастую они присоединяют неиндексированные столбцы. Во-вторых, они связывают слишком много таблиц. В случае простого соотношения разработчики, как правило, работают с ORM-пакетами. А раз дело дошло до самостоятельно написанного SQL-кода, значит, речь явно идет о каких-то необычных паттернах доступа. В-третьих, разработчики пытаются трактовать язык SQL как императивный или объектно-ориентированный, а не как реляционный язык для работы с наборами данных, коим он и является. Они создают запрос, связывающий пять таблиц, чтобы выбрать единственную нужную им строку, а затем выполняют этот запрос еще сто раз для поиска других отдельных строк. В-четвертых, разработчики любят упражняться с SQL-функциями, которых лучше вообще не касаться. Однажды я видел объединение восьми запросов, в котором каждая подвыборка требовала связывания пяти таблиц по неиндексированным столбцам. В распечатанном виде этот код занимал целую страницу. В плане запроса фигурировало примерно сорок сканирований таблицы.

Сканирование таблицы (table scan) — самый медленный способ поиска информации в большой таблице. Сервер базы данных циклически просматривает все строки таблицы в поисках совпадений.

Влияние правильно настроенной базы данных на работу системы настолько велико, что порой в это трудно поверить. У каждого администратора баз данных есть истории, когда добавление какого-либо индекса или анализ статистических свойств таблиц сокращали время некоего процесса с восемнадцати часов до трех

минут. Повышение скорости работы на порядок встречается повсеместно. Разработчики с подобными эффектами не сталкиваются, так как на стадии разработки и тестирования имеют дело с нереалистично маленькими наборами данных. Для выяснения производительности требуются реальные объемы данных. Иногда можно воспользоваться данными существующей рабочей системы. Однако для использования в нерабочей среде этим данным может потребоваться «очистка». Как правило, она заключается в замене конфиденциальных данных случайным набором символов, чтобы защитить личную информацию клиентов, особенно в случае, когда тестовые данные уходят за пределы компании. Но в случае разработки системы «с нуля» таких наборов данных может попросту не оказаться под рукой. В этом случае имейте в виду, что трата пары часов рабочего времени на создание генератора данных в будущем окупится многократно.

Кустарный SQL-код близко связан с динамически генерируемым SQL-кодом. Сюда не относится код, динамически создаваемый ORM-инструментами, вполне приемлемый в силу своей предсказуемости. Я хочу предупредить вас о другой разновидности динамически генерируемого кода. Именно он является причиной моих бессонных ночей. Это код, в котором, например, некоторые фрагменты выполняют запрос по образцу путем циклического просмотра атрибутов, то и дело прибегая к инструкции `WHERE`. Или код, в котором динамически собираются таблицы и результаты их объединения. Оставьте такие вещи для системы учета. В системах транзакций их лучше не применять.

Каждый такой запрос может столкнуться с крайне медленно реализуемым условием. Настройка быстрой реакции базы данных возможна только в случае определенного набора предсказуемых паттернов доступа. А обеспечить быструю реакцию системы на любой допустимый запрос попросту невозможно.

ЗАПОМНИТЕ

Минимизируйте количество кустарного SQL-кода

У вас может быть большой соблазн воспользоваться собственным SQL-кодом для оптимизации производительности. В подобных случаях всегда лучше посмотреть, нельзя ли реализовать задуманное средствами самой базы — с помощью всплывающих подсказок, индексирования или представлений.

Обращайте внимание, смеется ли администратор базы данных над запросами

Если тест на смех не проходит, в эксплуатацию это пускать не следует. Точка.

Проверяйте достигнутые результаты на реальных данных

Проверяйте, как написанный вами SQL-код работает с объемом данных, типичным для этапа эксплуатации. Улучшения, наблюдаемые в базе данных на этапе разработки, порой нивелируются при переходе к совершенно другому плану выполнения запросов, характерному для эксплуатационных условий.

9.8. Эвтрофикация базы данных



В экологии процесс «старения» водоемов называется эвтрофикацией (eutrophication). Это медленное наращивание слоя ила из мертвых микроорганизмов, рыбы и водорослей. На поздней стадии из-за этого возникает дефицит кислорода и жизнь в водоеме становится невозможной. Водоем умирает. С вашей базой данных может произойти то же самое, только кверху брюхом в этом случае всплывет ваша система.

На этапах разработки и тестирования эксперименты, как правило, проводятся с наборами данных, размер которых варьируется от небольшого до совсем незначительного. В сочетании с коротким временем, которое обычно выделяется на тестирование, база данных может быть передана в эксплуатацию без проверки, как она справляется с большими объемами данных. И выяснять, что с ней происходит в такой ситуации, приходится уже на практике.

Индексирование

Инструменты объектно-реляционного проектирования (ORM) позволяют легко создавать и отслеживать соотношения между таблицами. Заданные в созданном ORM-инструментами файле связи не заставляют администратора базы данных прибегать к редактированию. Соответственно, в то время как правила ссылочной целостности, заданные архитектором данных, напоминают фиксацию индекса в любом соотношении по внешнему ключу, таблицы, связанные через свойства объектов, скорее всего, не потребуют столь же пристального внимания. Выборка строки, то есть отслеживание объектных отношений, по неиндексированному столбцу приводит к ужасному «сканированию таблицы». Суть процесса понятна по его названию. Эти именно линейный поиск по всем строкам в поисках строк, столбцы которых совпадают с запросом.

Размер набора данных, который обычно используется на стадии разработки, не позволяет даже измерить разницу между запросом по индексированным и по неиндексированным столбцам. Более того, в случае «до смешного маленького» набора данных сканирование таблицы может оказаться самым быстрым способом выполнения запроса! Это особенно верно в типичных для маленьких наборов данных случаях, когда таблица уже находится в кэше. А через год-другой эксплуатации вдруг оказывается, что пользователям приходится по несколько минут ждать завершения самых простых операций. В общем случае индексироваться должны все столбцы, которые являются объектом объединения в ORM-представлении.

Схемы баз данных часто проектируются задолго до кода приложения, в котором они будут использоваться. В результате появившиеся на этапе проектирования индексы могут не совпасть с реальными паттернами доступа, реализованными в приложении. Человек, отвечающий за архитектуру базы данных, должен принимать

участие в разработке приложения, чтобы гарантировать постоянную подстройку схемы под функциональность приложения и ее максимальную эффективность. Особенно это актуально для гибкой методологии разработки, когда связи между элементами возникают и исчезают намного чаще, чем в традиционном подходе.

Секционирование

Следующий ключевой вопрос разработки касается возможности поддержания структуры базы данных в процессе эксплуатации. Архитекторы баз данных всегда хотят по возможности точно знать, как быстро будут расти и сколько раз пересматриваться все таблицы. Эта информация помогает им расположить таблицы на диске. Корректность проецирования логических таблиц в физической памяти оказывает долговременное влияние на производительность базы данных. К сожалению, зачастую сделанные на ранних этапах оценки темпов роста и сроков хранения оказываются далекими от истины. Иногда ситуация коренным образом меняется при переходе к новой версии. К примеру, я встречал систему управления заказами, которая генерировала 1 Гбайт файлов регистрации действий в год, а после перехода к новой версии файл такого размера стал появляться *ежедневно*. В подобных ситуациях схема размещения базы данных в памяти «превращается в лапшу», потому что таблицы распределяются по разным физическим связанным областям. При интенсивном дисковом вводе-выводе время отклика значительно возрастает. Одним из решений является получение до начала разработки приложения полной информации о будущей базе с детальными характеристиками.

Другим решением является секционирование, или разделение, таблиц. Если в каком-то столбце присутствует небольшой конечный набор значений, каждое из которых означает кластер связанных строк, этот столбец становится хорошим кандидатом для секционирования. К примеру, таблица, которая хранит заказы в процессе их обработки, будет часто обновляться. Добавлением столбца «день недели» ее можно разбить на семь секций. Каждая секция может быть отдельно реорганизована или перемещена в другую область памяти, даже если другая секция в это время интенсивно используется. Это позволяет менять структуру выделенной под базу данных памяти, не прекращая с ней работать.

Данные за прошедшие периоды

Лучшей реакцией на замедление скорости работы, связанное с ростом количества данных в базе, является радикальная процедура архивирования или удаления. Мы имеем дело с транзакционными системами. Это значит, что нам требуются только те данные, которые нужны для обработки пользовательских транзакций. Поэтому нет никакой нужды хранить данные за прошедшие периоды в той же самой базе данных. В частности, составление отчетов и специальный анализ никогда не должны проводиться на работающей базе данных. (Подобную нехорошую практику можно

прекратить, сделав так, что в результатах будут выводиться данные только за последние девяносто дней или за полгода!)

OLTP (Online Transaction Processing — обработка транзакций в реальном времени) — это схема, оптимизированная для быстрой вставки транзакционных записей. Как правило, она плохо подходит для составления отчетов и выполнения специальных запросов.

Извлечение информации из массивов данных, составление отчетов и любой анализ в любом случае должны проводиться только в хранилище данных. Транзакционная OLTP-система в качестве хранилища не подходит. Не нужно валить в одну кучу транзакции и отчеты.

Не смешивайте транзакции и составление отчетов.

Пользователям может потребоваться увидеть данные за прошедшие периоды. К примеру, на сайте Amazon я могу посмотреть историю своих заказов за почти десять лет. (Мне интересно, сколько десятилетий они планируют хранить эту информацию?) Насколько часто может требоваться информация о заказе, сделанном весной 1998-го? Многоуровневая система хранения позволяет держать в Интернете данные за прошедшие периоды средствами недорогой системы, в то время как большинство текущих транзакций реализуется средствами высокопроизводительной (и дорогостоящей) системы.

В этом разделе я лишь слегка коснулся архитектуры систем хранения данных. Это тема для целой серии книг. Но вы должны усвоить главное: жесткий режим очистки данных жизненно важен для долговременной стабильности и производительности вашей системы.

ЗАПОМНИТЕ

Создавайте индексы; это задача не только администратора базы данных

Вы знаете свое приложение и его предназначение лучше, чем администратор базы данных. Вы должны знать, какие столбцы будут использоваться для поиска, какие таблицы будут читаться чаще всего, а в какие чаще всего будет осуществляться запись. Поэтому именно вы должны предоставить первый набор индексов.

Удаляйте ненужную информацию

Старые данные только замедляют скорость выполнения запросов и добавление строк. Если только эти данные не важны пользователю — как, скажем, таблица с историей его заказов, — удаляйте их с рабочих серверов.

Вынесите средства составления отчетов за пределы рабочей системы

Отчеты могут и должны создаваться в другом месте. Не ставьте под угрозу производственные операции, выполняя запросы для составления отчетов, требующие большого количества ресурсов. Тем более, что для этого схема звезды подходит куда больше, чем OLTP.

9.9. Задержка в точках интеграции



Любое взаимодействие с другой системой происходит с определенной задержкой. Удаленный вызов совершается по меньшей мере в 1000 раз медленнее, чем локальный. Каким бы способом он ни осуществлялся — через DCOM, CORBA, веб-службы или бинарный протокол сокета, — вызывающей стороне приходится некоторое время ждать ответа. Вызывающая сторона должна обработать некие транзакции, в противном случае обращений к ней не возникало бы. Соответственно, следует помнить, что вызывающей стороне для ответа требуется как минимум столько же времени, сколько удаленной системе.

Задержка в точках интеграции может стать серьезной проблемой производительности. Она становится особенно острой, когда точка интеграции использует какой-то вид протокола удаленного объекта. Философия «независимости удаленных объектов от их местоположения» требует, чтобы вызывающая сторона не видела разницы между локальным и удаленным вызовами. Но эта философия дискредитировала себя по двум основным причинам. Во-первых, режимы отказа удаленных и локальных вызовов различаются. Первые уязвимы для сетевых отказов, отказов удаленных процессов и несовпадения версий вызывающей стороны и сервера — и это только первое, что приходит на ум. Кроме того, требование независимости от местоположения заставляет разработчиков создавать интерфейсы удаленных объектов таким же способом, как и локальных, в результате интерфейс получается слишком «разговорчивым». В таких проектных решениях для одного взаимодействия используются вызовы нескольких методов. Каждый вызов вносит свой вклад в задержку. В результате время реакции системы становится слишком большим.

Проблемы производительности, с которыми сталкиваются отдельные пользователи, превращаются в проблемы вычислительной мощности системы в целом. Даже если обрабатывающий транзакцию поток временно простаивает в ожидании ответа, он все равно удерживает множество ресурсов. Он потребляет память и кванты процессорного времени. Еще он может удерживать соединение с базой данных, которое требуется другим потокам. Больше того, поток в режиме ожидания может блокировать строку или страницу в базе данных, вызывая на этом уровне конкуренцию.

Фактически, заставляя заблокированный поток ждать ответа от точки интеграции, мы многое теряем. Другая работа выполняться не может, даже если она поставлена в очередь и ждет, когда поток ею займется.

С RMI ЧЕРЕЗ ОКЕАН

Однажды я работал с толстым клиентом Java-системы, который для общения с сервером пользовался RMI-интерфейсом. Сервер обеспечивал объектно-ориентированное представление клиентского корпоративного хранилища данных, обладающее хорошими функциональными возможностями. Клиентское приложение позволяло маркетологам формировать связи между продуктами из иерархии и различными носителями информации, например изображениями, PDF-файлами, текстовыми документами.

Внутри страны система работала прекрасно. Но как только появились пользователи из Великобритании, начались требования многомиллионных инвестиций в локальный

сервер и хранилище данных. И оказалось, что открытие одного узла в иерархии (элементе управления деревьями), на которое в США уходило меньше секунды, у британских пользователей занимало двадцать минут!

Открытие узла требует от клиента поиска некоторых сведений о дочерних узлах: их имена, типы, число их потомков. Проблема была в том, что когда клиент удаленно вызывал родительский узел, запрашивая у него коллекцию потомков, это приводило к перечислению списка и трехкратному вызову каждого дочернего узла. Прекрасный пример проблемы «1+N»: к единственному вызову списка добавляются один или несколько вызовов всех элементов этого списка.

Мы добавили к родителю метод, возвращавший коллекцию «итоговых объектов», которая содержала необходимые для управления деревом три бита информации. После внедрения этого обновления пользователи в Великобритании тоже смогли насладиться мгновенным временем реакции, не устанавливая целое хранилище данных!

ЗАПОМНИТЕ

Старайтесь как можно реже иметь дело с задержками

Задержка в точках интеграции напоминает ситуацию с казино и игроком — казино всегда имеет преимущество. Чем чаще вы играете, тем чаще это работает против вас. Избегайте «общительных» удаленных протоколов. Они намного медленнее работают и на больший период связывают бесценные потоки, отвечающие за обработку запросов.

9.10. Cookie-монстры



Есть технологии, которые так и хочется использовать не по назначению. Возьмем петарды. Сколько бы предостережений производители на них ни наклеивали, это все равно фейерверк с *ручкой*. Спросите у Дона Норманна — ручки нужны для того, чтобы за них дергать¹. Можно взять человека, который никогда в жизни не видел петарду, и уже через десять минут он будет держать эту штуку в руке, направив ее в товарищей. Это не самое лучшее поведение воспроизводится в каждом следующем поколении.

Файлы cookie наравне с петардами относятся к «штукам, приглашающим их взорвать». Есть моменты, когда их применение вполне оправданно, но варианты злоупотреблений заставляют изумленно выпучить глаза. На первый взгляд файлы cookie кажутся безвредным, хотя и неизящным способом обойти тот факт, что веб-взаимодействия не хранят данных о состоянии. Сервер отправляет дополнительный заголовок или два с данными, которые в какой-то будущий момент клиент должен прислать назад.

Авторы документа RFC 2109² недвусмысленно написали, что файлы cookie предназначены для управления сеансами. Этот документ был озаглавлен «Механизм

¹ См. в книге *The Design of Everyday Things* про «воспринимаемые свойства».

² См. <http://www.ietf.org/rfc/rfc2109.txt>.

управления HTTP-состоянием». К их чести, Дэвид Кристал и Лу Монтулли не просто добавили новый HTTP-заголовок для идентификатора сеанса. Их изобретение имеет более широкое применение. Им можно воспользоваться не только для идентификации сеанса, но и для многого другого. (Отдавая должное Тиму Бернерсу Ли, следует упомянуть, что именно он добился расширения HTTP посредством заголовков и потребовал, чтобы агенты игнорировали непонятные им заголовки, а не сообщали об ошибке.)

Несколько разработчиков независимо друг от друга обнаружили антипаттерн хранения анонимных постоянных данных посредством файлов cookie. Например, одна группа добавила к процессу обработки запроса перехватчик, который использовал Java-сериализацию для сохранения в виде файлов cookie корзин анонимных пользователей. Это было нужно, чтобы не создавать в базе данных записей для анонимных посетителей, которые могли никогда не вернуться на сайт. А благодаря файлу cookie в случае возвращения такой посетитель видел в своей корзине все, что было туда положено при предыдущем визите.

Реализация этой идеи была непростой; сложно было понять, с чего начинать. Во-первых, Java-сериализация имеет ряд особенностей. Браузеры хранят файлы cookie до истечения срока их действия (или до обновления компьютера, установки другого браузера, удаления сохраненных браузером личных данных и т. п.). В результате возраст сериализованной формы объекта-корзины может измеряться месяцами или даже годами. И велики шансы, что изменения кода сделают эту форму недействительной задолго до возвращения пользователя. Ошибки в десериализации Java-объектов приводят к выбрасыванию исключений `IOException`, что большинство разработчиков считает крайне маловероятным событием. Сериализация превращает эти исключения в рутинную часть рабочего процесса.

И даже при неизменности исходного кода продукты в корзине пользователя могут поменяться или даже исчезнуть. То есть перехватчик должен еще и обрабатывать проверку целостности ссылочных данных, а также разбираться с версиями кода.

А что делать, если браузер попросту *лжет*? Как и любой другой протокол, HTTP представляет собой лишь соглашение о способе взаимодействия сторон. И любая из сторон может оказаться злоумышленником. Что помешает злонамеренному пользователю поменять файлы cookie и установить на все товары цену 0,01 доллара? Опытный злоумышленник, вооруженный информацией о программной платформе компании, в состоянии использовать сериализованный объект `Cart` для изменения правил ценообразования и проведения рекламных акций. Файлы cookie — это всего лишь данные, верить им не стоит! Многочисленные инструменты позволяют пользователям мошенничать с HTTP-запросами и HTTP-ответами на них (расширения Firefox, умные HTTP-прокси, воспроизведение сеансов и т. п.).

Кроме того, встает вопрос о вычислительной мощности. Изначально файлы cookie предназначались для пересылки маленьких фрагментов данных, не более 100 байт. Для идентификации сеанса этого вполне достаточно. При таком малом объеме данных тот факт, что браузер посылает файл cookie при каждом HTTP-запросе, не

имел особого значения. А теперь представьте 4 Кбайт сериализованного объекта или фрагмента XML и подумайте, как это повлияет на полосу пропускания. А ведь это по большей части бесполезные издержки. Сериализованная корзина десериализуется всего один раз. После этого она оказывается в памяти пользовательского сеанса. Пересылаемые туда и обратно при каждом запросе данные являются неоправданной тратой полосы пропускания.

После получения запроса веб-сервер должен преобразовать все заголовки в собственную внутреннюю структуру. При наличии сервера приложений веб-сервер отправляет ему параметры запроса. В общей сложности дополнительные файлы cookie прогоняются по проводам дважды (а иногда и четыре раза) и дважды анализируются. Разумеется, дополнительное время на выполнение одного запроса может оказаться не таким уж большим, но не стоит забывать об эффекте умножения.

На пользователях это тоже отражается. Даже у тех из них, кто сидит на широкополосном соединении, как правило, пропускная способность восходящего канала намного меньше, чем нисходящего. Загрузка дополнительных 4 Кбайт при каждом запросе начинает накапливаться. (Да, несколько миллисекунд тут, несколько миллисекунд там, и вот речь уже идет о реальном времени!)

Все эти дополнительные издержки обработки выливаются в увеличение стоимости эксплуатации. Дополнительный код для обработки файлов cookie требует постоянных затрат на обслуживание, и все только для того, чтобы избежать добавления к базе данных строк с корзинами анонимных посетителей. Единственным аргументом против помещения данных о корзинах в базу является стоимость и сложность работы по периодической очистке базы от неиспользуемых данных.

Тем не менее в качестве общего механизма файлы cookie позволяют реализовывать интересные вещи. Достаточно помнить, что клиент может солгать, отправить обратно устаревшие или недействительные файлы cookie или не отправить их вовсе.

ЗАПОМНИТЕ

Используйте компактные файлы cookie

Пользуйтесь файлами cookie для идентификаторов, а не для целых объектов. Храните данные сеанса на сервере, где их не сможет изменить злоумышленник.

9.11. Заключение

Статистика утверждает, что большинство программистов никогда не работали с реально сложным, критически важным программным обеспечением. Во-первых, исследования заработной платы постоянно показывают, что опыт большинства программистов не превышает десяти лет. По достижении десятилетнего рубежа

многие уходят на руководящую работу или просто прекращают заниматься программированием. Во-вторых, гистограмма размеров проектов имеет значительный уклон в сторону меньшего конца шкалы. Поэтому берите на крупные проекты представителей молодого поколения с относительно небольшим опытом и не удивляйтесь тому, как сложно найти специалиста нужного уровня.

Без должного опыта программисты по невежеству или движимые неверными намерениями, скорее всего, столкнутся со многими рассмотренными в этой главе убийцами вычислительной мощности. Эти вопросы, разумеется, не рассматриваются в колледжах и университетах. *Оптимизация* там сводится к настройке некоторых алгоритмов поиска.

Убийцы мощности выглядят как явные ошибки, особенно после того, как извлекаются из системы и предъявляются для анализа. Тем не менее я видел, как их снова и снова делают разные команды на разных клиентах. При этом никто не выбирает дизайн, намеренно уменьшающий вычислительную мощность системы; нет, просто при выборе функционального дизайна никто не обращает внимания на то, какое влияние он оказывает на вычислительную мощность.

10 Паттерны вычислительной МОЩНОСТИ

Сэр Чарльз Энтони Ричард Хоар, как известно, сказал: «Преждевременная оптимизация — корень всех зол». Эту фразу часто используют не по назначению, объясняя ею небрежность проектирования. Полная цитата Хоара гласит: «О небольших выгодах примерно в 97 % случаев следует забыть: преждевременная оптимизация — корень всех зол». Она предупреждает, что не стоит гнаться за небольшими улучшениями, платя за это возрастающей сложностью и увеличивающимся временем разработки.

Проблема в том, что при плотном расписании (а когда оно не бывает плотным?) оптимизация делается на поздних сроках, что зачастую означает ее полное отсутствие. Более того, оптимизация может на какой-то процент повысить производительность отдельных процедур, но не способна дать фундаментально более удачное проектное решение. В процессе оптимизации невозможно перейти от пузырьковой к быстрой сортировке. Выбор лучшего проектного решения или оптимизированной под эффекты масштабирования архитектуры является действием, диаметрально противоположным преждевременной оптимизации; такой выбор вообще устраняет необходимость в оптимизации.

Кроме того, проектируя свою распределенную систему, вы на свой страх и риск игнорируете вопросы производительности и вычислительной мощности. Как я показал в разделе 8.5, небольшие изменения в расходе процессорного времени, памяти и места на диске после перехода к эксплуатации системы умножаются на достаточно большие коэффициенты. Я знаю случай, когда плохо написанный код

заставил организацию внести в смету десять миллионов долларов на закупку дополнительного аппаратного обеспечения, которое позволило бы фирме продержаться, пока не кончится сезон отпусков. Устранение ряда антипаттернов и реализация нескольких паттернов вычислительной мощности¹ избавила ее от таких трат, равных примерно недельному доходу от продаж.

Устранение антипаттернов — важный шаг, но можно сделать кое-что еще. В этой главе мы поговорим о паттернах вычислительной мощности, которые переведут ваши приложения из любительской в профессиональную лигу.

10.1. Организация пула соединений



Как упоминалось в разделе 9.1, пулы ресурсов кардинально увеличивают вычислительную мощность. Пулы ресурсов сокращают время установки соединения. Для нового соединения с базой данных требуется подключение по протоколу TCP, проверка прав доступа к базе и настройка сеанса. Все вместе это может занять от 400 до 500 миллисекунд. Но начало работы нового программного потока требует больше ресурсов, чем установление соединения с базой данных.

Когда CGI-сценарии на языке Perl только появились, такой сценарий мог открывать соединение, выполнять определенную работу, а затем разрывать связь. Это была безопасная, четкая и простая в отладке стратегия. К сожалению, этот подход исчерпал свои возможности, когда серверы баз данных начали тратить на управление соединениями столько же времени, сколько на обработку транзакций. Чтобы обойти это ограничение, отдельные запросы к страницам должны иметь общие, или многократно используемые, соединения. Сейчас практически любой язык и стиль программирования разрешают организацию пула соединений. И избегать данного подхода можно разве что в случае, если вы не знаете, как это делать.

При организации пула соединений следует учесть некоторые факторы. Состояние соединений может оказаться некорректным. В этом случае любой запрос, попытавшийся воспользоваться соединением, получит сообщение об ошибке. Некорректное соединение будет выгружено, после чего оно сгенерирует ошибку и вернется в пул. Корректные соединения служат для выполнения реальной работы, поэтому они дольше находятся в выгруженном состоянии. В результате при возникновении запроса вероятность, что доступным окажется именно некорректное соединение, выше. Такая ситуация вызывает непропорционально большое число ошибок. Одно некорректное соединение из десяти приводит к генерации ошибки в более чем 10 % запросов.

¹ Обсчитывайте контент на стадии компиляции и аккуратно используйте кэширование, когда оно имеет место.

Жизненно важным вопросом является выбор размера пула соединений. Недостаточный размер пула приводит к возникновению конкуренции (это явление мы рассматривали в разделе 9.1). Избыточный размер пула может стать причиной избыточной нагрузки на серверы базы данных.

С этим близко связан выбор проектного решения, выполняющего проверку соединений в пуле и вне его. Для систем на базе веб-технологий существует несколько возможных решений. Самой простой является «постраничная» модель: соединение выгружается для целой страницы. И возвращается обратно, когда работа со страницей завершается.

Если пулов соединений несколько, эта модель становится средством борьбы с взаимной блокировкой, ведь для всех запросов можно принудительно установить один и тот же порядок выгрузки и возвращения потоков. В то же время при этом требуется более высокое соотношение числа соединений и числа потоков, обрабатывающих запросы, так как период выгрузки каждого соединения будет дольше.

«Постраничная» модель, как правило, внедряется, когда диспетчер транзакций настроен на создание единой транзакции, обрабатывающей страницу целиком. При страницах, динамически создаваемых из набора фрагментов, применение этой модели не всегда возможно. «Пофрагментный» подход позволяет каждому фрагменту выгрузить собственное соединение, сделать какую-то работу и вернуть соединение в пул. Эта модель более уязвима для взаимной блокировки, но позволяет достичь более высокой пропускной способности, чем в случае «постраничной» модели. Требуется меньшее число соединений на один поток обработки запроса, потому что отдельные соединения быстрее возвращаются в пул. Основным преимуществом является тот факт, что отдельным фрагментам не требуется глобальных сведений о контексте транзакции. Каждый из них в состоянии работать независимо.

Гибридный подход позволяет каждому фрагменту управлять собственными соединениями, но при этом для страницы в целом создается транзакция базы данных. Именно к ней подсоединяются все соединения до своего завершения или отмены. В данном случае меры предосторожности против взаимной блокировки из «постраничной» модели сочетаются с простотой и изолированностью «пофрагментной» модели. Правда, в этой ситуации требуется больший пул соединений, чем при постраничном подходе. Гибридная модель порой сложнее поддается отладке, так как отдельные фрагменты могут видеть незафиксированные данные, введенные в транзакцию предшествующими фрагментами. В результате получается, что фрагмент видит данные, которые не видны вам. Диагностировать поведение фрагмента в такой ситуации крайне сложно.

Но какая бы стратегия ни использовалась в вашем приложении, вы должны следить за тем, чтобы в пуле соединений не возникало конкуренции, иначе вместо увеличения вычислительной мощности вы ее быстро убьете.

ЗАПОМНИТЕ**Организируйте пул соединений**

Пул соединений — это база. И нет никаких причин от нее отказываться.

Защищайте программные потоки обработки запросов

Не позволяйте навсегда блокировать вызывающие потоки. Убедитесь, что все вызовы, приводящие к выгрузке соединений, снабжены таймаутами, а вызывающая сторона знает, что делать, когда соединение невозможно вернуть в пул.

Выбирайте размер пула, обеспечивающий максимальную пропускную способность

Пулы ресурсов слишком маленького размера ведут к конкуренции и увеличению задержки. В первую очередь это противоречит цели пула соединений. Следите за обращениями к пулу соединений, чтобы понять, сколько времени ваши потоки ждут выгрузки соединения.

10.2. Будьте осторожны с кэшированием



Кэширование может стать мощным лекарством для лечения проблем производительности. Оно снижает нагрузку на сервер базы данных и серьезно ускоряет реакцию системы. Однако неверное применение этого механизма может привести к новым проблемам.

Максимальное использование памяти всех кэшей прикладного уровня должно быть настраиваемой величиной. Кэши, не ограничивающие максимальное потребление памяти, в конечном счете съедают всю память в системе. После этого сборщик мусора начинает тратить все больше времени, пытаясь высвободить достаточный для обработки запросов объем памяти. Кэш, который потребляет необходимую для выполнения других заданий память, серьезно замедляет скорость работы системы.

Сколько бы памяти ни было выделено под кэш, нужно следить за уровнем обращений к кэшированным элементам. При небольшом количестве обращений кэш не дает никакого выигрыша в производительности, более того, без него система может работать быстрее. Храня какой-то элемент в кэше, вы предполагаете, что совокупные затраты на его однократную генерацию вместе с затратами на хэширование и поиск будут меньше затрат на генерацию этого элемента каждый раз, когда в нем возникнет необходимость. Если какой-то кэшированный объект за время жизни сервера используется всего один раз, смысла в его кэшировании нет.

Также стоит избегать кэширования элементов, генерация которых не требует особых затрат. Я выдывал кэши контента с сотнями элементов, состоящих из единственного пробела. Определенный JSP-фрагмент был снабжен условной инструкцией, проверявшей, является ли человек сотрудником фирмы; это делалось

путем поиска логического флага в его профиле. В большинстве случаев результат проверки оказывался отрицательным, и фрагмент визуализировал себя как пустое пространство. В какой-то момент я стал сомневаться, что затраты на эту условную конструкцию достаточно велики, чтобы оправдать кэширование результата, особенно при условии, что кэшированный объект имеет отношение всего к одному пользователю.

В языке Java для построения кэшей используются объекты `SoftReference`, в которых, собственно, и хранятся кэшированные элементы. При нехватке памяти сборщик мусора получает позволение убрать любой объект, достижимый только посредством мягких ссылок. В результате кэши, использующие мягкие ссылки, помогают сборщику мусора освобождать память, вместо того чтобы мешать этому процессу.

В экстремальных случаях возникает необходимость перехода к многоуровневому кэшированию. При таком подходе данные, обращение к которым осуществляется чаще всего, хранятся в памяти, а для вторичного кэша используется дисковое пространство. Это хорошо работает, когда в кэше оказываются экстремально большие объекты (например, изображения) или рабочий набор слишком велик, чтобы держать его в памяти. Помогает многоуровневый кэш и в ситуациях, когда для загрузки неэкшированных данных требуется доступ через WAN-соединение.

Предварительное вычисление результатов позволяет уменьшить потребность в кэшировании или вообще от него отказаться.

Наконец, в любом кэше могут оказаться устаревшие данные. Поэтому каждому кэшу нужна индивидуальная стратегия удаления элементов, источник данных которых изменился. Выбор этой стратегии может радикально повлиять на вычислительную мощность вашей системы. К примеру, прямое уведомление хорошо работает, если у вас есть десять-двенадцать серверов приложений. Если же их количество исчисляется сотнями, это становится уже неэффективным, и лучше подумать об очереди сообщений или групповой рассылке. В последнем случае, разумеется, нужно позаботиться о том, чтобы все серверы приложений не бросались на базу данных одновременно в попытке перезагрузить ставший недействительным элемент.

ЗАПОМНИТЕ

Ограничивайте размеры кэша

Неограниченный кэш потребляет память, которую лучше пустить на обработку запросов. Хранение в памяти всех когда-либо загруженных элементов не приносит пользователям никакой выгоды.

Создайте механизм очистки

Рано или поздно, через определенное время или из-за каких-то событий в сети, любой кэш начинает нуждаться в очистке. Но это процедура зачастую требует изрядных ресурсов, поэтому ограничьте частоту проведения, чтобы собственноручно не спровоцировать системный сбой.

Не кэшируйте тривиальные объекты

Далеко не все доменные объекты и HTML-фрагменты стоят того, чтобы их кэшировали. Редко используемым, небольшим или не требующим затрат на свою генерацию объектам в кэше не место: издержки, связанные с их хранением и уменьшением объема свободной памяти, перевесят выигрыш в производительности.

Сравнивайте частоту обращений и изменений

Не имеет смысла кэшировать объекты, которые, скорее всего, изменятся до того, как пользователь снова к ним обратится.

10.3. Предварительное вычисление контента



Все мы, архитекторы и проектировщики приложений, любим динамический контент. Во-первых, он намного интереснее статического. Во-вторых, для создания статической страницы никто не будет нанимать программиста. Мы даже уничижительно называем такие сайты *брошюрными* (brochureware). Как только мы видим в списке требований, что контент сайта может в любой момент измениться, мы немедленно переходим к динамически генерируемым сайтам на основе базы данных. По этому пути нас ведут все современные технологии: JSP, ASP, Ruby on Rails и пр.¹

Проблема, как обычно, возникает из-за эффекта умножения. Представьте типичный сайт магазина. Практически на всех таких сайтах посетители могут просматривать иерархическую структуру категорий продуктов. Категории могут быть представлены в виде дерева с одним корнем, набора деревьев или направленного ациклического графа, но в любом случае они приведут пользователя на страницу с информацией об интересующем его продукте. На каждой странице видны по крайней мере категории высшего уровня. Обычно также демонстрируется второй уровень категорий. Как правило, для генерации меню категорий делается запрос к базе данных, а затем категории для фрагмента страницы обрабатываются в цикле, чтобы для каждой из них визуализировался свой HTML- или JavaScript-код. И этот фрагмент кода выполняется, возможно, миллион раз в день. А насколько часто меняются категории верхнего уровня? Раз в три месяца? Небольшие изменения могут совершаться раз в неделю. Но код все равно выполняет запрос и динамически генерирует HTML-код из-за мизерного шанса, что за те десять миллисекунд, которые прошли с момента генерации предыдущей версии страницы, что-то могло измениться. Даже если поместить результаты запроса базы данных в кэш, визуализация HTML-кода все равно займет изрядное время из-за огромного количества задействованных в ней строк.

А зачем вообще тратить время на визуализацию HTML-кода? Если можно выделить разделы сайта, контент которых меняется намного реже, чем генерируются страницы,

¹ Я уже молчу про Struts, Tiles, Tapestry, WebWork, WebObjects, SpringMVC и т. п.

имеет смысл рассчитать все визуализируемые HTML-фрагменты заранее. Это особенно ценно в случаях, когда вы точно знаете, в какой момент происходит изменение контента. В этот момент достаточно обновить заранее подготовленный контент и снова пустить его в работу, а не обсчитывать один и тот же HTML-код миллион раз.

Сайты новостных порталов, такие как Slashdot и Fark, заранее обсчитывают свои главные страницы. Фигурирующие на них истории меняются по меньшей мере раз в час, иногда чуть чаще. А общее количество комментариев меняется каждую минуту. Тем не менее главная страница этих сайтов представляет собой шаблон, содержащий изрядные куски заранее обсчитанного контента. Их «живость» обеспечивается повторным обчетом контента каждые несколько минут, но каждая версия страницы до следующего обновления оказывается просмотренной сотни и даже тысячи раз.

МАСТЕР НЕНОРМАТИВНОЙ ЛЕКСИКИ

Один из вопиющих примеров чрезмерного применения динамического контента попался мне на глаза во время описанного в главе 7 запуска сайта торговой компании. Анализ статистики сборщика мусора показал, что каждый запрос страницы приводит к появлению почти 10 Мбайт мусора. Нет, сборщик мусора, работающий с виртуальными машинами Java 2 и Java 5, намного лучше стал справляться со своей задачей, но 10 Мбайт? С кодом явно было что-то не то. Когда я понял, где таился корень проблемы, то не знал, смеяться мне, плакать или ругаться. Ответственным за творившееся безобразие оказался разработчик, и я выкрикнул в его адрес «0x7f». (Навел на него порчу.)

Каждая страница с информацией о продукте и каждая «вставка продукта» — для нее применялись компактные поля на главной странице и на страницах категорий — использовали нестандартный компонент, который назывался Profanity Masker (маскировщик ненормативной лексики). Эта маленькая жемчужина представляла собой ATG-дропплет, напоминавший нестандартный JSP-тег. Любой дропплет обрабатывает переданный ему контент и генерирует некий результат. Вводимые данные до активизации дроплета буферизуются. Буферизуется и результат перед тем, как его включают в состав страницы. Маскировщик ненормативной лексики работал с названиями продуктов, короткими и подробными описаниями и характеристиками. Для музыкальных файлов он применялся, если указывалось имя трека. В случае фильма он работал с именами всех актеров. На одной странице с информацией о продукте могло оказаться двадцать экземпляров этого гадкого компонента.

Для анализа контента компонент использовал класс StringTokenizer, сравнивая все слова предоставленного ему фрагмента со списком из одиннадцати непристойных слов. Если слово в списке не обнаруживалось, оно добавлялось в объект StringBuffer, в котором происходило формирование результата. В случае же совпадения все буквы в слове, кроме первой, заменялись звездочками, чтобы читатели могли понять, что именно было в оригинале.

А теперь попробуем представить, как все это выглядело. Каждый бит текстового контента помечался метками, производилось его пошаговое сравнение со структурой Vector (да, Vector, а не ArrayList) из плохих слов, после чего все снова собиралось вместе. И все это на странице делалось двадцать раз при количестве ежедневных просмотров этой страницы, превышающем пять миллионов. Фактически каждый запрос страницы сопровождался сравнением несметного количества строк и генерацией 10 Мбайт мусора. И главное — весь контент обычно публикуется по ночам. Чему была равна вероятность внезапного появления нецензурных слов днем? И зачем эту работу нужно было выполнять при каждом просмотре страницы?

Самым забавным оказался эпилог данной истории. Когда мы начали спрашивать, нельзя ли остановить работу маскировщика ненормативной лексики (который представлял угрозу для здоровья нашего рассудка, хотя и защищал сайт), ответственный за контент представитель инвестора буквально взорвался. Он кричал, что описания продукции, имена альбомов, примеры текстов и названия песен являются авторскими материалами. Группируются они поставщиком данных и поставляются звукозаписывающими компаниями и киностудиями. В соответствии с контрактом мы не можем просто взять и изменить в этом контенте даже одну букву, пусть даже с благородными целями. Он был до ужаса расстроен, узнав, что до сих пор при визуализации на сайте контент менялся. Мы, конечно же, сразу убрали маскировщик ненормативной лексики, но так никогда и не выяснили, зачем он вообще был написан, раз менять контент сайт права не имел.

Разумеется, предварительный обсчет контента сам по себе требует определенных затрат. Прежде всего, нужно место для хранения его результатов. Какие-то издержки вносит процедура, которая проецирует идентификатор на файл и читает файл. В случае широко используемого контента эти издержки могли бы мотивировать вас на кэширование этих данных в памяти. Издержки на генерацию контента возникают главным образом при внесении в него изменений. Если контент будет многократно использоваться до момента внесения в него изменений, имеет смысл обработать его заранее.

Препятствием к предварительному обсчету контента является персонализация. Если страница персонифицирована целиком, заранее вычислить для нее контент невозможно. В то же время при персонализации только некоторых фрагментов или разделов ничто не мешает обсчитать остальную часть страницы заранее.

Обсчитывайте большую часть страницы заранее, а затем добавляйте к ней персонализированные фрагменты.

Предварительное вычисление контента отличается от кэширования. Кэширование в памяти результатов, извлеченных из базы данных, является компромиссом, увеличивающим нагрузку на один элемент (оперативную память сервера приложений), чтобы снять нагрузку с другого (процессора базы данных или с операций ввода-вывода в зависимости от того, что в вашей системе страдает больше). Кэширование в памяти фрагментов страниц относится к более слабым компромиссам, призванным сбалансировать время реакции сервера приложений и требования к памяти этого сервера. Однако первая проблема здесь состоит в том, что такое кэширование может в конечном счете негативно сказаться на времени отклика системы. Если возникнет нехватка памяти или размер кэша окажется меньше рабочего набора обслуживаемых фрагментов, серверу приложений придется тратить время на перегрузку кэша. Хуже то, что ему будет не хватать памяти для временных заданий по обслуживанию страниц. Серверы Java-приложений при нехватке памяти очень сильно замедляют свою работу из-за мегабайтов кэшированного контента в пространстве кучи старого поколения. Вторая проблема заключается во времени,

которое требуется, чтобы «разогреть» (то есть заполнить) кэши в памяти. Когда протокол объявления служб запрашивает первую страницу у сервера с «холодными» кэшами, ожидание может растянуться на несколько *минут*. Кэширование в памяти является полезным инструментом, но хранение в таком кэше визуализированных фрагментов страниц с большим количеством контента является примером его нецелевого использования.

При предварительном вычислении контента принцип «все или ничего» не применяется. Некоторые часто посещаемые области сайта допускают предварительное вычисление, в то время как менее востребованные страницы могут оставаться полностью динамическими. Владельцы магазинов, исключая Amazon, решают, что должно находиться на главной странице, и все посетители видят там один и тот же набор товаров. При этом сайт может приветствовать посетителя по имени и даже демонстрировать ему сохраненную с прошлого визита корзину, но это лишь около 100 байт относящейся к пользователю информации. Оставшиеся 100 Кбайт (я решил проявить щедрость) для всех посетителей одинаковы.

ЗАПОМНИТЕ

Редко меняющийся контент обсчитывайте заранее

Любой контент, который до момента своего редактирования будет демонстрироваться многократно, имеет смысл подготовить заранее, чтобы сэкономить время в процессе обработки запросов. Учтите издержки на генерацию контента из индивидуальных запросов и процесс развертывания.

10.4. Настройка сборщика мусора



В Java-приложениях настройка механизма сборки мусора является самым быстрым и простым способом добиться роста производительности. Без такой настройки приложение, работающее с промышленными объемами данных и трафиком, будет тратить примерно 10 % времени на сборку мусора. Этот показатель следует уменьшить до 2 % или даже еще меньше.

Современные сборщики мусора оптимизированы под типичное бимодальное распределение времени жизни объектов. Большинство объектов живет недолго; они создаются и уничтожаются в течение микросекунд. (В компании Sun это называют *детской смертностью*.) Количество долгоживущих объектов намного меньше, но их время жизни зачастую сопоставимо со временем работы программы (объекты-одиночки, службы реестров, сокет серверов и т. п.). Всем объектам выделяется память в области кучи, которая называется райской областью (eden space). Пережившие хотя бы одну сборку мусора объекты перемещаются в область выживших (survivor space). Райская область и область выживших относятся к так называемым областям «молодого поколения».

Работающий сборщик мусора первым делом проверяет живые объекты в райской области. Если таких объектов там не обнаруживается, вся область быстро пускается в переработку. Все живые объекты из этой области перемещаются в одну из областей выживших. Поэтому быстрая утилизация объектов для сборщика мусора проблемой не является. Выжившие объекты в конечном счете перемещаются сборщиком мусора в область долгоживущего поколения (*tenured generation*), которая посещается им намного реже, чем молодое поколение. Еще одно, третье, поколение является *несменяемым* (*permanent generation*) и содержит определения классов и методов.

Посмотреть подробные протоколы сборки мусора можно, передав виртуальной Java-машине в момент запуска аргумент `-verbosegc`. Отчеты сборщика мусора отправляются на консоль, поэтому вы должны удостовериться, что где-то есть стандартное устройство вывода. (В случае серверов приложений для этого обычно требуется редактировать сценарий запуска.) Пользователи Java версии 5 и выше могут воспользоваться утилитой `jconsole`, поставляющейся вместе с Java SDK. Пример информации, предоставляемой этим инструментом, показан на рис. 26. Вкладка **Memory** демонстрирует загрузку кучи для разных поколений и областей, а также время, затраченное на сборку мусора. Это достаточно незагруженный сервер календарей, не подвергающийся избыточной сборке мусора.

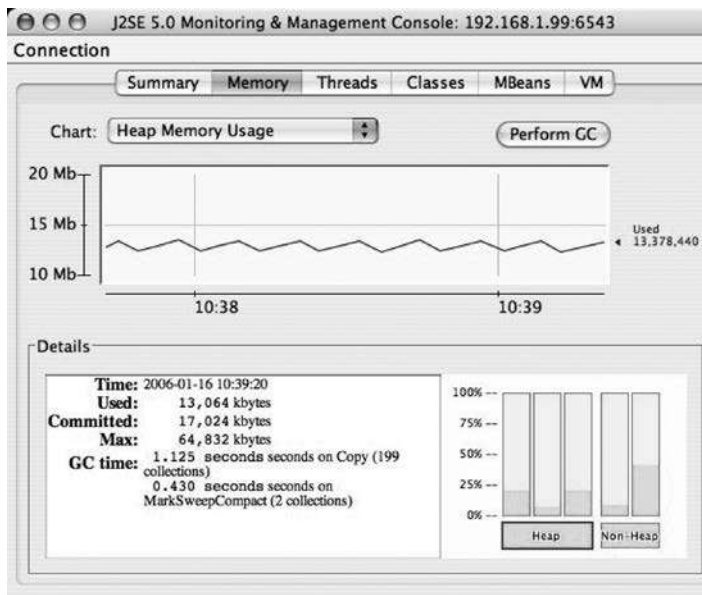


Рис. 26. Вкладка Memory утилиты JConsole

Как показывают различные модели сборки мусора, настройка сборщика — это в значительной степени вопрос обеспечения достаточного размера кучи и регулировка соотношений, контролирующих относительные размеры поколений. В Интернете существуют отличные руководства по настройке механизма сборки мусора

компании Sun¹. В качестве дополнительного бонуса при выполнении упражнений по настройке сборщика мусора зачастую удается обнаружить утечки памяти!

Настройка механизма сборки мусора одновременно является наукой и искусством. Поведение сборщика полностью зависит от поведения приложения и режимов нагрузок. Каждая следующая версия кода определенным образом меняет окружение. Даже относительно небольшие обновления — рекламные акции, изменения, направленные на большее удобство для пользователя, предоставление функциональности, требующей изрядных ресурсов, и т. п. — могут заставить пользователей вести себя совсем по-другому. Идеальные варианты настройки (если они вообще существуют в природе) для одной версии порой оказываются совершенно непригодными для следующей. Как вы увидите в главе 17, вам не обойтись без процедуры, которая будет постоянно заниматься настройкой и перенастройкой параметров.

ДЖО СПРАШИВАЕТ: А КАК НАСЧЕТ ПУЛА ОБЪЕКТОВ?

В ранних версиях Java (во время появления версии 1.2) идея долгоживущих объектов была весьма популярной. Я помню, как мне говорили, что «создание объекта является второй по дороговизне операцией, которую можно выполнить в Java» (в качестве первой называли создание нового программного потока). Предполагалось, что этой операции следует по возможности избегать. Вместо этого предлагалось многократно использовать уже существующие объекты. Корректность этого подхода является предметом жарких споров в Java-сообществе. В любом случае при современных виртуальных Java-машинах организация пула объектов не даст нам нужного результата. Некоторые системы в попытках избежать создания новых объектов становятся до смешного большими. Добавляется такой уровень сложности и учета системных ресурсов, что нивелируется любой возможный выигрыш производительности.

В следующей таблице показано время работы простого приложения, которое формирует 50 000 имен при помощи класса `NameFormatter`. В конфигурации с пулом пакет `commons` из проекта `Jakarta` заставляет пул объектов многократно использовать данный класс. В конфигурации без пула создаются 50 000 отдельных классов `NameFormatter`. Данные в таблице четко показывают, что издержки учета системных ресурсов пула превышают издержки создания объектов. (Все тесты выполнялись в JDK 1.4.2.)

Операционная система	Тактовая частота процессора	Издержки при наличии пула	Издержки без пула
Windows XP Pro	1,86 ГГц	20,30 %	20,30 %
Linux 2.6.14	2,66 ГГц	31,46 %	23,42 %
Mac OS 10.4.4	1,67 ГГц	24,69 %	15,69 %

Оставьте пулы объектов для случаев, когда создание объектов действительно требует множества ресурсов, например для сетевых соединений, соединений с базами данных и рабочих потоков.

¹ Для JDK 1.4.2 см. <http://java.sun.com/docs/hotspot/gc1.4.2/index.html>. Для Java 5 см. http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html.

ЗАПОМНИТЕ**Настраивайте сборщик мусора на этапе эксплуатации системы**

Оптимальные параметры сильно зависят от схемы пользовательского доступа, поэтому корректно отрегулировать сборщик мусора на этапе разработки или тестирования невозможно.

Не останавливайтесь

Настройка сборщика мусора требуется после выхода каждой следующей версии приложения. Если у вас есть цикл годового спроса, сборщик мусора нужно настраивать в зависимости от времени года и соответствующего смещения пользовательского трафика к другой функциональной возможности.

Обычным объектам пул не нужен

Единственными объектами, для которых имеет смысл организовать пул, являются внешние соединения и программные потоки. Заботу обо всем остальном отдайте на откуп сборщику мусора.

10.5. Заключение

Пулы ресурсов могут стать источниками конкуренции, приводящей к резкому снижению вычислительной мощности системы. Однако при правильной организации они значительно увеличивают данный показатель. Пул соединений экономит до 500 миллисекунд времени на каждой транзакции. Существуют разные стратегии управления соединениями, каждая со своими достоинствами и недостатками. К этим стратегиям относятся и упоминавшиеся в этой главе «постраничная», «по-фрагментная» и гибридная модели.

Кэширование представляет собой обоюдоострый меч. В случае корректной реализации оно может быть весьма полезным. Однако повсеместно встречаются случаи, когда кэш становится слишком большим, кэш никогда не очищается, и там оказываются объекты, хранение которых не сулит никакой выгоды. Принципиально важно ограничить размер кэша. Java-разработчикам для управления кэшированными объектами следует применять объекты `SoftReference` в комбинации со сборщиком мусора.

Перед тем как решиться на очередное усовершенствование, проанализируйте возможные эффекты умножения. К примеру, не имеет смысла миллион раз в день динамически визуализировать контент, который меняется всего раз в неделю. Выгоду от такого улучшения вы будете получать раз в неделю, в то время как платить за это придется миллион раз в день. А вот однократная визуализация такого контента сразу же после внесения в него изменений и хранение предварительно вычисленного результата приносят выгоду миллион раз в день, в то время как затраты на управление кэшем возникают только раз в неделю. Настройка сборщика мусора жизненно важна для Java-приложений. Это не однократная операция, а непрерывный процесс. Каждая новая версия кода способна до такой степени изменить паттерны сборки мусора, что потребуются перенастройка механизма сборки мусора.

Часть III

Общие вопросы проектирования

11 Организация сети

Организация сети в центрах хранения и обработки данных выходит далеко за рамки API сокетов прикладного уровня. В проектных решениях сетей для таких центров избыточность, безопасность и гибкость имеют куда большее значение, чем в сетях, связывающих домашние компьютеры. Для корректной работы в этой среде любое приложение требует дополнительной настройки.

11.1. Многоинтерфейсные серверы

Множественная адресация является основным отличием машины в среде разработки или тестирования от машины в центре хранения и обработки данных. Практически любой сервер в таком центре будет *многоинтерфейсным* (multihomed). Этот термин означает сервер с более чем одним IP-адресом; он существует в нескольких сетях одновременно. Подобная архитектура является более безопасной за счет распределения администрирования и текущего контроля по собственным, хорошо защищенным сетям. Производительность повышается благодаря отделению объемного трафика, возникающего, например, при резервном копировании, от обычного рабочего трафика. Я уверен, что вам приходилось сталкиваться с замедлением работы системы из-за резервного копирования. Эта процедура загружает любую сеть, так как всегда запускается на полную мощность. Если вы увеличите быстродействие сети, резервное копирование просто закончится раньше (разумеется, это верно до какого-то предела). Сети, которые мы будем рассматривать, имеют совсем другие требования к безопасности. Приложение, которое не знает о наличии нескольких сетевых интерфейсов, легко может принять подключение не от той сети. Например,

оно может принимать административные соединения из производственной сети или предложить производственную функциональность сети, предназначенной для резервного копирования.

Показанный на рис. 27 сервер имеет четыре сетевых интерфейса. В операционной системе Linux это были бы интерфейсы от `eth0` до `eth3`. В Solaris это были бы интерфейсы от `ce0` до `ce3` или от `qfe0` до `qfe3`, в зависимости от сетевой карты и версии драйвера. Операционная система Windows по умолчанию присваивает интерфейсам удивительно длинные и громоздкие имена.

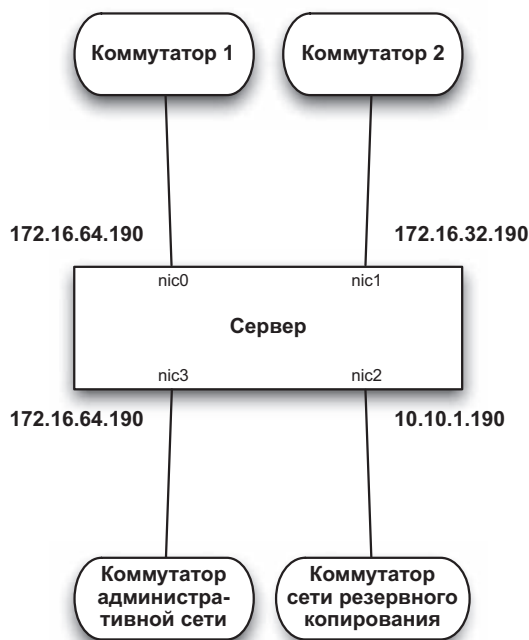


Рис. 27. Несколько сетевых интерфейсов

Два из четырех интерфейсов выделены под «производственный» трафик. Именно они поддерживают функциональность приложения. В случае веб-сервера именно они занимались бы обработкой входящих запросов и отправкой ответов. В приведенном примере оба интерфейса предназначены для производственного трафика. То, что они запущены на разных коммутаторах, означает, что конфигурация сервера предполагает высокую доступность. Эти интерфейсы могут отвечать за балансировку нагрузки или за реализацию аварийного переключения. Как показано на рисунке, пакеты для сервера будут получать два разных IP-адреса. Это, скорее всего, означает наличие DNS-записей для обоих адресов. Другими словами, у этой машины больше чем одно имя! У нее есть собственное внутреннее имя хоста — именно

оно возвращается в ответ на команду `hostname`, — но снаружи к этому хосту можно обращаться по нескольким именам.

Другой распространенной конфигурацией для множественных рабочих интерфейсов является *связывание* (bonding), или *агрегирование* (teaming). В этом случае оба интерфейса имеют один общий IP-адрес. Операционная система гарантирует, что отдельный пакет попадет только к одному из интерфейсов. Связанные интерфейсы можно настроить так, чтобы исходящий трафик автоматически балансировался или какое-либо из соединений считалось предпочтительным. Связанные интерфейсы, подключенные к разным коммутаторам, требуют дополнительной настройки, иначе может возникнуть заикливание маршрутов. Спровоцировав заикливание в центре обработки и хранения данных, вы станете очень известным человеком, но это будет дурная слава.

Два дополнительных «внутренних» интерфейса предназначены для трафика особого назначения. Так как при резервном копировании пакетами пересылаются большие объемы данных, они могут засорить эксплуатируемую сеть. Соответственно в хорошем проекте сети для центра хранения данных трафик, связанный с резервированием, проходит по собственному сетевому сегменту. Иногда такие сегменты обслуживаются отдельными коммутаторами, а иногда им выделяются отдельные сети VLAN на основных рабочих коммутаторах. Отделение связанного с резервным копированием трафика от производственной сети призвано помочь пользователям. Проблемы могут возникнуть, если у сервера не хватает полосы пропускания ввода-вывода для одновременного обслуживания резервного копирования и трафика приложений. Тем не менее на работу пользователей *остальных* приложений резервное копирование сервера не влияет.

Наконец, во многих центрах хранения и обработки данных для административного доступа существует отдельная сеть. Это важный шаг для обеспечения безопасности, потому что такие службы, как SSH, допускают привязку только к административным интерфейсам и, соответственно, недоступны из производственной сети. Это помогает, когда злоумышленникам удастся пройти за фаервол или когда сервер работает с внутренним приложением, не прячась за фаерволом.

Множественность интерфейсов влияет на ПО приложений. По умолчанию приложение, слушающее сокет, будет слышать и попытки подключения к любому интерфейсу. Например, в языке Java по состоянию на версию 5 класс `ServerSocket` имеет четыре конструктора. Три из них связываются с *любым интерфейсом* на сервере. И только длинная форма конструктора может принимать определенный локальный адрес, указывающий, с каким из интерфейсов нужно связываться.

```
InetAddress addr = InetAddress.getByName("alpha.example.com");
ServerSocket socket = new ServerSocket(80, 50, addr);
```

```
Socket local = socket.accept();
...
```

Без этого адреса класс `ServerSocket` будет связываться со всеми интерфейсами, допускающими подключение через сети для резервного копирования или для администрирования к рабочему серверу. Или, наоборот, станут возможными подключения к административному интерфейсу через рабочую сеть!

Чтобы указать, с каким интерфейсом следует связываться, приложению нужно сообщить его имя или IP-адреса. Именно в этом состоит серьезное отличие много-интерфейсных серверов. В процессе разработки сервер всегда может вызвать метод `InetAddress.getLocalHost()`, но на многоинтерфейсной машине этот метод просто вернет IP-адрес, связанный с внутренним именем сервера. Это может оказаться любой из интерфейсов в зависимости от локальных соглашений об именовании. Соответственно, приложения сервера, которым нужно слушать сокет, должны обладать настраиваемыми свойствами, позволяющими указать, с каким из интерфейсов следует связываться серверу.

11.2. Маршрутизация

Так как эксплуатируемые серверы обычно имеют несколько сетевых интерфейсов, периодически возникает вопрос, через какой интерфейс должен идти конкретный трафик. К примеру, относительно часто встречается ситуация, когда входной интерфейс сервера приложений подсоединен к одной виртуальной локальной сети (VLAN) для взаимодействия с веб-серверами, а внутренний сетевой интерфейс подсоединен к другой VLAN для взаимодействия с серверами базы данных. В подобных случаях серверу следует в явном виде указывать, каким интерфейсом он должен пользоваться для достижения конкретного IP-адреса.

В случае соседних серверов проблема маршрутов решается, скорее всего, легко; решение базируется на адресах подсети. В приведенном ранее примере с сервером приложений внутренний интерфейс, возможно, пользуется одной подсетью с сервером базы данных, в то время как входной интерфейс может пользоваться одной подсетью с веб-серверами. В случае удаленных серверов — например, серверов сторонних служб — выбор маршрута становится более сложной задачей.

Рассмотрим стороннюю службу *поставщика спама* (spam cannon). Данные, отправленные в эту службу серверами приложений, скорее всего, не пойдут через открытый Интернет. Вместо этого их отправят через внутренний интерфейс серверов приложений, через VPN.

Поставщик спама — это поставщик, осуществляющий массовую рассылку электронной почты, как правило, на основе XML-списка с именами и адресами. Он заполняет красивые, предустановленные HTML-шаблоны собранными подписчиком личными данными. У таких поставщиков есть широкополосное соединение, а порой и алгоритмы, позволяющие избежать падения SMTP-серверов у небольших провайдеров. Результаты их тяжелого труда, как правило, сразу же отправляются в мусорную корзину.

Корректное решение всех связанных с маршрутизацией вопросов требует внимания ко всем точкам интеграции. В случае ошибки вы рискуете сделать сайт менее доступным или, что еще хуже, сделать данные клиентов общим достоянием. Я рекомендую для каждого подключения к удаленному серверу сохранять в базе данных Microsoft Access имя адресата, его адрес и желаемый маршрут. Рано или поздно эта информация в любом случае понадобится, чтобы написать правила для фаервола.

11.3. Виртуальные IP-адреса

Можете мне не верить, но далеко не все приложения пишутся для их запуска в кластере. По каким-то причинам такие приложения в каждый момент времени могут быть активными только на одном сервере. Как же обеспечить доступность такого приложения, лишняя раз не запуская его на разных серверах?

Ответом на этот вопрос являются серверы кластеров. Сервер кластера — это приложение, играющее роль контроллера для других приложений. Такие продукты, как Cluster Server от Veritas, ServiceGuard от HP или Cluster Server от Microsoft¹, запускаются на нескольких серверах и гарантируют, что определенный «пакет» будет запущен в кластере всего один раз. Настройка пакетов для приложений, файловых систем и IP-адресов позволяет сделать так, чтобы сервер кластера выполнял корректное отключение приложения на одном сервере и его запуск на другом. По большей части приложение не будет знать, что входит в кластер.

Представим, что прекращает работу сервер, на котором располагается важное, но по своей природе не допускающее кластеризации приложение. Сервер кластера на узле обработки отказа заметит отсутствие обычной жизнедеятельности упавшего сервера. И решит, что исходный сервер прекратил работу.

После этого он запустит приложение на втором сервере, смонтировав там все необходимые файловые системы. Заодно он перенаправит виртуальный IP-адрес, назначенный кластеризованному сетевому интерфейсу.

Виртуальный IP-адрес представляет собой IP-адрес, который при необходимости можно перебросить от одной сетевой карты к другой. В любой момент времени этот адрес требуется ровно одному серверу. Когда возникает необходимость в переброске адреса, сервер кластера объединяется с операционной системой, чтобы проделать кое-какие интересные операции на нижних уровнях TCP/IP-стека. Они ассоциируют этот IP-адрес с новым MAC-адресом (аппаратным адресом) и объявляют

¹ Сейчас переименован в Microsoft Windows Server 2003 Clustering Services. Говорят, что новая версия работает в три раза быстрее.

новый маршрут (через протокол ARP). Рисунок 28 демонстрирует виртуальный IP-адрес до и после отказа активного узла.

ДВЕ РАЗНОВИДНОСТИ ВИРТУАЛЬНОГО IP-АДРЕСА

К сожалению, термин **виртуальный IP-адрес** стал слишком многозначным. Вообще говоря, он означает IP-адрес, не привязанный к определенному MAC-адресу в Ethernet. Серверы кластеров присваивают данный адрес различным членам кластера. Распределители нагрузки пользуются виртуальными IP-адресами для объединения набора служб (каждая со своим собственным IP-адресом) в меньшее число физических интерфейсов. В итоге значения термина несколько перекрываются, так как распределители нагрузки, как правило, работают парами, и виртуальный IP-адрес в смысле «адрес службы» может одновременно являться виртуальным IP-адресом в смысле «меняющийся адрес».

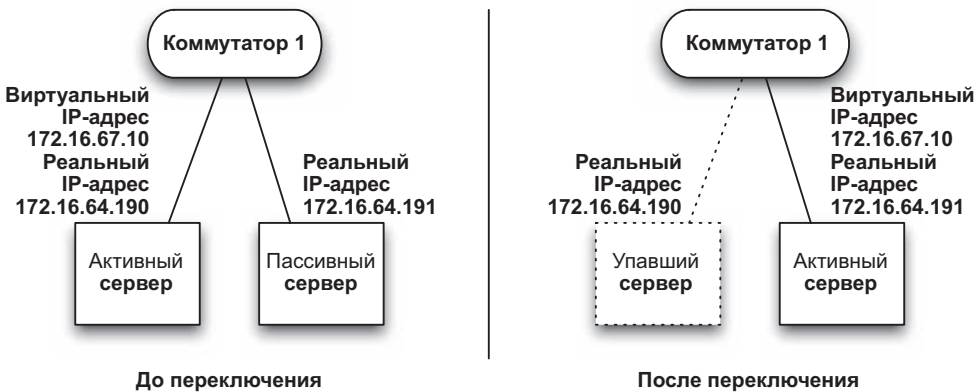


Рис. 28. Смена виртуального IP-адреса

Виртуальные IP-адреса часто используются для активных/пассивных кластеров баз данных. Клиенты получают инструкцию подсоединяться только к DNS-имени виртуального IP-адреса, а не к именам узлов кластера. В этом случае не имеет значения, какой из узлов в настоящий момент обладает данным IP-адресом, клиент может подключаться к одному и тому же имени.

Разумеется, этот подход не позволяет перемещать данные о состоянии приложения, находящиеся в памяти. В результате вся непостоянная информация о состоянии взаимодействий оказывается потерянной. В случае баз данных это относится к незавершенным транзакциям. Некоторые драйверы баз данных, например JDBC- и ODBC-драйверы в Oracle, автоматически повторно выполняют прерванные при аварийном переключении запросы. Однако обновления, вставки и вызовы хранимых процедур не допускают автоматического повторения. Поэтому любое обращающееся

к базе данных через виртуальный IP-адрес приложение должно быть готово к появлению в случае аварийного переключения исключения `SQLException`.

В общем случае, если приложение обращается через виртуальный IP-адрес к любой службе, следует быть готовым к тому, что следующий TCP-пакет будет направлен не к тому интерфейсу, к которому ушел предыдущий. Из-за этого возможно появление исключения `IOException` в странных местах. Логика приложения должна уметь обрабатывать такие ошибки — и обрабатывать их не так, как обычные ошибки, связанные с «недоступностью объекта назначения». По возможности приложение должно повторять свой запрос, но адресуясь уже к другому узлу (внимательно изучите раздел 5.2, в котором рассказывается о важных ограничениях безопасности, связанных с повторением операций).

12

Безопасность

Исчерпывающий рассказ о безопасности приложений выходит за рамки темы данной книги. В этой главе мы лишь коснемся вопросов, находящихся на стыке таких тем, как программная архитектура, эксплуатация и безопасность.

12.1. Принцип минимальных привилегий

Принцип «минимальных привилегий» гласит, что привилегии любого процесса должны находиться на минимальном уровне, при котором он в состоянии справиться со своими задачами. Скажем, для прикладных программ сюда никогда не входит возможность запуска от имени пользователя root (в UNIX/Linux) или пользователя Administrator (в Windows). Вся функциональность таких приложений доступна пользователям, не обладающим правами администратора.

Мне приходилось видеть серверы Windows, у которых вход в систему с правами администратора длился неделями — с доступом к удаленному рабочему столу, — потому что этого требовал какой-то фрагмент программного обеспечения. (Этот пакет ко всему прочему был не в состоянии работать как служба NT, поэтому требовалось долговременное функционирование интерактивных приложений Windows. Это явно *не* тот случай, когда центр обработки и хранения данных можно назвать готовым к работе!)

Программное обеспечение, требующее для выполнения прав привилегированного пользователя, автоматически становится мишенью для взломщиков. Любая уязвимость такого ПО сразу же превращается в серьезную проблему. Как только

злоумышленник получает права привилегированного пользователя, единственным способом гарантировать безопасность сервера является его форматирование и переустановка. Хуже всего то, что в случае горизонтально масштабируемых приложений может потребоваться переустановка целого кластера.

В целях ограничения уязвимости каждое важное приложение должно иметь собственного пользователя. Например, пользователь «apache» не должен иметь никакого доступа к пользователю «websphere».

Открытие сокета на порту с номером меньше 1024 должно быть единственной операцией, для которой UNIX-приложению могут потребоваться полномочия пользователя root. Веб-серверы часто хотят, чтобы порт 80 был открыт по умолчанию. Любой веб-сервер, находящийся за распределителем нагрузки (см. раздел 13.3), может использовать любой порт, в том числе порт с номерами 1024 и больше. Слушать порт 80 должен только распределитель нагрузки.

Веб-сервер, у которого отсутствует распределитель нагрузки (полагаю, для этого есть веские причины, хотя сам я подобную ситуацию придумать не в состоянии), должен запускаться с правами администратора, чтобы слушать порт 80. Но если посмотреть, как устроена система Apache, вы увидите, что в ней имеет место «разделение полномочий» с намеренным отказом от привилегированного доступа после открытия сокета. При помощи низкоуровневых C-функций собственный статус понижается до конфигурируемого пользователя (обычно это пользователь «apache»). Данное действие необратимо; процесс не может восстановить привилегии суперпользователя после того, как он от них отказался.

12.2. Настроенные пароли

Пароли являются ахиллесовой пятой системы безопасности приложений. Очевидно, что возможности интерактивно вводить пароли при каждой загрузке сервера приложений не существует. Соответственно, пароли баз данных и учетные данные, необходимые для аутентификации в других системах, должны храниться в каких-то файлах.

Но как только пароль оказывается в текстовом файле, он становится уязвимым. Любой пароль, дающий доступ к базе данных с информацией о клиентах, является лакомым кусочком для злоумышленника и может стоить компании многих тысяч долларов из-за возможности освещения в средствах массовой информации или вымогательства. Поэтому такие пароли должны быть защищены на максимально возможном уровне.

Минимальной мерой защиты является хранение файла с паролями доступа к рабочей базе данных отдельно от остальных конфигурационных файлов. Ни в коем случае нельзя помещать их в каталог установки программного обеспечения. Мне доводилось видеть, как папку установки целиком помещали в архив и отправляли

разработчику для анализа. Чаще всего подобное происходит, когда возникают вопросы по поводу рабочей конфигурации, а также корректности установки и настройки приложения. Следует сделать так, чтобы доступ на чтение файлов, содержащих список паролей, был только у владельца, который является пользователем приложения. Если приложение написано на языке, допускающем разделение полномочий, имеет смысл сделать так, чтобы приложение читало файлы с паролями до понижения своих привилегий. В данном случае владельцем файлов с паролями может быть сделан пользователь `root`. Но имейте в виду, если приложение сохраняет эти файлы в памяти, дампы памяти будут содержать все пароли. Файлы ядра в системах семейства UNIX представляют собой всего лишь дампы памяти приложений. Если злоумышленник сможет спровоцировать создание дампа ядра, то при наличии доступа к файловой системе сервера он получит все пароли. Поэтому у рабочих приложений лучше отключить функцию формирования дампа ядра. В операционных системах семейства Windows «синий экран смерти» означает ошибку ядра, которая всегда сопровождается дампом памяти. Файл этого дампа можно проанализировать предлагаемыми Microsoft инструментами отладки ядра. В некоторых конфигурациях сервера дамп может содержать полную копию физической памяти компьютера, включая пароли и все остальное¹.

Перенос паролей в удаленное резервное хранилище (password vaulting) позволяет хранить их в зашифрованных файлах, что снижает остроту проблемы безопасности, ведь обезопасить один ключ шифрования куда проще, чем несколько текстовых файлов. Это может помочь в защите паролей, но не означает полного решения проблемы. Так как случайно поменять или переписать права доступа к файлам очень легко, следует задействовать программное обеспечение для обнаружения вторжений, например Tripwire² или одноименное ПО с открытым исходным кодом³, которое будет следить за правами на доступ к этим жизненно важным файлам.

¹ Более того, в редакторе реестра можно включить комбинацию клавиш (нет, тут я ее упоминать не буду), которая вызывает *принудительное* формирование дампа памяти и остановку компьютера. Если вы когда-нибудь увидите «синий экран смерти» с текстом **the end-user manually generated the crashdump** (конечный пользователь вручную сгенерировал аварийный дамп), значит, ваш компьютер под угрозой. Лучше всего сразу же убедиться, что данная «функциональная возможность» не включена на каком-то из ваших рабочих серверов.

² См. <http://www.tripwire.com/>.

³ См. <http://www.sourceforge.net/projects/tripwire>.

13 Доступность

Если «хочу» рассматривается в отрыве от того, сколько это будет стоить, возникают нереалистичные желания. Спросите у детей, сколько мороженого они хотят, и в ответ гарантированно услышите: «Давайте все, что есть». Они не имеют понятия, чем придется платить за съеденную бочку мороженого: к сумме, которую нужно будет отдать при покупке, следует прибавить испорченное здоровье, набранный вес и тошнотворную тяжесть в животе.

В этой главе мы обсудим непростые вопросы поиска баланса между раздирающими нас противоречивыми желаниями: одновременно обеспечить большую доступность и при этом обойтись минимальными затратами. Эти желания диаметрально противоположны; гарантированная доступность обязательно ведет к увеличению затрат.

13.1. Сбор требований доступности

Задав спонсорам системы вопрос: «Насколько высокая доступность вам требуется?», вы, скорее всего, получите один из двух вариантов ответа. Менее опытные спонсоры просто скажут, что им нужно «100 %». Более осведомленные блеснут словосочетанием «пять девяток», так как это звучит круто и технически грамотно. Но разве им *действительно* требуется столь высокая доступность?

Корректно сформулировать решение данного вопроса позволяют финансовые термины: сравнение фактических затрат с предотвращенными убытками. К примеру, «доступность 98 %» ежемесячно превращается в 864 минуты простоя. И такой простой означает прямые издержки по причине упущенной выгоды. Предположим, что в часы наивысшего спроса сайт приносит по 1500 долларов в час, — это максимально

невыгодное время для простоя. В этом случае при 98-процентной доступности потери составят примерно 21 600 долларов. А теперь посмотрим, на сколько минут можно сократить время простоя, построив более надежную систему. Увеличение доступности до 99,99 % снижает ожидаемые потери от простоя до 108 долларов в месяц — вы выигрываете в месяц 21 492 долларов по сравнению с системой, доступность которой составляет 98 %.

Так стоит ли строить системы в расчете на доступность 99,99 %? Это напоминает вопрос о мороженом. Без учета затрат желания, конечно же, будут неограниченными. Каждая девятка в показателе доступности повышает стоимость внедрения примерно в десять раз, а годовую стоимость эксплуатации — примерно в два раза. В этом примере дополнительная стоимость всего жизненного цикла — затраты на внедрение плюс затраты на эксплуатацию за пятилетний срок службы системы — достигает 98 700 долларов. Потратить 98 700 долларов, чтобы сэкономить 1 289 520 долларов, — вполне обоснованный с финансовой точки зрения выбор.

Доступность, %	98	99,99
Время простоя, мин/мес.	864	4
Время простоя, долларов/мес.	21 600	108
Дополнительные расходы, долларов	0	98 700
Чистая экономия, долларов	0	1 289 520

13.2. Документирование требований доступности

Хотите гарантировать неприятные конфликты? Возьмите многозначное, нечеткое определение, заставьте людей подписать соглашение по поводу этого определения, добавьте изрядную сумму денег и через год-другой посмотрите, как они будут драться друг с другом.

Я описываю неизбежный эффект плохо составленного соглашения об уровне обслуживания, касающегося доступности. Определения в SLA — все равно что подробности в вашей медицинской страховке. Никто в них не вчитывается, пока не случится что-то ужасное. Если через год после запуска системы вы продолжаете дискутировать по поводу определений в SLA, скорее всего, эта острая дискуссия возникла в результате какого-то инцидента. В этой ситуации уже ничего не сделаешь.

Предотвратить (или, по крайней мере, сделать не таким болезненным) поиск виновного позволяет изначально проработанное вместе со спонсорами системы точное

соглашение об уровне обслуживания. Недостаточно написать на бумаге «Система должна быть доступна 99,9 % времени». Неопределенность скрывается в каждом слове этого предложения.

Что такое «система»? Кроме самых тривиальных случаев в понятие «система», как правило, входят вызовы других систем, находящихся как внутри, так и вне предприятия. Вы действительно собираетесь взять на себя ответственность за все эти вызовы? Я бы не стал! Кроме того, если система использует такие паттерны стабильности, как предохранитель (Circuit Breaker), то в любой момент может оказаться, что как целое система вполне работоспособна — она отвечает на запросы или генерирует веб-страницы, — но определенные программные компоненты не функционируют. Поэтому вместо постулата о доступности «системы» лучше сформулировать SLA относительно отдельных программных компонентов системы. К примеру, сеть отелей может быть заинтересована в предоставлении своим сайтом нескольких ключевых функциональных возможностей: поиск конкретных гостиниц, бронирование через Интернет, подписка на программу лояльности, регистрация на различные мероприятия и т. п. Каждая из этих функций по-своему важна для бизнеса. Регистрация на мероприятия и бронирование через Интернет влияют на получаемый доход, поэтому, скорее всего, с этими функциями будут связаны самые строгие соглашения об уровне обслуживания. Не стоит забывать и о таком антипаттерне, как SLA-инверсия (см. раздел 4.10): предлагаемый вами уровень обслуживания не может быть выше, чем у самой плохой внешней зависимости, обеспечивающей функциональность программного компонента. К примеру, программы лояльности часто обслуживаются сторонними фирмами, поэтому вы можете гарантировать уровень обслуживания, который в лучшем случае равен уровню обслуживания поставщика услуги.

Определить требуемый уровень доступности можно только после выделения каждого программного компонента или бизнес-процесса. Возможно, я начинаю напоминать юриста, но само понятие «доступность» также требует определения. Каким образом проверяется эта функциональная возможность? Это вооруженный мышкой человек, который сидит перед компьютером? (Надеюсь, это — не ваш случай.) Или это целая армия пользователей и их отчеты о проблемах, передаваемые через службу поддержки. (Опять-таки надеюсь, что это не так!) В идеале доступность должна проверяться автоматизированной системой, которая выполняет искусственные транзакции. В SLA следует указать, какое устройство или устройства будут следить за доступностью конкретной функциональной возможности. Более того, должно быть указано, каким образом эти следящие устройства отчитываются о выявленных проблемах.

Искусственная транзакция (synthetic transaction) — это предложенная системе работа, эмулирующая реального пользователя. Вы должны каким-то способом — например, выделив идентификатор пользователя, — пометить, что эта работа делается с целью мониторинга, чтобы избежать засорения производственных данных.

Если программный компонент работает, но реагирует через 27,5 минут, то с точки зрения большинства пользователей он недоступен. Значит, понятие «доступность» включает в себя такую вещь, как время. А как оценить транзакцию, отвечающую за 50 миллисекунд сообщением об ошибке? В данном случае речь о доступности тоже не идет, значит, в определение следует включить описание приемлемого ответа. А как насчет программного компонента, которого болтает из стороны в сторону, но который работает как автомобиль с механической коробкой передач и кажется вполне функциональным в момент проверки следящим устройством? Хорошее определение должно четко оговаривать следующие аспекты:

- ☐ Насколько часто следящее устройство выполняет искусственную транзакцию?
- ☐ Каково максимальное приемлемое время ответа на каждый шаг этой транзакции?
- ☐ Какие коды ответа или текстовые шаблоны свидетельствуют об успехе транзакции?
- ☐ Какие коды ответа или текстовые шаблоны указывают на отказ?
- ☐ Насколько регулярно должна выполняться искусственная транзакция?
- ☐ С какого количества адресов она должна выполняться?
- ☐ Куда будут записываться данные?
- ☐ По какой формуле будет рассчитываться процент доступности? На чем он будет основываться — на времени или на числе выполнений?

Когда дело доходит до разборок, документация вряд ли послужит вам надежной защитой, но наличие четких определений в состоянии поспособствовать тому, что в фокусе внимания окажутся не личные претензии, а данные о системе.

13.3. Балансировка нагрузки

В горизонтально масштабируемых системах доступность и масштабируемость достигаются за счет избыточности. Добавление компьютеров с целью увеличения вычислительной мощности одновременно увеличивает устойчивость к импульсам. Применение в горизонтально масштабируемой архитектуре серверов меньшего размера позволяет снижать расходы и увеличивать мощность небольшими порциями. Что в этом плохого?

Импульс — короткая, резкая встряска системы. Как удар молотком.

Построение горизонтально масштабируемых систем автоматически подразумевает некую форму распределения нагрузки. Фактически под этим термином

подразумевается распределение запросов по пулу или серверной ферме таким образом, чтобы все запросы были корректно обработаны за минимально возможное время. В последующих главах я то и дело буду ссылаться на проекты и ситуации, включающие одну из перечисленных далее форм распределения нагрузки.

DNS с карусельной диспетчеризацией

Технология *DNS с карусельной диспетчеризацией* (DNS round-robin) является самой старой технологией распределения нагрузки из всех, которые обсуждаются в этой главе, — фактически она появилась вместе с Интернетом. Она работает на прикладном уровне (уровень 7) стека OSI, но не во время запроса, а в процессе разрешения адресов.

В рамках этой технологии с именем службы просто связывается несколько IP-адресов. Чаще всего она используется на сайтах, относящихся к малому или среднему бизнесу, и вместо обнаружения одного IP-адреса для сайта, например `bobscleaners.example.com`, клиент получает один из нескольких адресов. При этом каждый IP-адрес указывает на один и тот же сервер. Соответственно, клиент подключается к одному из пулов серверов, как показано на рис. 29.

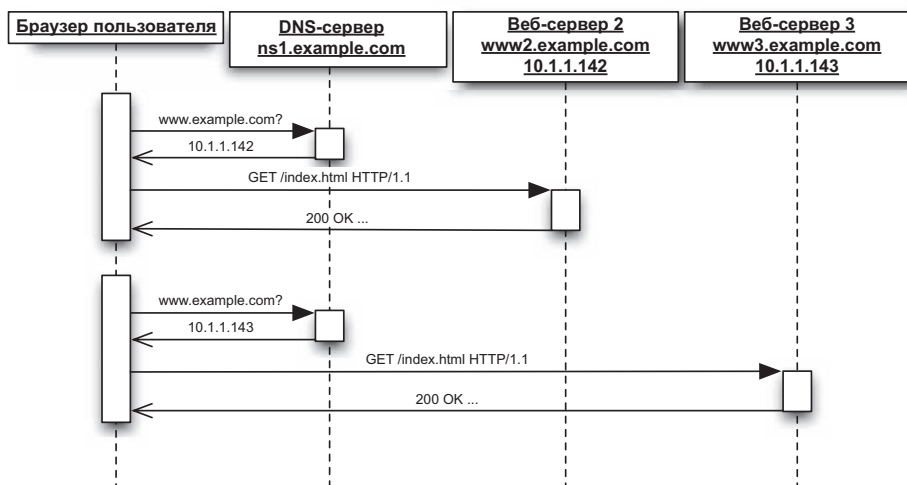


Рис. 29. Распределение нагрузки по принципу DNS с карусельной диспетчеризацией

Такой подход хорошо справляется с основной задачей — распределением работы среди машин группы, — но в остальном он слабоват. Прежде всего, все серверы в пуле должны быть «маршрутизируемыми». То есть несмотря на то, что они находятся за фаерволом, их внешние IP-адреса видимы и доступны клиентам. В наши дни это прямое приглашение к атаке.

Помимо вопросов, касающихся безопасности, технология DNS с карусельной диспетчеризацией дает клиентам слишком большую степень контроля. Так как каждый клиент подключается непосредственно к одному из серверов, в случае прекращения этим сервером работы возможность перенаправить куда-либо трафик отсутствует. У DNS-сервера нет данных о состоянии остальных серверов, поэтому ничто не мешает ему предлагать IP-адреса вышедших из строя веб-серверов. Более того, раздача IP-адресов в стиле карусельной диспетчеризации не гарантирует равномерного распределения *нагрузки*, равномерно распределяются только начальные подключения. Некоторые клиенты потребляют больше ресурсов, чем остальные, что ведет к дисбалансу. Опять же DNS-сервер не имеет возможности узнать, что какой-то из серверов уже занят, поэтому он продолжает посылать каждое одиннадцатое (или любое другое) соединение на перегруженный сервер.

Техника распределения нагрузки по принципу DNS с карусельной диспетчеризацией неприемлема, когда вызывающей стороной является другая давно работающая промышленная система. Все, что написано на языке Java, будет кэшировать первый полученный от DNS-сервера IP-адрес, гарантируя, что все следующие соединения будут направляться на тот же самый узел, полностью аннулируя попытки распределения нагрузки.

Некоторые конфигурации Apache-сервера используют карусельную диспетчеризацию при перезаписи URL-адресов. Пользователь может это наблюдать, когда адрес `www.example.com` внезапно превращается в `www7.example.com`. К сожалению, этот подход работает еще хуже, чем DNS с карусельной диспетчеризацией, так как пользователи ставят закладки на конкретные серверы, а не на адреса их «входных дверей».

Обратный прокси-сервер

Обратный прокси-сервер (*reverse proxy*) позволяет избавиться от ограничений системы DNS с карусельной диспетчеризацией, действуя как перехватчик всех запросов. При этом DNS разрешает имя службы ровно в один IP-адрес. Слушающее этот IP-адрес устройство представляет собой типичный хост, на котором работает прокси-сервер, настроенный в режиме «обратного прокси-сервера», как показано на рис. 30.

Обычный прокси-сервер объединяет множество исходящих запросов в один IP-адрес источника. Обратный прокси-сервер производит обратное действие: распределяет вызовы к одному IP-адресу по разным адресам. В случае веб-серверов замечательный обратный прокси-сервер получается из прокси-сервера с открытым исходным кодом Squid¹. Apache-модуль `mod_proxy` также позволяет серверу специального назначения действовать как обратный прокси-сервер для ферм или кластеров других Apache-серверов.

¹ Инструкции по настройке вы найдете на сайте <http://www.squid-cache.org/>.

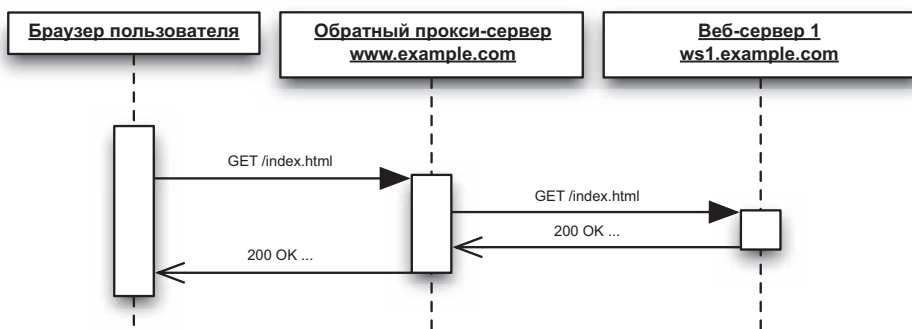


Рис. 30. Обратный прокси-сервер

Как и DNS с карусельной диспетчеризацией, обратные прокси-серверы работают на прикладном уровне. Поэтому прозрачными они не являются, но их настройка производится достаточно легко. Серверы приложений и веб-серверы следует настроить на генерацию URL-адресов не для их собственных имен, а для имен хостов. Регистрация исходного адреса запроса (как это делает Apache в случае формата журнала «Common») не имеет смысла, так как видимым является только прокси-сервер¹.

Еще обратные прокси-серверы можно настроить таким образом, что они будут снижать нагрузку на веб-серверы путем кэширования статического контента. Это выгодно, поскольку сокращает трафик во внутренней сети. (Статический контент, отправленный обратным прокси-сервером, по дороге из сети пересекает только один или два интерфейса. Когда источником является веб-сервер, он проходит через два или три интерфейса.) Если веб-серверы являются узким местом для вычислительной мощности системы (см. раздел 8.2), избавление от этого трафика увеличивает общую мощность. Разумеется, это никак не повлияет на ситуацию, когда в роли узкого места выступает сам прокси-сервер.

Самым большим в мире «кластером» обратных прокси-серверов является Akamai. Основная служба Akamai функционирует в точности как система Squid, настроенная как кэширующий прокси-сервер. Akamai кое в чем превосходит Squid, включая большое количество серверов, расположенных рядом с конечными пользователями, но по большому счету логически они эквивалентны².

¹ Тем не менее можно использовать нестандартный формат журнала для фиксации заголовка X-Forwarded-For, который Akamai и другие корректно работающие прокси-серверы добавляют к запросу. Следует заметить, что некорректно работающие или вредоносные прокси-серверы, скорее всего, не будут соответствовать этой части стандарта. В результате в случаях, когда необходимость отследить источник трафика максимальна, например при нападении, надежность заголовка будет минимальной. В этом случае остается полагаться на соответствие ваших журналов и журналов прокси-сервера.

² В данном случае я имею в виду только базовую службу Akamai — так называемый ускоритель веб-приложений (Web Application Accelerator). В плане других служб Akamai значительно превосходит Squid!

Так как обратный прокси-сервер работает со всеми запросами, он очень быстро может оказаться перегруженным. А если вы решите строить *перед* обратными прокси-серверами слой распределителей нагрузки, значит, настало время изучить другие варианты. Кроме того, так как прокси-сервер находится в центре любого запроса, он должен быть в состоянии отследить, какие из серверов-источников являются работоспособными и отзывчивыми. К сожалению, чаще всего используемые прокси-серверы — Squid и Apache — такой функциональностью не обладают. Они благополучно передают входящий запрос мертвому серверу, ждут истечения времени ожидания и возвращают вызывающей стороне сообщение об ошибке.

Аппаратный распределитель нагрузки

Аппаратные распределители нагрузки представляют собой специализированные сетевые устройства, которые играют ту же роль, что и обратные прокси-серверы. Такие устройства, как 11500 Series Content Services Switch от Cisco или BigIP от F5, реализуют механизмы перехвата и перенаправления. Располагаясь ближе к сети, аппаратные распределители нагрузки зачастую предоставляют больше возможностей для администрирования и резервирования. К примеру, CSS умеет периодически проверять работоспособность серверов в своем пуле, как показано на рис. 31. Этот распределитель нагрузки удаляет из своего пула вышедшие из строя серверы и направляет соединения работоспособным серверам.

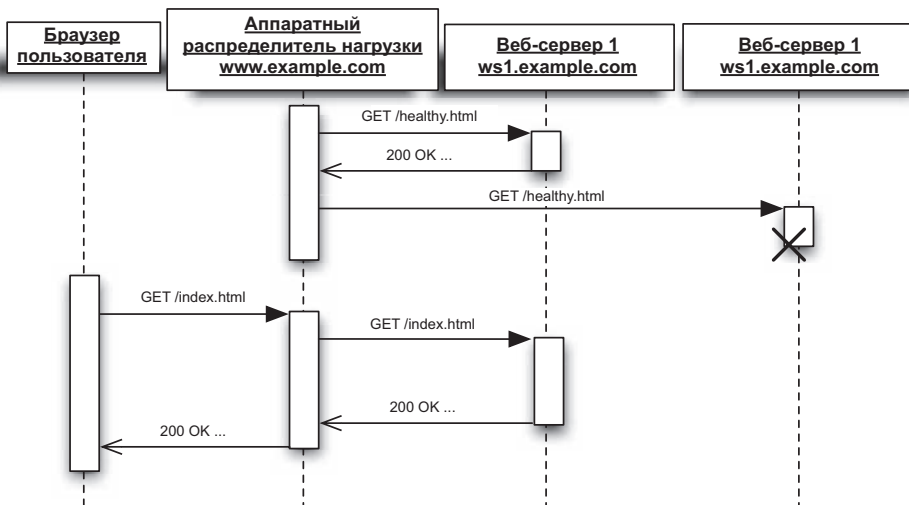


Рис. 31. Аппаратный распределитель нагрузки

Аппаратные распределители нагрузки умеют учитывать особенности используемых приложений и выполнять переключения в слоях с 4 по 7 стека OSI. Это означает, что они могут работать с любыми протоколами с установлением соединения, а не

только с HTTP или FTP. Я встречал примеры их успешного применения с группой поисковых серверов, не имевших собственных диспетчеров нагрузки. Еще они умеют передавать трафик от одного сайта к другому, что особенно полезно, когда требуется переход на резервную версию сайта для аварийного восстановления.

Проблемой для любого из этих решений становится протокол SSL. Аппаратные распределители нагрузки часто используются как «SSL-ускорители». Честно говоря, с моей точки зрения, это нарушает как закон Мура (каждые восемнадцать месяцев веб-серверы могут заменяться более быстрыми), так и принцип масштабируемости; как-никак, веб-серверов получается больше, чем аппаратных распределителей нагрузки. Более того, веб-серверы, скорее всего, выполняют не очень большую работу, в то время как распределитель нагрузки участвует в судьбе каждого запроса. Применение распределителя нагрузки для расшифровки SSL упрощает управление сертификатами SSL, так как для обновления остается всего пара устройств, а не десяток веб-серверов. По сути, если распределители нагрузки обрабатывают набор виртуальных IP-адресов для разных доменов, все сертификаты SSL можно поместить на одну пару машин (при условии, что они в состоянии справиться с нагрузкой!).

Слой защищенных сокетов (Secure Sockets Layer, SSL) — это соединения, зашифрованные для сохранения конфиденциальности и целостности. SSL относительно сильно нагружает процессор.

Аппаратные распределители нагрузки могут предоставлять также ряд служб прикладного уровня, в том числе службы проверки кодов HTTP-ошибок и перенаправления на различные страницы ошибок в зависимости от ответа сервера. В случае протокола HTTPS у вас есть два варианта действий: можно воспользоваться распределителем нагрузки для SSL и сохранить службы прикладного уровня, а можно прервать SSL-соединения с веб-серверами, в основном используя распределитель нагрузки как управляемый коммутатор и остановив службы прикладного уровня. При попытке передать SSL-соединения непосредственно веб-серверам с одновременным исследованием контента распределитель нагрузки, по сути, осуществит активное вмешательство в зашифрованный канал, вломившись между браузером и сервером.

Серьезным недостатком этих машин является их цена. Приготовьтесь к тому, что даже за минимальную конфигурацию придется выложить пятизначную сумму. За профессиональные конфигурации, необходимые солидным торговым фирмам, цена может возрасти до шестизначной.

РАСПРЕДЕЛЕНИЕ НАГРУЗКИ С АРАСНЕ 2.2

Начиная с версии 2.2 сервер Apache включает в себя новый модуль `mod_proxy_balancer`. Это крайне умный модуль распределения нагрузки, обладающий дополнительными возможностями. Он может «регулировать» нагрузку пропорционально мощности расположенных за ним веб-серверов, причем, пропорции вы выбираете самостоятельно.

Сервер Apache работает на уровне 7 модели OSI¹ — это приложение. Специализированная аппаратура, как правило, функционирует быстрее и справляется с большими объемами. Но для многих сайтов возможность сэкономить 50 000 на покупке пары специализированных аппаратных распределителей нагрузки значит куда больше, чем более низкий уровень ограничителя. Кроме того, проекты не являются статическими, и как только сайт упрется в ограничение, связанное с распределителями нагрузки с открытым исходным кодом, ничто не мешает заменить их более мощным аппаратным решением.

13.4. Кластеризация

Распределение нагрузки не требует взаимодействия отдельных серверов. Серверы, которые знают о существовании друг друга и активно принимают участие в распределении нагрузки, формируют кластеры. Для этой цели используются кластеры модели *активный-активный*. Еще кластеры можно применять для обеспечения избыточности на случай отказа. Такая модель, которая называется *активный-пассивный*, означает, что один сервер справляется со всей нагрузкой, пока не выйдет из строя, после чего работу возьмет на себя второй сервер, становясь активным.

Фермы с полностью распределенной нагрузкой масштабируются практически линейно. О кластерах с распределенной нагрузкой этого сказать нельзя. Кластеры тратят некоторое количество ресурсов на передачу сердцбиений и синхронизацию состояния. Из-за этих издержек вычислительная мощность кластера масштабируется немного нелинейно и может стать почти горизонтальной по мере увеличения числа серверов. Это накладывает практическое ограничение на размер кластера, хотя точный предел зависит от объединяемых в кластер серверов.

Сердцебиения (heartbeat) — это пакеты, передаваемые между соединенными в кластер серверами (обычно в отдельной сети) и сообщающие: «Я еще живой».

Некоторые приложения обладают встроенной кластеризацией. У WebSphere, WebLogic, Oracle и Microsoft SQL Server есть собственные технологии кластеризации. С их помощью прикладное программное обеспечение координирует собственную доступность и обработку ситуаций отказа. В таких приложениях неизбежно присутствуют узлы «центрального управления», отвечающие за передачу работы разным узлам кластера. Эти управляющие узлы могут оказаться слабыми местами архитектуры — например, если такой узел всего один — и обычно становятся первыми узлами, сталкивающимися с ограничениями мощности.

Приложения, не имеющие собственного механизма кластеризации, можно запускать под управлением таких решений, как Cluster Server от Veritas или Windows Server

¹ См. https://ru.wikipedia.org/wiki/Сетевая_модель_OSI. — *Примеч. пер.*

Clustering Services (whew) от Microsoft. Эти решения работают своего рода внешним каркасом, запуская объединенные в кластер приложения как дочерний процесс. Такие кластерные серверы передают сердцбиения друг другу и часто используют «кворумный том» на сетевом диске для синхронизации своей деятельности. При обнаружении сбоя в каком-нибудь узле такие серверы запускают заранее заданную последовательность операций для возвращения объединенных в группу приложений на функционирующий узел. Эти операции включают в себя различные варианты замены одной или нескольких файловых систем, запуска приложения «с нуля» или активации приложения из образа в памяти и смены виртуального IP-адреса.

У меня неоднозначное мнение о кластерных серверах. Они великолепны, но вместе с тем содержат ошибки. Они могут добавлять избыточность и отказоустойчивость приложениям, которые для этого не предназначены. Настройка такого сервера представляет собой крайне кропотливое занятие, а приложения обычно испытывают некоторые затруднения при переключении. Вероятно, самым большим их недостатком является запуск в режиме активный-пассивный. В результате избыточность достигается, а масштабируемость — нет. Я считаю кластерные серверы своего рода лейкопластырем для приложений, не способных обеспечивать кластеризацию своими силами.

14 Администрирование

Если вашей системой легко управлять, она будет хорошо работать. Более того, вы легко сможете получать помощь и ресурсы производственного процесса. Если же управлять вашей системой сложно и неприятно, она останется без присмотра и, возможно, будет некорректно реализована. Она может даже оказаться неработоспособной.

Работа администратора сложна и неблагодарна. С ним практически никогда не консультируются на этапе проектирования и выбора архитектуры системы. Вместо этого он получает наполовину готовое программное обеспечение, которое каким-то образом нужно адаптировать к обычным операциям.

Существует и другой, более фундаментальный конфликт. Разработчики и пользователи смотрят на изменения положительно. Каждая новая версия означает новые функциональные возможности — с соответствующим увеличением дохода — исправление ошибок и конструктивные усовершенствования. Для администраторов все ровно наоборот. Им приходится делать дополнительную работу по внедрению новой версии. Более того, после этого система начинает вести себя не так, как раньше. Исчезают старые сообщения журналов или команды, заменяясь новыми. Старые режимы отказов могут оказаться исправленными, но вместо них порой появляются новые, которые пока никто не умеет распознавать. Когда каждый урок дается потом и кровью, поневоле становишься крайне консервативным. И при этом оба мнения верны одновременно!

Администраторы могут быть как важными союзниками, так и серьезными врагами. (В конце концов, они, скорее всего, общаются с руководителями отдела ИТ намного чаще вас.) Впрочем, чтобы завоевать их лояльность, не потребуется ни взятка, ни клятва верности. Достаточно понять их мотивацию и действовать так, чтобы облегчить им жизнь. Когда выполнять работу проще, человек с ней лучше справляется, поскольку может и хочет это делать.

В этой главе мы поговорим о том, как сделать вашего администратора счастливым, создав более простое в управлении программное обеспечение.

14.1. «Совпадают ли условия тестирования с условиями эксплуатации?»

Хотел бы я, чтобы мне давали доллар каждый раз, когда я слышу вынесенный в заголовок вопрос. При каждом неудачном внедрении или проявившейся во время эксплуатации ошибке меня начинают спрашивать, почему проблема не была обнаружена на этапе тестирования. Этот вопрос важно задавать и искать на него ответ. В девяти случаях из десяти непосредственно за этим вы услышите следующий вопрос: «Различается ли конфигурация системы на этапах тестирования и эксплуатации?»

Этот вопрос легко задать, но ответ на него требует изрядных усилий, и *какое-то* отличие обязательно будет обнаружено, например имена хостов и IP-адреса или что-то еще. Возможно, вы недоумеваете: «Почему ответ на этот вопрос требует усилий? Разве не для этого предназначены системы управления конфигурациями?» И да, и нет. Системы управления конфигурациями могут точно сказать вам, когда были сделаны изменения, кто их сделал, а иногда и по каким причинам. Некоторые наиболее совершенные версии оснащены действующими модулями, позволяющими произвести опрос систем и идентифицировать различия. Однако проблема в том, что нам нужно определить, какие из расхождений известны, ожидаемы и безвредны, а какие неожиданны и рискованны. К примеру, при тестировании качества повсеместно используют сокращенный набор оборудования. Некоторые производители — как правило, это производители интегрированных аппаратно-программных комплексов, такие как HP, — предоставляют исправления фирменного программного обеспечения, исправления драйверов устройств, обновления функциональности операционной системы и исправления уязвимостей в рамках одного пакета. В этих случаях различные модели аппаратного обеспечения могут потребовать разных наборов исправлений.

В то же время вы всегда можете сказать, что среда тестирования *должна* отличаться от среды эксплуатации. В конце концов, если бы совпадало все, вплоть до имен хостов и номеров портов, это была бы не среда тестирования, а среда эксплуатации!

Мой опыт показывает, что наиболее распространенные причины отказов связаны отнюдь не с конфигурацией. Я провел много часов за обработкой конфигурационных файлов, проверяя ожидаемые и выявляя неожиданные отличия. Изредка мне удавалось накопить что-то примечательное, но, как правило, отличия в конфигурации были ни при чем.

Зачастую реальным виновником оказывалось несовпадение топологии сред тестирования и эксплуатации. Что я имею в виду под термином *топология*? Это возможность подключения, а также число серверов и приложений. Если каждый экземпляр сервера и приложения рассматривать как узел, как каждую связь или зависимость, как ребро, можно нарисовать граф, представляющий топологию системы. UML-диаграммы развертывания являются полезным, но недостаточно распространенным средством ее описания. Очевидно, что основным препятствием на пути совмещения топологий сред тестирования и эксплуатации является цена. Далее мы рассмотрим способы, которые позволяют решить данную задачу, не пробивая дыру в бюджете.

Делай их раздельными

Часто приложения во время тестирования пользуются одним и тем же хостом, а потом этого не наблюдается. Это может стать причиной скрытых зависимостей: два приложения будут ожидать синхронизированного контента папки. Они и в процессе тестирования-то работали только потому, что имелась всего одна папка. А после перехода к эксплуатации механизма, поддерживающего синхронизацию этих данных, не будет. (Демон `stop`, в последнюю минуту запускающий программу `rsync`, не в счет!) Если исходить из того, что вы не можете просто взять и купить для тестирования столько же серверов, сколько используются во время эксплуатации, что можно сделать, чтобы избежать подобной скрытой зависимости?

Для таких случаев я рекомендую VMware. Эта программа позволит создать нужное количество виртуальных хостов на одном физическом компьютере. Каждая виртуальная машина будет выглядеть и действовать как независимый сервер с собственными операционной системой, IP-адресом и именем хоста. Если два приложения в процессе эксплуатации должны запускаться на разных машинах, а при тестировании вынуждены делить аппаратное обеспечение, запустите на этой машине два экземпляра VMware. (Заодно это позволит вам сохранять снимки отдельных версий приложений, что очень поможет при тестировании процесса развертывания!)

Ноль, один, много

Существует старая поговорка, что единственными осмысленными числами в компьютерной науке являются 0, 1 и много. Есть фундаментальная разница между соотношениями «один к одному» и «один ко многим». Иногда на этапе тестирования вы используете всего один экземпляр там, где в процессе эксплуатации работает целая группа экземпляров. К примеру, именно это создает разницу между стратегией устаревания кэша между отдельными хостами и стратегией с использованием групповых или широковещательных запросов. Это еще один случай, когда сервер виртуализации, например VMware, позволяет на этапе тестирования симитировать многообразие режима эксплуатации. Если рабочая версия системы включает в себя десяток экземпляров, вовсе не обязательно запускать при тестировании ровно десять экземпляров (тем не менее обратите внимание на замечания о пропорциональности). Но при этом вам, без сомнения, следует запустить более одного экземпляра.

Просто купи эту штуку

Я видел часовые простои, возникавшие из-за появления в рабочей системе фаерволов или распределителей нагрузки, которых не было на этапе тестирования. При этом стоимость простоя превышала стоимость недостающего фрагмента сети. Для тестирования вам вряд ли потребуется модель высокого класса, но это должен быть продукт того же производителя и из той же линейки. В конце концов, как бы вы без него тестировали изменения конфигурации фаервола?

ВЫ РАБОТАЕТЕ ТАК, КАК ТРЕНИРУЕТЕСЬ

Спортивные команды на выступлениях и тренировках делают примерно одно и то же. Аналогично, если группа разработчиков с первого же дня использует в архитектуре брандмауэры, соответствующим образом она будет подходить и к проектированию. Нет ничего ужаснее ситуации, когда ты сидишь с готовой на 95 % системой и пытаешься вспомнить все правила фаервола, которые позволят начать эксплуатацию функции. А вот если команда разработчиков все это время работала с фаерволами, все правила у них уже документированы.

Отслеживание правил фаервола идет рука об руку с отслеживанием точек интеграции (см. раздел 4.1). Зачем в этом месте требуется разрешающее правило фаервола, если не для обращения к другой системе? Это точка интеграции. Дополнительное преимущество: именно эти конфигурационные параметры вам, скорее всего, придется менять при переходе в режим эксплуатации.

14.2. Конфигурационные файлы

Среда разработки Ruby on Rails может сколько угодно поддерживать соглашения по поводу конфигурации, но каждый встречавшийся мне фрагмент корпоративного программного обеспечения содержал множество конфигурационных файлов, включавших в себя имена хостов, номера портов, местоположения файловых систем, номера идентификаторов, магических ключей, имен пользователей, паролей и лотерейных номеров. (Последний я создал лично.) Перепутайте какой-либо из этих параметров, и система работать не будет. И даже когда кажется, что система работает, это может быть не так, просто никто об этом не знает!

Конфигурационные файлы часто имеют непонятные имена, скрываются где-то в глубине каталогов исходного кода или попросту непостижимы. Когда свойству присваивают имя `hostname`, что это значит? «Имя моего хоста», «имя авторизованной вызывающей функции» или «хост, к которому я обращаюсь в день летнего солнцестояния?» Файлы свойств перенасыщены скрытыми ссылками и избыточной сложностью — двумя основными факторами, ведущими к ошибкам управления. Эти файлы содержат наиболее важную информацию в масштабе всего предприятия (исключая сведения о зарплатах высшего руководства) — пароли доступа к рабочей базе данных. Поэтому они весьма конфиденциальны, и случайные ошибки в других частях системы будут искать уже не разработчики, понимающие, в чем разница между «`wsdlServer`» и «`uddlServer`».

Одной из наиболее распространенных ошибок проектирования конфигурационной схемы является объединение рабочей конфигурации и основных связующих компонентов. К примеру, среда Spring прекрасно подходит для связывания компонентов EJB, но при этом требует сохранять конфигурацию в одном файле¹. В этот файл

¹ Существуют способы превращения таких файлов в набор модулей, но, с моей точки зрения, решение проблемы в данной ситуации выглядит так же плохо, как и сама проблема.

входят не только свойства, которые должны меняться при переходе от разработки к эксплуатации, но и жизненно важные детали, например касающиеся создания экземпляров объектов и соотношения между объектами. В результате, чтобы поменять один пароль к базе данных, администраторам приходится вручную редактировать файлы XML, состоящие из 5000 строк, из которых 4999 являются минами замедленного действия, только и ждущими, чтобы их кто-то случайно изменил. При этом выясняется, что все это ужасное «внедрение зависимостей» жизненно необходимо для работы приложения. Сделайте ошибку, случайно отредактировав что-нибудь, отличное от пароля, и приложение прекратит работу по непонятной причине. Возможности для непредсказуемого нанесения ущерба бесконечны.

У администратора в принципе не должно быть шанса на вмешательство в связи объектов внутри приложения. Важная внутренняя информация должна находиться вне пределов его досягаемости. По возможности храните рабочие конфигурационные параметры отдельно от базовых параметров среды исполнения. Они должны находиться в разных файлах, чтобы администраторы не могли случайно внести изменения во внутреннее устройство системы.

Не менее важно то, что рабочим конфигурационным файлам не место в каталоге установки приложения. С большой вероятностью при следующем обновлении содержимое этого каталога изменится. Вы действительно рассчитываете на то, что администратор помнит все внесенные в конфигурацию за последние шесть месяцев изменения и после установки обновления сможет вручную восстановить статус-кво? Я уже не говорю о том, что администраторы повсеместно копируют установленное ПО целиком с одного сервера на другой, избавляясь таким образом от долгого и нудного процесса установки. (Например, первым шагом при установке сервера приложений BEA WebLogic 9 является распаковка файла объемом 500 Мбайт. Только это на машине x86 с процессором 2 ГГц занимает 10 минут.) Операции резервного копирования и восстановления тоже работают против вас. При восстановлении с ленты легко можно записать поверх актуальной рабочей конфигурации куда более старую версию.

Когда одно приложение запускается на нескольких машинах, какая-то часть конфигурационных параметров будет совпадать, а какая-то нет. Эти параметры нужно хранить по отдельности, чтобы никому не пришлось спрашивать: «А они должны различаться?» Кроме того, опыт показывает, что полезно проводить периодическую проверку синхронизации машин в горизонтально масштабируемом слое. Даже при наличии четких процедур управления изменениями следует руководствоваться правилом: «Доверяй, но проверяй». Наконец, конфигурационные параметры являются частью пользовательского интерфейса системы. Того интерфейса, который система предоставляет самой игнорируемой аудитории: людям, поддерживающим ее ежедневную работоспособность. Поэтому имена параметров (свойств) должны быть достаточно четкими, чтобы пользователь (администратор) мог делать свое дело, не совершая «невынужденных» ошибок.

Существует полезная договоренность именовать свойства в соответствии с их функцией, а не их природой. Не нужно называть имя хоста `hostname`. Это все равно, что

присвоить переменной имя `integer`, потому что она содержит целое число, или имя `string`, потому что она содержит строку. Такой подход, конечно, отражает истинное положение дел, но не помогает понять назначение переменной. Вместо этого лучше выбирать такие имена, как, к примеру, `authenticationServer`. При виде такой переменной администратор будет искать протокол LDAP или хост Active Directory.

В том, что касается управления конфигурацией, наиболее популярные Java-серверы приложений вызывают смешанные чувства. Серверы WebLogic, WebSphere и JBoss обладают великолепным графическим интерфейсом на базе HTML, дающим доступ к редактированию конфигураций.

Серверы WebSphere и WebLogic надежно идентифицируют свойства, принадлежащие не системе, а приложению. В JBoss они несколько перемешаны. Но во всех случаях конфигурационные файлы находятся в каталоге установки сервера приложений. У WebSphere, по крайней мере, существует такое понятие, как «конфигурационный репозиторий», который может использоваться как основная копия свойств и источник развертывания при переносе приложений из одной среды в другую. Однако при этом отсутствуют специальные места для хранения свойств, связанных с приложением, и свойств уровня системы.

Чуть лучше картина у .NET-разработчиков, но только в случае использования для управления конфигурациями последней версии Visual Studio. В то же время только самые смелые (и наиболее отчаянные) разработчики будут пытаться заняться .NET-разработкой без Visual Studio.

ДЖО СПРАШИВАЕТ: ПОЧЕМУ БЫ НЕ ПОМЕСТИТЬ КОНФИГУРАЦИОННЫЕ ФАЙЛЫ В СИСТЕМУ КОНТРОЛЯ ВЕРСИЙ?

Давным-давно программисты придумали для себя решение проблемы потерянных файлов и пропущенных изменений — систему контроля версий. Если конфигурационные файлы представляют такую проблему для администраторов, почему для работы с ними нельзя использовать систему контроля версий?

Некоторые так и делают. В кое-каких наиболее совершенных вычислительных центрах администраторы выполняют управление и развертывание конфигураций при помощи таких инструментов, как Subversion¹ и cfengine². Все большее число ИТ-отделов для управления версиями прибегает к таким вариантам ПО администрирования, как OpenView от HP, BladeLogic и Opsware.

Тем не менее зачастую администраторы при решении ИТ-задач не имеют времени на внедрение и запуск системы контроля версий или же просто не знают о такой возможности. Некоторые слышали об управлении версиями, но никогда не думали об этом применительно к системным конфигурационным файлам.

Как бы то ни было, лично я рекомендую задействовать систему контроля версий для управления конфигурационными файлами, но с рядом ограничений:

- ♦ Используйте безопасный репозиторий. Именно в этих файлах хранятся пароли от базы данных!

¹ См. <https://ru.wikipedia.org/wiki/Subversion>. — *Примеч. пер.*

² См. <http://cfengine.com/> и http://www.opennet.ru/docs/RUS/cfengine_v2/. — *Примеч. пер.*

- ◇ Свяжите управление версиями с более глобальным процессом управления изменениями. Вы должны видеть, почему имело место изменение конфигурации, а не только констатировать сам факт изменения.
- ◇ Автоматизируйте развертывание авторизованных изменений непосредственно из репозитория.
- ◇ Создайте автоматизированный процесс аудита. Иногда файлы меняются в процессе урегулирования инцидента с целью восстановления работы службы в кризисной ситуации. При этом случается и так, что позже они не обновляются системой контроля версий. Вы должны быть в состоянии обнаружить все отклонения без их автоматического перезаписывания.

В качестве альтернативы вы можете купить и развернуть у себя двадцать семь продуктов от BMC, которые будут делать все перечисленное за вас.

14.3. Начало и завершение работы

Когда разработчик запускает приложение и возникает отказ, разработчик останавливает приложение и исправляет ошибку. Когда компьютер перезагружается глубокой ночью, а отказ приложения происходит при попытке его загрузки, узнать об этом можно только в случае, если приложение само сообщит о проблеме. Если система мониторинга (см. главу 17) уже работает, это может быть обычная ошибка в файле журнала. Разумеется, чтобы приложение могло сообщить о том, что оно не смогло корректно стартовать, оно первым делом должно само понять, что что-то пошло не так.

В приложение следует встроить четкую последовательность запуска, чтобы перед тем, как оно начнет выполнять работу, гарантировать, что все компоненты стартовали в правильном порядке и что загрузочная последовательность успешно выполнена. Эта последовательность может быть связана, например, с сокетами, но пока не будет «повернут главный рубильник», соединения устанавливаться не должны. Представьте себе магазин, который утром готовится к открытию. Вы не начнете запускать покупателей внутрь только потому, что один из сотрудников уже пришел на работу. Вы подождете, пока все не будут стоять на рабочих местах, готовые корректно обслужить клиентов.

Не принимайте подключений до завершения процесса запуска.

Если приложению требуется пул соединений, по меньшей мере несколько соединений должно быть инициализировано на этапе загрузки приложения. Это одна из форм быстрого отказа (этот паттерн мы рассматривали в разделе 5.5), позволяющая быстро провести самотестирование. Ведь приложение без базы данных не сможет толком проводить транзакции! Соответственно, если по причине невозможности установок соединений произойдет сбой инициализации пула соединений, все приложение окажется неработоспособным. Обращаю ваше внимание, что ситуация *коренным* образом отличается от прерывания и выхода во время запуска в случае

отказа. Работающее приложение может быть опрошено на предмет своего внутреннего состояния (см. главу 17), а остановленное — нет.

Аналогичным образом для готовности к эксплуатации важно четкое завершение работы. Как наш воображаемый хозяин магазина не будет запира́ть дверь, если по магазину все еще бродят покупатели, так и работа приложений не должна заканчиваться грубым прерыванием. Каждому приложению необходим режим, в котором оно завершает все существующие транзакции, но не берется ни за какие новые задачи. После завершения последней транзакции приложение может закончить работу. Обязательно добавьте к этому правилу таймаут (см. раздел 5.1), чтобы не попасть в ситуацию, когда работа приложения никак не может завершиться.

14.4. Административные интерфейсы

Великолепные графические Java-интерфейсы выглядят очень здорово. Они придают программному обеспечению более «корпоративный» вид. К сожалению, в процессе эксплуатации они превращаются в настоящий кошмар. Основной проблемой при работе с графическим интерфейсом являются эти проклятые щелчки мышью. Я не могу написать сценарий для кучи щелчков. Графические Java-интерфейсы замедляют работу, заставляя администраторов вручную выполнять один и тот же процесс на каждом сервере (а их может быть много!) каждый раз, когда в этом возникает необходимость. Например, последовательность чистого выключения в одной системе управления заказами, с которой мне пришлось работать, требовала щелчка — и ожидания в течение нескольких минут — на каждом из шести серверов. Угадайте, насколько часто в этой системе выполнялось чистое выключение? Если обновление окна занимает целый час, я не могу позволить себе половину этого времени сидеть в ожидании обновления GUI.

Удаленный доступ при применении графических Java-интерфейсов также порой становится нетривиальной задачей. Администраторы часто обращаются к своим машинам по запутанному маршруту из SSH-тоннелей. Маршрутизация как X-, так и HTTP-подключений по таким туннелям — сама по себе большая головная боль. Здесь нет места дополнительным усложнениям. То же самое для чистой перезагрузки — я предсказываю огромное количество команд `kill -9`.

В сухом остатке мы получаем тот факт, что графические пользовательские Java-интерфейсы серьезно осложняют жизнь администраторам при долговременной эксплуатации системы. Самым лучшим интерфейсом для долговременной работы остается командная строка. Она легко позволяет создавать базу для сценариев, протоколирования и автоматизированных действий, поддерживающих работу вашего программного обеспечения. Большим шагом назад, хотя и вполне приемлемым, является административный GUI на базе HTML. Так как в языках Perl и Ruby существуют замечательные клиентские библиотеки для HTTP, написание сценариев в таком административном интерфейсе не превращается в сложную задачу.

15

Заключение по теме проектирования

Привлечь внимание к вопросам, рассматриваемым в этой части книги, особенно в шуме и суете разработки проекта, тем более если дело доходит до отказа, крайне сложно. В данном случае у меня для вас две новости: хорошая и плохая; вы можете вообще не думать об этих вещах на этапе разработки. Но вам придется столкнуться с ними на этапе эксплуатации... а это время и еще раз время. Решение данных вопросов в процессе разработки вовсе не обязательно отнимет у вас много времени или усилий, и в любом случае долговременные издержки из-за их игнорирования обойдутся вам в куда большую сумму.

Помните, что ваше приложение будет запускаться на сервере с несколькими сетевыми интерфейсами. Убедитесь в том, что оно связывается с корректным адресом для всех сокетов, которые оно слушает, а также что все особые требования к маршрутизации соблюдены и документированы. Административные функции не должны быть доступны ни в сетях для администрирования и мониторинга, ни в рабочей сети.

Убедитесь, что для доступа к кластерным службам, предоставляемым другими системами, например к серверам баз данных или к веб-серверам, используются виртуальные IP-адреса. Применение этих адресов позволяет поставщикам услуг осуществлять переключение на резервные мощности — как плановое, так и незапланированное — без необходимости менять конфигурацию вашей системы.

Приложения должны быть в состоянии работать в режиме обычного пользователя, не требуя привилегий администратора или суперпользователя. Конфиденциальные

конфигурационные параметры, например пароли от баз данных или ключи шифрования, следует хранить в отдельных конфигурационных файлах.

Не все системы требуют доступности, выражаемой пятью девятками. Затраты на обеспечение доступности резко возрастают при переходе к каждому следующему уровню. Обсуждение с заказчиками требований доступности как компромисса между затраченными средствами и полученной выгодой помогает найти общий язык.

Я предпочитаю оговаривать не доступность системы как целого, а доступность определенных программных компонентов или функций, выполняемых системой. Обязательно оговорите случаи, когда причиной потери доступности становятся внешние системы.

Распределение нагрузки и кластеризация создают предпосылки к высокой степени доступности. Можно задействовать решения с *разным* диапазоном цен. Вы можете по мере необходимости применять решения, включающие в себя распределение нагрузки и кластеризацию, стараясь сочетать выполнение требований к доступности с экономической эффективностью. Каждое такое решение имеет свой набор влияющих на ситуацию факторов, поэтому определение архитектуры с высокой степенью доступности на ранних этапах проекта значительно облегчает разработку и развертывание.

Администраторы, которые будут работать с вашим приложением, никогда не будут знать его внутреннее устройство так же подробно, как и вы. Вы можете снизить вероятность ошибки оператора, сделав процесс выбора конфигурации приложения очевидным. Это означает отделение среды исполнения, например файлов `beans.xml` компании Spring, от конфигурации, связанной с окружением. Их смешение эквивалентно размещению кнопки катапультирования рядом с кнопкой включения радио. Рано или поздно случится плохое.

Потратьте время и сделайте свое приложение простым в управлении. Его запуск и остановка должны быть не заметны пользователям, а задачи администрирования должны допускать создание автоматизированных сценариев. Симпатичные графические интерфейсы для Java-администрирования помогут новичкам, но вряд ли кому-то захочется без конца щелкать мышью.

Часть IV

Эксплуатация

16

Феноменальная мощь и маленькое жизненное пространство

16.1. Время максимальной нагрузки

В середине 1500-х годов итальянский врач Алоизий Лилий предложил новый календарь, чтобы исправить ошибку в привычном юлианском календаре. Эта ошибка была накапливающейся. Из-за нее через несколько сотен лет день весеннего равноденствия в календаре оказался бы на несколько недель раньше реального события. В календаре Лилия сложная система исправлений способствовала тому, что даты равноденствий и солнцестояний более-менее соответствовали астрономическим событиям. За 400-летний цикл календарные даты стали различаться на 2,25 дня, но это различие является предсказуемым и периодическим; кроме того, ошибка не циклическая и, соответственно, не накапливается. Этот календарь, введенный римским папой Григорием XIII, в конечном счете, пусть и не без борьбы, приняли все европейские страны, и даже Египет, Китай, Корея и Япония (последние три с модификациями). Некоторые народы стали пользоваться григорианским календарем еще в 1582 году, в то время как другие приняли его только в 1920-х годах.

Неудивительно, что этот календарь создала католическая церковь. Григорианский календарь, подобно большинству календарей, был создан, чтобы отмечать церковные

праздники (то есть выходные). После этого его стали применять для слежения за повторяющимися событиями в других областях, зависящих от годового солнечного цикла, например в сельском хозяйстве. Тем не менее ни один бизнес в мире не живет по григорианскому календарю. Бизнес-сообщество использует его даты в качестве удобного маркера для собственного внутреннего цикла деловой активности.

В каждой отрасли существует свой собственный внутренний календарь. В американской страховой компании год структурирован вокруг дней, когда сотрудники предприятий продлевают предоставляемые фирмой льготы, в том числе медицинскую страховку. Все планы строятся относительно этого периода. Мышление флористов сконцентрировано вокруг Международного женского дня и Дня святого Валентина. Случилось так, что эти дни отмечены в григорианском календаре особым образом, но в сознании флористов и работающих с ними поставщиков эти сезоны имеют собственное значение, не привязанное к календарной дате.

В торговле год начинается и заканчивается так называемым «сезоном праздников». В данном случае мы наблюдаем совпадение между различными религиозными календарями и календарем, применяемым в торговле. В США Рождество, Ханука и Кванза празднуются примерно в одно время. А так как слово «Рождествоханукакванза» невозможно выговорить с невозмутимым видом, на собраниях это время стали называть «сезоном праздников». Но пусть это радостное название не вводит вас в заблуждение. Интерес торговых работников к этому сезону носит сугубо прагматический характер. Ведь до 50 % годового дохода торговые фирмы получают в период с 1 ноября по 31 декабря.

В США торговый сезон праздников начинает День благодарения — четвертый четверг ноября¹. По давней традиции именно в этот момент население начинает массово задумываться о покупке подарков к Рождеству, так как до этого праздника остается чуть меньше 30 дней. Очевидно, что мотивация в данном случае никак не связана с датой в календаре религиозных праздников. Возникает покупательский ажиотаж, приводящий к таким явлениям, как Черная пятница. Магазины вдохновляют и подогревают всеобщий спрос, меняя ассортимент, увеличивая число распродаж и рекламируя удивительные вещи. Число покупателей в магазинах за сутки может увеличиться в четыре раза. Трафик в интернет-магазинах может возрасти на 1000 %. И это настоящее нагрузочное тестирование, единственное тестирование, которое имеет реальное значение.

16.2. Первое в жизни Рождество

Мой клиент запустил новый интернет-магазин летом. Недели и месяцы после этого события в очередной раз показали, что запуск нового сайта — все равно что

¹ Некоторые торговые сети лоббировали Конгресс, прося правительство сделать так, чтобы день Благодарения праздновался на две недели раньше.

рождение ребенка. Стоит готовиться к определенным вещам, например к тому, что вам придется просыпаться ночью и то и дело мучиться ужасными вопросами (например, «Боже мой! Чем кормить этого ребенка? Оранжевым пластилином?» или «Зачем они анализируют контент во время визуализации страницы?»). Тем не менее, несмотря на все проблемы, с которыми нам пришлось столкнуться, мы подошли к сезону праздников со сдержанным оптимизмом.

Наш оптимизм базировался на нескольких факторах. Во-первых, мы практически удвоили число рабочих серверов. Во-вторых, мы достоверно знали, что при текущем уровне нагрузок сайт работает стабильно. Несколько внезапных всплесков (в основном из-за неверно оцененных товарных позиций) предоставили нам возможность произвести измерения при пиковых отклонениях трафика. Эти пики были достаточно большими, чтобы мы смогли заметить, в каких именно местах начинает расти задержка открытия страниц. В итоге мы были осведомлены, при каком уровне нагрузки сайт начинает захлебываться. Третьей причиной для оптимизма была наша уверенность в том, что мы сможем справиться с любыми сюрпризами, которые преподнесет нам сайт. Внутренние резервы сервера приложений и встроенные в него инструменты предоставили нам больше возможностей для наблюдения и контроля, чем в любой другой системе, с которой мне доводилось работать. Но в конечном счете, оказалось, что есть разница между сложными, но успешными выходными в День благодарения и полным провалом.

Некоторым сотрудникам, работавшим в выходные перед Днем труда, был предоставлен небольшой отпуск. У меня появилось четыре свободных дня, чтобы вместе с семьей навестить моих родителей, живших в одном из соседних штатов, и совместно отпраздновать День благодарения. Заодно мы наметили суточные дежурства на сайте в течение этих выходных. Как я уже упоминал, мы были полны *сдержанного* оптимизма. Учтите, что мы представляли местную команду инженеров; основной центр управления сайтом (Site Operations Center, SOC), укомплектованный высококвалифицированными инженерами, дежурившими круглые сутки, — находился в другом городе. Обычно именно они отвечали за мониторинг и управление сайтом ночами и по выходным. Локальные инженеры были как бы дублерами SOC и всегда могли перенаправить в основной центр проблемы, решения которых не знали. Наша местная группа была слишком малочисленна для круглосуточной поддержки на постоянной основе, но мы придумали способ, позволяющий продержаться в таком режиме во время празднования Дня благодарения. И как бывший бойскаут («Будь готов!»), я на всякий случай взял с собой свой ноутбук.

16.3. Рука на пульсе

Когда в среду ночью я со своим ноутбуком прибыл в дом родителей, то сразу устроился в кабинете отца. Я могу работать везде, где есть широкополосный Интернет и сотовая связь. В данном случае у меня было кабельное широкополосное соединение

на 3 Мбайт, и я воспользовался Putty — моим любимым SSH-клиентом — для авторизации на машине jump host и запуска моих сценариев выборки.

Jump host — компьютер с высоким уровнем безопасности, который может через SSH-соединение подключаться к рабочим серверам.

Еще во время подготовки к запуску я принимал участие в нагрузочном тестировании данного сайта. Результат большинства тестов становился доступным после их завершения. Так как данные поставлялись генераторами нагрузки, а не брались из тестируемых систем, фактически мы работали по принципу «черного ящика». Чтобы получить больше информации, я воспользовался административным графическим HTML-интерфейсом сервера приложений, который позволял следить за такими важными показателями, как задержка, свободная динамическая память, активные потоки обработки запросов и активные сеансы.

Тому, кто заранее не знает, какие показатели его интересуют больше всего, GUI дает прекрасную возможность познакомиться с системой¹. Если же вы точно знаете, что вам нужно, GUI начинает тяготить. Более того, для слежения за тридцатью серверами одновременно трудно придумать более неподходящую вещь, чем GUI.

Чтобы получить больше информации от нагрузочных тестов, я написал набор Perl-модулей, которые анализировали административный GUI и интерпретировали значения HTML. Эти модули помогали мне получать и устанавливать значения свойств и вызывать методы как для встроенных, так и для созданных своими силами компонентов сервера приложений. Так как весь административный графический пользовательский интерфейс был создан на базе HTML, сервер приложений не видел разницы между Perl-модулем и веб-браузером. Благодаря этим модулям я смог написать набор сценариев, которые извлекали из всех серверов приложений образцы их статистических данных, выводили на печать детали и обобщающие результаты, некоторое время отдыхали и снова начинали свою деятельность.

Это были очень простые индикаторы, но просматривая эту статистику с момента запуска сайта, мы все получили представление о нормальном ритме жизни нашего детища. Мы с первого взгляда могли определить, какая ситуация являлась нормой в полдень июльского четверга. Мы сразу отмечали выходящее за границы привычного диапазона количество сеансов и некорректный вид счетчика размещенных заказов. Удивительно, насколько быстро вырабатывается настоящий нюх на проблемы.

¹ Тем не менее у этого конкретного сервера приложений ATG Dynaмо самый лучший административный графический интерфейс. Он выглядит не так красиво, как WebLogic или WebSphere, но демонстрирует все компоненты сервера приложений. Вы можете видеть все, как в оригинальном движке Volkswagen Beetle, насколько хорошо работает каждая часть, каким образом она связана с другими компонентами. ATG реализовал внедрение зависимостей задолго до того, как Мартин Фаулер ввел этот термин.

Технологии мониторинга¹ дают колоссальную поддержку, засекая проблемы в момент их появления, но в деле распознавания паттернов ничто не сравнится по силе с человеческим мозгом.

16.4. День благодарения

Проснувшись в День благодарения, я, даже не попив кофе, помчался в родительский кабинет, чтобы посмотреть окно статистики, которая собиралась в течение ночи. Мне пришлось посмотреть два раза, чтобы удостовериться, что глаза меня не обманывают. Количество сеансов ранним утром уже почти сравнялось с пиковыми показателями в самые загруженные дни обычной недели. Количество заказов было так велико, что я позвонил администратору базы данных и спросил, не отправляются ли заказы по два раза. Ничего подобного не было.

К полудню клиенты разместили столько же заказов, сколько обычно появлялось за неделю. Страница задержки — наш сводный показатель времени отклика и общей производительности сайта — явно показывала наличие нагрузки, но пока все в пределах нормы. Но лучше всего было то, что этот показатель оставался стабильным, несмотря на рост числа сеансов и заказов. Я чувствовал себя счастливым, отправляясь на обед с индейкой. К вечеру заказов стало больше, чем за период с начала месяца. К полуночи их число превысило показатель за целый октябрь — а сайт работал. Первый убийственный нагрузочный тест был пройден.

16.5. Черная пятница

На следующее утро, в Черную пятницу, я перед завтраком снова побрел в родительский кабинет, взглянуть на статистику. Количество заказов почти побило вчерашние показатели. Число сеансов было большим, но период ожидания страницы все еще не превышал 250 миллисекунд — установленную нами норму. Я решил погулять с мамой по городу, попутно купив ингредиенты для курицы карри. (В пятницу мы собирались обедать тем, что осталось от Дня благодарения, но в субботу я хотел приготовить карри, а наш любимый тайский рынок по субботам закрыт.)

Разумеется, если бы все было в порядке, эту историю я бы не рассказывал. Катастрофа, как назло, случилась, когда я был далеко от своей точки доступа. Звонок настиг меня на другом конце города.

«Привет, Майкл. Это Дэниел из центра управления сайтом».

«Кажется, ты хочешь сообщить мне что-то неприятное, не так ли?» — спросил я.

¹ Различные технологии мониторинга обсуждаются в следующей главе.

«В SiteScope все DRP-индикаторы красные. Мы по кругу их перезагружаем, но все сразу же падает. Дэвид организовал телефонную конференцию и просит тебя присоединиться».

В лаконичной манере, выработавшейся у нас за долгое время сотрудничества, Дэниел рассказал, что сайт упал, и ситуация крайне серьезная. Наш инструмент для внешнего наблюдения — показанная на рис. 32 программа SiteScope — обращается к сайту тем же способом, что и обычный пользователь. Красные индикаторы сразу дают понять, что клиенты не могут попасть на сайт и мы теряем доход. На сайте ATG¹ запросы к страницам обрабатывались экземплярами приложений, которые ничем другим не занимались. Веб-сервер общается с сервером приложений через протокол DRP (Dynamo Request Protocol), и именно эту аббревиатуру мы используем для обозначения экземпляров, отвечающих за обработку запросов. Выражение «красный DRP-индикатор» означает, что один из этих экземпляров прекратил отвечать на запросы страниц. А фраза «все DRP-индикаторы красные» — сигнал того, что сайт прекратил работу, и мы теряем заказы со скоростью примерно миллион долларов в час. «Перезагрузка по кругу» — это отключение и запуск серверов приложений с максимально возможной скоростью. Для поднятия всех серверов приложений на одном узле требуется около десяти минут. Можно поднимать одновременно четыре или пять узлов, но не более того, иначе база данных начнет реагировать медленнее, что, в свою очередь, замедлит процесс запуска. Словом, рабочая группа из всех сил пыталась удержаться на плаву, но все их попытки были тщетны.

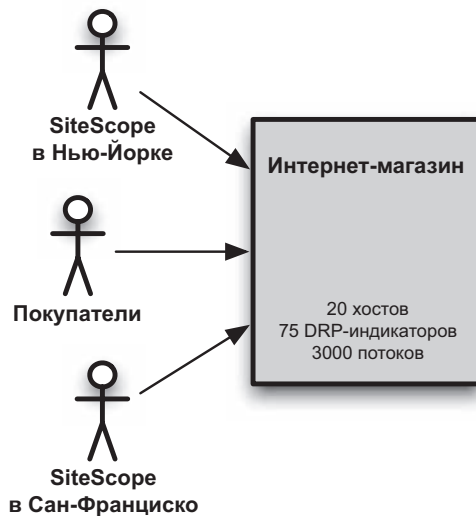


Рис. 32. Программа SiteScope использует интерфейс магазина, имитируя пользователей

¹ Сервер J2EE-приложений, хорошо подходящий для коммерческих приложений. См. <http://www.atg.com>.

Я ответил Дэниелу: «Хорошо, я сейчас позвоню, но до рабочего компьютера смогу добраться не раньше чем через полчаса».

Я позвонил и услышал шум голосов. В конференц-зале была включена громкая связь. Ничто не сравнится с попытками разобрать, о чем идет речь, когда в гулком конференц-зале одновременно говорят пятнадцать человек, попутно то и дело принимая звонки от людей, спешащих донести до нас крайне полезную информацию: «Сайт не работает». Да, мы знаем. Спасибо большое и до свидания.

16.6. Жизненно важные функции

Проблемы начались примерно за двадцать минут до того, как Дэниел мне позвонил. Оперативный центр обратился к команде, дежурившей по сайту, а руководитель проекта Дэвид решил привлечь меня.

Для нашего клиента слишком многое было поставлено на карту, чтобы волноваться о прерванных выходных. Кроме того, я сам просил звонить мне в случае необходимости.

На данный момент, спустя двадцать минут после инцидента, были известны следующие вещи:

- ❑ Счетчики сеансов показывали очень большие значения, превышающие вчерашние показатели.
- ❑ Загрузка полосы пропускания была высокой, но предельного значения пока не достигла.
- ❑ Время ожидания страниц сервера приложений (время реакции) было высоким.
- ❑ Загрузка процессора у веб-серверов, серверов приложений и сервера базы данных была небольшой.
- ❑ Серверы поиска, которые обычно были главными источниками проблем, отвечали хорошо. Статистика системы выглядела вполне приемлемо.
- ❑ Почти все отвечающие за обработку запросов потоки были заняты. Многие из них занимались своими запросами дольше пяти секунд.

На самом деле время ожидания страниц было не просто высоким. Оно было, по сути, бесконечным, поскольку запросы прерывались по таймауту. Статистика показывала нам только среднее значение завершенных запросов.

Оставшиеся незавершенными запросы при вычислении среднего не учитывались. Кроме долгого времени реагирования, о котором мы уже знали, так как программа SiteScore не смогла завершить свои искусственные транзакции, все остальное выглядело нормально.

Для получения дополнительной информации я стал исследовать дампы потоков вышедших из строя серверов приложений. Попутно я попросил одного из наших блестящих инженеров, находившихся в конференц-зале, проверить серверную часть системы управления заказами. Он обнаружил там точно ту же самую картину, что и на клиентской части: низкая загрузка процессора, но при этом большинство потоков постоянно занято.

Прошел почти час с момента, когда мне позвонили, или почти девяносто минут с момента падения сайта. Это означало не только потерянные нашим клиентом заказы, но и несоблюдение нами соглашения об уровне обслуживания в процессе урегулирования серьезного отказа. Я терпеть не могу, когда не соблюдаются соглашения об уровне обслуживания. И принимаю это близко к сердцу, как и мои коллеги.

16.7. Диагностические тесты

Дампы потоков серверов приложений с входной стороны показали одинаковую ситуацию на всех DRP-индикаторах. Несколько потоков пытались обращаться к внутренней части, а большинство оставшихся ожидали доступного соединения, чтобы сделать то же самое. Ожидающие потоки были заблокированы пулом ресурсов, не имеющим таймаута. В такой ситуации если внутренняя часть системы перестает отвечать, выполняющие запрос потоки никогда его не завершат, а у заблокированных потоков так и не появится возможность сделать запрос. Короче говоря, каждый из потоков, обрабатывающих запрос, а их было 3000, ничем не занимался, что идеальным образом объясняло низкую загрузку процессора: все 100 DRP-индикаторов простаивали в ожидании ответа, который никогда не придет.

Внимание сместилось к системе управления заказами. В этом случае дампы потоков показали, что некоторые из 450 потоков совершали запросы к внешней точке интеграции, как показано на рис. 33. Как вы, вероятно, уже догадались, все остальные потоки были заблокированы и ожидали возможности обратиться к этой внешней точке интеграции. Данная система отвечала за планирование доставки на дом. Мы немедленно послали сообщение отвечавшей за эту систему рабочей группе. (В этой группе поддержка в режиме 24/7 отсутствовала. Они по скользящему графику передавали друг другу пейджер.)

Кажется, примерно в это время жена принесла мне тарелку с индейкой, чтобы я пообедал. Между отчетами о состоянии системы я выкроил минутку, чтобы отключить телефон и перекусить. К этому моменту батарейка моего сотового телефона села, а батарейка беспроводного телефона была близка к этому состоянию. Обычный телефон я использовать не мог, так как к нему не подходила моя гарнитура. Я надеялся, что к моменту, когда умрет беспроводной телефон, сотовый успеет достаточно подзарядиться.



Рис. 33. Интернет-магазин обращается к системе управления заказами

16.8. Обращение к специалисту

Казалось, что прошла половина вечности (хотя, скорее всего, прошло не более получаса), когда к конференции присоединился инженер из службы поддержки. Он объяснил, что из четырех серверов, которые обычно отвечали за планирование заказов, два отправлены на техническое обслуживание, а один из оставшихся по неизвестным причинам функционирует некорректно. Я до сих пор не могу понять, почему техническое обслуживание серверов было назначено на эти выходные, радикально отличающиеся от всех прочих выходных! Рисунок 34 демонстрирует относительные размеры трех задействованных систем.



Рис. 34. Система управления заказами обращается к «корпоративной» системе планирования

Единственный оставшийся сервер до того, как он замедлил свою работу и завис, мог одновременно обработать до 25 запросов. Мы обнаружили это в момент, когда число запросов от системы управления заказами приближалось к 90. Разумеется, когда дежурный инженер проверил единственный работающий сервер, оказалось, что его процессор загружен на 100 %. Ему несколько раз посылали на пейджер сообщение о высокой загрузке процессора, но он на него не реагировал, так как прежде его группа уже не раз получала сообщения о внезапных увеличениях данного параметра, но каждый раз выяснялось, что это ложная тревога. Все эти ложные срабатывания эффективно приучили группу игнорировать состояние процессора.

Наш инвестор в процессе телефонной конференции мрачно проинформировал нас об объявлении отдела маркетинга, которое в пятницу утром должно было попасть в газеты. Объявление предлагало бесплатную доставку на дом всех заказов, совершенных до понедельника. Все участники, пятнадцать человек в конференц-зале и десяток звонящих по телефону, впервые за четыре часа замолчали.

Итак, напомним, что у нас была внешняя система, наш интернет-магазин с 3000 потоками на 100 серверах и коренным образом изменившаяся схема трафика. Это захлебывающаяся система управления заказами, с ее 450 потоками, разрывающимися между обработкой запросов от внешней части и обработкой заказов. И она, в свою очередь, заставляет захлебнуться систему планирования, которая едва в состоянии одновременно справиться с 25 запросами. И все это должно было продолжаться до понедельника. Кошмарный сценарий. Сайт лежит, а у нас нет схемы выхода из ситуации. Решение нужно было искать экспромтом.

16.9. Сравнение вариантов лечения

Начался мозговой штурм. Многие предложения отвергались, главным образом потому, что было непонятно, как себя при этом поведет код приложения. Быстро стало ясно, что единственным решением было прекращение запросов, проверяющих доступность системы планирования. Маркетинговая кампания, в рамках которой клиентам пообещали бесплатную доставку, не давала надежды на уменьшение числа пользовательских запросов. Но нужно было найти способ отрегулировать количество вызовов. Система управления заказами тут нам помочь не могла.

Луч надежды блеснул, когда мы посмотрели код магазина. Для организации подключений к системе управления заказами там использовался подкласс стандартного пула ресурсов. По сути, имелся отдельный пул подключений именно для запросов планирования. Я не знаю, зачем был спроектирован этот отдельный пул подключений, вероятно, это было проявление закона Конвея, но это спасло ситуацию. Предназначенный именно для этих подключений компонент можно было использовать в качестве регулирующего механизма.

Если бы разработчики добавили свойство `enabled`, осталось бы только присвоить ему значение `false`. Но мы придумали другой изящный выход. Пул ресурсов

с нулевым максимальным значением в любом случае работать не будет. Я спросил у разработчиков, что будет, если пул вместо подключения начнет возвращать значение `null`. Они ответили, что код обработает это значение и покажет клиенту вежливое сообщение, информирующее, что в настоящий момент система планирования доставки недоступна. Вполне приемлемый выход.

16.10. Есть ли реакция на лекарство?

Один из моих Perl-сценариев мог установить значение любого свойства у любого компонента. В качестве эксперимента я заставил этот сценарий присвоить переменной `max` данного пула ресурсов (всего на одном DRP-индикаторе) нулевое значение. Аналогичная операция была проделана с переменной `checkoutBlockTime`. Ничего не случилось. Поведение системы не изменилось. Тут я вспомнил, что переменная `max` считывается в момент загрузки пула.

Я воспользовался сценарием, умеющим вызывать методы компонентов, и вызвал методы `stopService()` и `startService()` пула. Вот оно! DRP-индикатор снова начал обрабатывать запросы! Это была огромная радость.

Но так как у нас отвечал только один DRP-индикатор, диспетчер загрузки стал пересылать ему все запросы страниц. В результате этот экземпляр просто снесло, как последний открытый киоск с пивом на чемпионате мира по футболу. Но теперь мы по крайней мере знали, что нужно делать.

На этот раз я запустил свои сценарии с флагом «для всех DRP-индикаторов». Они присваивали переменным `max` и `checkoutBlockTime` значение ноль и перезапускали службу. Возможность перезапуска компонентов, а не серверов целиком является ключевой концепцией вычислений, ориентированных на восстановление. Конечно, у нас отсутствовал предлагаемый ROC уровень автоматизации, но восстановить работу службы, не перезагружая всю систему, мы могли. Если бы нам пришлось редактировать конфигурационные файлы и перезагружать все серверы, при текущем уровне нагрузки этот процесс занял бы более шести часов. Динамическое изменение конфигурации пула подключений и его перезагрузка заняли меньше пяти минут (после того, как мы поняли, что нужно делать).

Как только мои сценарии завершили работу, мы увидели, что трафик пошел. Задержка страниц начала снижаться. Еще через девяносто секунд все DRP-индикаторы в программе SiteScope стали зелеными. Сайт заработал.

ВЫЧИСЛЕНИЯ, ОРИЕНТИРОВАННЫЕ НА ВОССТАНОВЛЕНИЕ

Вычисления, ориентированные на восстановление (Recovery-Oriented Computing, ROC), — это совместный проект университетов Беркли и Стэнфорд¹. Вот его основные принципы:

- Как аппаратные, так и программные отказы неизбежны.

¹ См. <http://roc.cs.berkeley.edu/>.

- ☐ Моделирование и анализ никогда не будут исчерпывающими. Заранее предсказать все режимы отказов невозможно.
- ☐ Основной причиной системных отказов является человеческий фактор.

Эти исследования противоречат большинству работ, посвященных надежности систем. В то время как в подавляющем большинстве случаев речь идет об устранении источников отказов, ROC признает их неизбежность — именно эта тема является центральной в моей книге! Исследования в рамках этого проекта способствуют сохранению работоспособности систем при отказах.

Современные языки и платформы программирования обеспечивают реализацию многих ROC-концепций. Следуйте их принципам сдерживания ущерба, автоматического распознавания ошибок и возможности перезагрузки на уровне компонентов, и вы получите большую выгоду.

16.11. Выводы

Я написал новый сценарий, выполнявший все, что требовалось для установки максимального значения пула подключений в нулевое состояние. Он задавал значение свойства `max`, останавливал службу, а потом заново ее запускал. Одной командой инженер в центре управления или в «командном пункте» (то есть в конференц-зале) мог установить нужное максимальное число подключений к сайту. Позднее я узнал, что мой сценарий использовался на протяжении всех выходных. Так как присвоение нулевого значения полностью отключало службу доставки на дом, инвестор попросил, чтобы этот параметр увеличивался при снижении нагрузки и уменьшался до 1, а не до 0, когда нагрузка становилась чрезмерной.

Телефонная конференция завершилась. Я повесил трубку и пошел укладывать детей спать. Это заняло некоторое время. Они хотели рассказать мне, как ходили в парк, играли с пульверизатором и обнаружили на заднем дворе крольчат. Я же жаждал узнать все подробности.

17

Прозрачность

Опытные судовые инженеры могут по звуку гигантских дизельных двигателей определить, что что-то идет не так. Жизнь рядом с двигателями научила их различать нормальные, допустимые и ненормальные ситуации. Но это тот самый случай, когда жить, не слыша звуков и ритмов окружающей среды, невозможно. И когда что-то выходит из строя, знание внутреннего устройства двигателя ведет инженера напрямик к месту возникновения проблемы настолько быстро и точно, что возникают мысли об экстрасенсорных способностях.

Наши системы не настолько открыты. Они заключены в безликие корпуса. Нет движущихся частей, за которыми можно наблюдать, а равномерный шум вентиляторов почти не дает информации о том, что происходит внутри. (Впрочем, *остановка* вентилятора сразу сигнализирует о проблеме!) Мы не сидим в одной комнате с компьютером, наоборот, можем находиться в другом здании и даже в другом городе. И чтобы обрести понимание рабочей среды, которое судовые инженеры получают естественным образом, нужно добавить к нашим системам *прозрачность*.

Понятие «прозрачность» относится к качествам, позволяющим операторам, разработчикам и инвесторам получать представление о текущих трендах системы, текущих условиях эксплуатации, мгновенном состоянии и прогнозах на будущее. Прозрачные системы умеют взаимодействовать и в процессе этого взаимодействия обучают обслуживающих их людей. Огромная дизель-генераторная установка на корабле дает информацию посредством звуков и вибрации, через датчики, предоставляющие количественные данные, а в экстремальных (обычно плохих) случаях и через запахи.

Решая проблему Черной пятницы (о ней я рассказывал в главе 16), мы полагались на визуальный контроль поведения системы на уровне компонентов. Этот визуальный контроль появился не случайно. Это был результат передовой технологии,

реализованной с учетом прозрачности и обратной связи. Без визуального контроля такого уровня, мы, скорее всего, получили бы только информацию о замедлении работы сайта (например, после звонка разгневанного пользователя или попытки члена рабочей группы зайти на сайт), но понятия не имели бы, почему это происходит. Это как заболевшая золотая рыбка в вашем аквариуме — вы ей ничем не можете помочь, и остается только ждать, выздоровеет она или умрет.

Прозрачность имеет практические и психологические достоинства. С практической точки зрения отладка прозрачной системы происходит значительно быстрее, поэтому прозрачные системы совершенствуются более оперативно, чем непрозрачные. Что касается психологической стороны дела, представьте, что раз в год нужно бросить кость, на грани которой нанесены слова «бонус», «продвижение», «увольнение», «много сверхурочной работы», «всеобщее презрение» и «признание экспертов», но при этом кость кидает кто-то другой. Вы никак не можете контролировать этот процесс. Будущее любого ответственного за успех системы — от разработчиков приложений до бизнес-аналитиков и инвесторов — зависит от поведения игровой кости, но никто не знает, каким оно в итоге будет.

Когда требуется увеличить вычислительную мощность, вы полностью зависите от данных, собранных в существующей инфраструктуре. Предсказание будущего невозможно без комбинации технических данных и бизнес-показателей, а также понимания состояния системы в прошлом и в настоящем. Хорошие данные позволяют принять правильное решение. Если достоверные данные отсутствуют, решения будут приниматься за вас на основе чьего-то политического влияния, предрассудков или манеры причисляться.

Наконец, без прозрачности система не может долго эксплуатироваться. Если администраторы не знают, что она делает, как они смогут ее настроить и оптимизировать? Если разработчики не знают, что работает на стадии эксплуатации, а что нет, как они могут постепенно повышать надежность системы и улучшать ее работоспособность? А если инвесторы не знают, получают ли они доход, станут ли они финансировать новые проекты? Без прозрачности система будет постепенно деградировать, с каждой новой версией функционируя все хуже и хуже. Системы могут совершенствоваться тогда и только тогда, когда являются до некоторой степени прозрачными.

В этой главе прозрачность будет обсуждаться с четырех точек зрения: исторический тренд, прогнозирование, текущее состояние и текущее поведение. Также мы рассмотрим существующие технологии, допускающие прозрачность и пробелы в технологиях, которые ждут своего заполнения. В случае веб-систем ряд поставщиков рассматривает возможности, касающиеся измерения производительности и отчетов. Сейчас наблюдается волна консолидации с целью создания интегрированных пакетов «управления приложениями масштаба предприятия». Это неисчерпаемая область, поэтому я больше чем обычно буду говорить об отдельных продуктах. Скорее всего, к моменту, когда вы будете читать эту книгу, многие из работающих в этой области компаний будут куплены, а их продукты интегрированы в более крупные программные пакеты.

17.1. Точки зрения

Разные специалисты по-разному смотрят на проблему. Разные точки зрения требуют разных взглядов на систему. Как вопрос «Какая стоит погода?» имеет разное значение для садовника, пилота и метеоролога, так как смысл вопроса «Как идут дела?» зависит от того, кто его задает — генеральный директор или системный администратор.

Исторические тренды

История — достаточно гибкое понятие. Впрочем, существует не так много областей, в которых словосочетание «на прошлой неделе» может относиться к истории. Возможно, это единственный аспект, объединяющий миры ИТ и высокой моды.

Тем не менее даже при столь близорукое подходе мы должны позаботиться о представлении исторических данных. Вопреки заявлениям в духе Уолл-стрит¹, предсказывать поведение систем путем экстраполяции ранее полученных результатов вполне возможно. Это касается как бизнес-показателей (покупатели, заказы, коэффициент обращаемости, доход, затраты на оказание услуг и т. п.), так и системных параметров (свободное пространство в хранилище данных, средняя загрузка процессора, пропускная способность сети, количество запрототолированных ошибок).

Очевидно, что данные за предшествующий период должны некоторое время где-то храниться. Лучше всего для этой цели подходит база данных. Она позволяет анализировать отклонения и тренды. Благодаря наличию в ней как системных показателей, так и бизнес-параметров, становится возможным поиск корреляций во времени и среди отдельных слоев.

Исторический тренд может быть представлен в виде электронной таблицы, диаграммы, презентации PowerPoint или аналитического отчета. Эти данные уже не являются актуальными и, как правило, не подходят для представления на панели инструментов (dashboard view).

Так как данные за предшествующие периоды могут использоваться для обнаружения новых и интересных взаимосвязей, они должны быть доступны для экспорта в Microsoft Access и Microsoft Excel. Если ваша компания обзавелась инструментами для бизнес-анализа (business intelligence) или генерации отчетов, они обязательно будут применяться к данным за предшествующие периоды.

Но не поддавайтесь соблазну предоставить этим инструментам непосредственный доступ к рабочей транзакционной базе данных (см. подраздел «Данные за прошедшие периоды» в разделе 9.8).

¹ Я имею в виду известное высказывание «Past performance is no guarantee of future results» («Былые достижения не являются гарантией будущих успехов»).

Вот пример распространенных вопросов, указывающих на исторический тренд:

- ☐ Сколько заказов мы приняли за вчерашний день?
- ☐ Сколько это в сравнении с этим же днем¹ год назад?
- ☐ Сколько дискового пространства мы использовали за первый квартал?
- ☐ Какая системы была узким местом при последнем всплеске трафика?
- ☐ Как в течение последних трех лет рост покупательского трафика соотносится с ростом загрузки процессора?

Предсказание будущего

В своей книге *Planning Extreme Programming* Кент Бек и Мартин Фаулер ввели в обиход выражение «вчерашняя погода». Это выражение отсылает нас к бородатой истории о сложной системе предсказания погоды, побежденной куда более простым алгоритмом: утверждение, что сегодня будет такая же погода, как и вчера, верно примерно в 70 % случаев. Каким бы необоснованным ни казалось это утверждение с точки зрения метеорологии, оно четко демонстрирует тот факт, что будущее можно предсказывать, экстраполируя прошлое.

В случае наших систем в роли хрустального магического шара выступают данные за предшествующие периоды. Предсказания будущего практически всегда ориентируются не на прямые измерения, а на корреляции и взаимосвязи. Вне всяких сомнений, чаще всего задается вопрос: «Какова вычислительная мощность системы?» (см. разделы 8.1 и 8.3). Кажется, что данный вопрос касается единственного параметра, но на самом деле это вопрос: «Окажут ли следующие N пользователей пропорциональное влияние на N предыдущих пользователей?»

Предсказания всегда базируются на модели системы. Неудачные прогнозы являются следствием плохой модели (например, линейных показателей). Разработать отличную модель сложно. Для этого требуются люди, прекрасно разбирающиеся в таких ужасных вещах, как теория массового обслуживания и стохастическое моделирование, — специалисты по теории исследования операций. Эти люди только на базе диаграмм архитектуры способны построить модель, предсказывающую производительность системы до трех знаков после запятой. Я не уверен, что подобный уровень усилий (и затрат) имеет смысл в такой хорошо изученной области, как веб-системы электронной коммерции. Мне кажется, что такие вещи нужны, к примеру, при моделировании глобальных систем связи.

В то же время можно разработать «достаточно хорошие» модели путем поиска корреляций в статистических данных, которые затем в определенных

¹ Следует заметить, что словосочетание «в этот же день» не всегда подразумевает аналогичную календарную дату — для согласования практически всегда выбирается тот же самый день недели, а не день месяца.

пределах¹ могут применяться для получения прогнозов. Эти корреляционные модели можно встраивать в электронные таблицы, позволяя менее технически подкованным пользователям выполнять сценарии вида «что будет, если». Прогноз на будущее, как правило, представляет собой информацию для ограниченного круга лиц. Кроме того, прогнозы не являются актуальными в отличие от оперативных данных. Соответственно, в панель инструментов эти модели не встраиваются.

Выход новой версии приложения может изменить или сделать недействительными корреляции, лежащие в основе прогноза. Так как подобные показатели функционируют как производные от поведения системы, после появления каждой следующей версии их следует пересматривать, чтобы понять, насколько они применимы в новых условиях. (Разумеется, вам придется подождать, пока не появится адекватный набор новых измерительных инструментов.) Кроме того, предполагается, что любые операции, основанные на этих прогнозах, должны снабжаться ссылкой, указывающей, каким именно набором показателей вы пользовались.

Вот общие вопросы, показывающие будущие перспективы:

- ☐ Сколько покупателей в день мы можем обслужить?
- ☐ Когда мне нужно покупать дополнительные серверы (или жесткие диски, или полосу пропускания, или любой другой ресурс)?
- ☐ Мы переживем этот сезон праздников? (Обратите внимание, что в данном случае требуется сделать прогноз по поводу прогноза, что повышает вероятность ошибки не в два, а в четыре раза².)

Текущее состояние

Разницу между текущим состоянием и текущим поведением проще всего проиллюстрировать на примере бегающего трусцой толстяка. Его текущее поведение ассоциируется с улучшением здоровья, но при этом он может находиться в одном шаге от сердечного приступа из-за перегрузок.

«Текущее состояние» относится к общему состоянию системы, причем не столько к тому, что система делает в настоящий момент, сколько к тому, что она *уже* сделала. Это касается состояния каждого аппаратного фрагмента, каждого сервера приложений, каждого приложения и каждого пакетного задания.

¹ Хотелось бы заметить, что законы движения Ньютона тоже применимы только в определенной области... но они смогли через двадцать лет полета привести космический зонд к планете Нептун.

² Я понимаю, что это академический взгляд. Я не жду скрупулезного соблюдения методологии прогнозирования от инвесторов, но надеюсь, что разработчики и системные инженеры будут ее соблюдать!

Состояние каждого компонента определяется комбинацией событий и параметров. События возникают в какой-то момент времени. Порой это нормальные и даже необходимые вещи, а порой — отклонения, вызывающие беспокойство. Примером необходимого события является выдача ежедневных системных сводок. Его возникновения ждут, а его отсутствие должно стать поводом для тревоги. Аномальные события обычно классифицируют по шкале «низкий, средний, высокий» (low-medium-high) или «предупреждающий, серьезный, критический» (warn-severe-critical).

Параметры являются непрерывными метриками или дискретными состояниями, которые могут наблюдаться в системе. Именно в этой области прозрачность жизненно необходима. Приложения, открывающие свое внутреннее состояние, предоставляют более точные параметры, имеющие практическую ценность. На самом базовом уровне мы имеем дело с количественными показателями операционной системы: загрузкой процессора, объемом свободной памяти, скоростью подкачки памяти, пропускной способностью сети на один интерфейс и свободным местом на диске.

В случае сервера приложений в общем случае используется большое количество показателей и состояний.

□ Память

Минимальный размер кучи, максимальный размер кучи, размеры поколений.

□ Сборка мусора

Тип, частота, восстановленная память, размер запроса.

□ Рабочие потоки для каждого пула потоков

Количество потоков, занятые потоки, потоки, занятые более пяти секунд, самый высокий уровень (максимальное число одновременно используемых потоков), самый низкий уровень, количество ситуаций недоступности потока, невыполненные запросы.

□ Пулы соединений с базой данных с учетом каждого пула

Число соединений, действующие соединения, самый высокий уровень, самый низкий уровень, количество ситуаций недоступности соединений, невыполненные запросы.

□ Статистика трафика для каждого канала запроса

Всего обработанных запросов, среднее время ответа, прерванные запросы, запросов в секунду, время последнего запроса, прием или отсутствие приема трафика.

Само приложение тоже должно демонстрировать множество своих показателей.

❑ **Бизнес-транзакции каждого типа**

Количество обработанных транзакций, количество прерванных транзакций, стоимость в долларах, устаревание транзакции, коэффициент обрабатываемости, скорость завершения.

❑ **Пользователи**

Демографические данные или классификация, сегментирование по техническим признакам, процент зарегистрированных пользователей, число пользователей, образцы работы, встречающиеся ошибки.

❑ **Точки интеграции**

Текущее состояние, переходы на режим ручного управления, число обращений к точке интеграции, среднее время ответа от удаленной системы, количество отказов.

❑ **Предохранители**

Текущее состояние, переходы на режим ручного управления, количество неудачных вызовов, время последнего успешного вызова, число переходов из одного состояния в другое.

Как видите, даже приложение среднего размера может обладать сотнями параметров. И для каждого из них существует диапазон нормальных и приемлемых значений. Это могут быть допустимые отклонения от дискретных значений или порог, превышать который нельзя. Параметр считается номинальным, пока его значение находится в границах допустимого диапазона. Часто существует еще один, более жесткий диапазон, подающий сигнал «внимание», предупреждая, что параметр приближается к пороговому значению.

Для непрерывных показателей существует удобное эмпирическое правило определения номинала: «среднее значение за данный период времени плюс-минус два стандартных отклонения». Все упирается в выбор временного периода. Для большинства показателей, зависящих от трафика, временным периодом, демонстрирующим наиболее стабильную корреляцию, является «час недели», например 14 часов во вторник. День месяца большого значения не имеет. В некоторых отраслях, например в туризме, торговле цветами и занятиях спортом, самые релевантные измерения отсчитываются от даты праздника или мероприятия.

В розничной торговле паттерн «день недели» сильно перекрывается с циклом «неделя года». Словом, не существует периода времени, подходящего сразу всем отраслям¹.

¹ В качестве яркой иллюстрации можете почитать литературу, посвященную «автоматическому определению характеристик рабочей нагрузки». При всей сухости и академичности формулировок это битва, которая продолжалась долгие годы.

Пресловутая панель инструментов

На панели инструментов состояние зачастую представлено в виде всем знакомой «красно-желто-зеленой» схемы. Классификация состояний сложной, многоуровневой, распределенной системы при помощи всего трех цветов может показаться чрезмерным упрощением, но эта схема хороша тем, что знакома всем и каждому. Соглашения по поводу *значения* каждого из цветов давно выработаны. В таблице 1 я привожу определения, которыми пользовался в прошлом.

Таблица 1. Соответствие цветовых обозначений

Зеленый	<p>Должны соблюдаться следующие условия:</p> <ul style="list-style-type: none"> ■ произошли все ожидаемые события; ■ не было аномальных событий; ■ у всех параметров номинальные значения; ■ все состояния полностью работоспособны
Желтый	<p>Верно хотя бы одно из утверждений:</p> <ul style="list-style-type: none"> ■ не произошло ожидаемое событие; ■ возникло хотя бы одно аномальное событие средней степени опасности; ■ один или несколько параметров имеют значения выше или ниже номинала; ■ некритическое состояние не полностью работоспособно (например, предохранитель запрещает доступ к некритичной функциональной возможности)
Красный	<p>Верно хотя бы одно из утверждений:</p> <ul style="list-style-type: none"> ■ не произошло ожидаемое событие; ■ возникло хотя бы одно аномальное событие высокой степени опасности; ■ один или несколько параметров сильно отклоняются от номинального значения; ■ критическое состояние имеет неожиданное значение (например, параметр, отвечающий за принятие запросов, имеет значение false, а не true)

Текущее состояние системы можно представить в виде панели инструментов. (Оно практически *образует* панель инструментов.) Эта панель должна иметь хороший обзор; можно даже спроецировать ее на стену в столовой. Чем больше людей, понимающих, каким должно быть нормальное функционирование системы, будет ее

видеть, тем лучше. Лучше всего, если панель сможет показывать разным пользователям разные аспекты системы. Инженера из группы поддержки, скорее всего, в первую очередь интересует представление на уровне компонентов. Разработчику будет интереснее представление, ориентированное на приложение, в то время как инвестор больше заинтересован в просмотре бизнес-аспектов функционирования системы. Очевидно, что панель инструментов должна знать, как связаны между собой все эти представления. Фиксируя сбой на уровне компонентов — например, по причине сетевой ошибки, — администратор должен быть в состоянии увидеть, какие бизнес-аспекты затрагивает данная ситуация. Это облегчает общение с инвесторами и помогает правильно определить приоритет проблемы.

У большинства систем существует суточный ритм ожидаемых событий. Это могут быть потоки данных от других систем, извлечение фрагментов данных для отправки другим системам или просто фоновые задания по интеграции с предыдущими версиями систем. Вне зависимости от их назначения все эти задания становятся такой же частью системы, как веб-серверы или серверы баз данных. Их выполнение попадает в категорию «необходимых ожидаемых событий». Панель инструментов должна отображать их вне зависимости от того, возникли они или нет и удалось ли их завершить. Порой выясняется, что причиной бизнес-проблемы является сбой в пакетном задании, который незаметно для администратора возникал много дней подряд.

Текущее состояние важно для любого человека, финансовые или карьерные ожидания которого связаны с повседневным функционированием систем. Специалисты в области ИТ, однозначно попадают в эту категорию. Существуют и инвесторы из областей, напрямую не касающихся ИТ карьера которых связана с компьютерными системами. Всем этим людям важно максимально хорошо знать ситуацию, чего можно добиться, наблюдая за текущим состоянием системы в разное время и при различных условиях.

Текущее поведение

Текущее поведение позволяет ответить на вопрос: «Что за *****?». Люди обычно начинают интересоваться этим аспектом, когда инцидент уже в самом разгаре.

Очевидно, что текущее поведение связано с текущим состоянием. Аномалии такого поведения часто, хотя и не всегда, являются результатом некорректного состояния. Трасса стека, дампы потоков, ошибки, занесенные в файл системного журнала, и некорректные ответы пользователям — все это примеры поведенческих проблем, которые могут повлиять, а могут не повлиять на состояние системы. К примеру, пользователи не любят видеть сообщения об ошибке. Увидев такое сообщение, они, как правило, уходят. И это в конечном счете приводит к тому, что показатель «количество транзакций в час» опускается до значений, выходящих за границу номинального диапазона. Лучше выявлять некорректное поведение до того, как пользователи начнут уходить.

Текущее поведение — это область применения систем мониторинга. Самые разные системы, от доморощенных анализаторов журналов до стоящих миллионы долларов продуктов семейства OpenView, подобно Большому брату надзирают за тем, чтобы все шло по плану. Еще это область применения дампов потоков (см. примечание в разделе 2.3). Такие среды разработки, как JMX, также позволяют наблюдать за текущим поведением, предоставляя администраторам доступ к внутренним компонентам запущенных приложений.

Кто может увидеть текущее поведение? Некоторые вещи следует ограничить из-за потенциальной возможности нанести вред. Далеко не у каждого человека будет доступ к консоли JMX, которая позволяет останавливать работу серверов и менять жизненно важные параметры. Некоторые методы получения данных о состоянии системы (например, дампы потоков) требуют привилегированного доступа к серверам. Но даже если не брать эти откровенно опасные моменты, зачастую отдел, отвечающий за управление системой, ощущает потенциальную угрозу, когда разработчики или инвесторы просят продемонстрировать им поведение системы, особенно в ситуации отсутствия доверия или даже накалившейся атмосферы. Администраторы беспокоятся, что их могут завалить вопросами по поводу каждого всплеска или аномалии. Хуже того, заказчики — люди, которые платят за систему, — могут первыми заметить проблему. Нельзя вообразить более неловкую ситуацию, чем звонок от руководителя службы маркетинга с вопросом: «Почему загрузка процессора у одного из серверов выше, чем у всех остальных?» — на такой вопрос можно ответить только беспомощным: «Я не знаю. Я еще не смотрел сегодняшние графики».

Одним из способов разрядки этой напряженности является увеличение прозрачности текущего состояния. Просьба предоставить дополнительную информацию со стороны инвестора, как правило, означает, что его интересует не текущее поведение системы, а ее текущее состояние. (Именно поэтому я для начала разделяю эти две сферы.) Их вполне можно понять. Они отвечают за финансовый успех системы, а значит, должны представлять, что с этой системой происходит. Но им вовсе не нужно знать о каждом случае, когда приложение оставляет результат трассировки стека в журнале. На самом деле их интересует «Соответствует ли наш доход запланированному?», «Увеличила ли последняя рекламная кампания коэффициент обращаемости?» или «Не следует ли закупить дополнительные киоски и нанять дополнительных агентов?». Для ответов на эти вопросы потребуется панель инструментов, демонстрирующая состояние, исторические тренды и прогнозы на будущее. Показывать SNMP-ловушки и дампы потоков излишне.

17.2. Проектирование с учетом прозрачности

Прозрачность совершенно осознанно вводится на этапе проектирования и выбора архитектуры. «Ввод прозрачности» на поздних этапах разработки так же эффективен, как «повышение качества». Даже если это и можно сделать, то исключительно

ценой куда больших усилий и вложений, чем потребовалось бы при изначальном построении системы с учетом этого аспекта.

Осведомленности о том, что происходит внутри одного приложения или сервера, недостаточно. Исключительно локальная осведомленность ведет к столь же локальной оптимизации. К примеру, продавец инициировал большой проект, способствующий более быстрому появлению на сайте новых элементов. Ночные обновления шли до 5 или 6 часов утра, в то время как их нужно было завершить ближе к полуночи. Проект оптимизировал строку пакетных заданий, отвечающих за подачу материала на сайт. Поставленная цель была достигнута, так как пакетные задания стали завершаться на два часа раньше. Тем не менее новые товары на сайте не появлялись, пока к 5 или 6 часам утра не завершился длительный параллельный процесс. Локальная оптимизация пакетного задания в глобальном смысле не оказала никакого эффекта.

Возможность наблюдать только за одним приложением порой мешает распознавать эффекты масштабирования. К примеру, наблюдая за очисткой кэша только на одном из серверов приложений, вы не увидите, что каждый сервер удаляет данные из кэшей всех прочих серверов. При каждой визуализации элемента он случайным образом обновляется, что приводит к рассылке всем остальным серверам уведомления о недействительности кэша. Проблема стала очевидной, когда появилась возможность увидеть статистику всех серверов на одной странице. Без этого мы могли бы добавлять все новые и новые серверы, пытаясь обеспечить необходимую вычислительную мощность, но каждый следующий сервер только усугублял бы проблему.

Для обеспечения прозрачности при проектировании внимательно следите за взаимозависимостью. Средства мониторинга могут достаточно легко вторгаться во внутреннюю структуру системы. Далее мы будем обсуждать стандарты, позволяющие избежать слишком сильной взаимозависимости. Системы мониторинга и формирования отчетов должны быть своего рода внешней надстройкой над вашей системой, а не становиться ее частью. В частности, решения о том, какие показатели должны вызывать рассылку оповещений, какими должны быть пороговые значения, как по переменным состояния определить общее здоровье системы, должны приниматься вне приложения. Это стратегические решения, темп изменения которых отличается от темпа изменения кода приложения.

17.3. Применение технологий

Процесс, выполняющийся на сервере, полностью непрозрачен по своей природе. Если не запустить для него отладчик, никакой информации о себе этот процесс не предоставит. Он может прекрасно работать, может запустить свой последний поток или вообще ходить по кругу, не выполняя никакой работы. Как и в случае с котом Шрёдингера, невозможно сказать, жив процесс или мертв, пока вы его не увидите.

Значит, наша первая задача — получить от процесса информацию. Я расскажу вам о двух наиболее важных современных технологиях, увеличивающих прозрачность границ процесса. Их можно классифицировать как «белый ящик» и «черный ящик».

Технология, работающая по принципу черного ящика, находится вне процесса, исследуя его внешние проявления. Такая технология допускает внедрение в работающую систему силами системного администратора. Хотя наблюдаемая система не знает о существовании данной технологии, есть ряд полезных приемов, применяемых на стадии разработки, которые облегчают ее использование.

Технологии, построенные по принципу белого ящика, напротив, работают внутри объекта, за которым ведется наблюдение. Этим объектом может быть как один процесс, так и система в целом. Система сознательно проявляет себя через подобные инструменты. Их внедрение осуществляется на этапе разработки. Технологии белого ящика всегда связаны с системой намного теснее технологий черного ящика.

17.4. Протоколирование

Несмотря на миллионы долларов, которые тратятся на исследования и разработку пакетов, отвечающих за «управление корпоративными приложениями», и великолепные рабочие центры с гигантскими плазменными мониторами, демонстрирующими карты сети с цветовым кодированием, старые добрые файлы журналов являются наиболее надежными и универсальными носителями информации. Иногда можно даже посмеяться над тем, что мы живем в 21 веке, а файлы журналов все еще остаются одним из наиболее ценных инструментов.

Протоколирование однозначно относится к технологиям, работающим по принципу белого ящика; его приходится глубоко внедрять в исходный код программы. Тем не менее протоколирование по ряду веских причин используется повсеместно. Файлы журналов воспроизводят всю деятельность, происходящую внутри приложения. Таким образом, именно они показывают текущее поведение приложения. Еще они хранятся долгое время, а значит, их анализ дает ключ к состоянию системы — хотя зачастую для отслеживания перехода в текущее состояние требуется определенная «обработка».

Если вы хотите избежать сильной взаимозависимости с конкретным инструментом мониторинга или конкретной средой разработки (см. подраздел «Коммерческие системы мониторинга» в разделе 17.5), значит, нужно обратить внимание на файлы журналов. Невозможно придумать менее слабосвязанный компонент; анализировать файлы журналов способна любая существующая среда разработки или инструмент. Подобная слабая взаимозависимость означает, что файлы журналов пригодятся вам и в процессе разработки, когда под рукой вряд ли будут программные продукты семейства OpenView.

Несмотря на все это, файлы журналов зачастую используются некорректно. Далее вы найдете ряд ключей к успешному протоколированию.

Конфигурация

Вопреки инстинктивному желанию разработчиков поместить каталог файлов журналов внутрь каталога установки приложения, администраторы зачастую хотят хранить журналы не на том диске или логической единице, где находится операционная система или контент. Файлы журналов могут быть большими. Они быстро растут и сильно нагружают ввод-вывод. Поместив их на отдельный диск, вы распараллелите процессы ввода-вывода и одновременно снизите конкуренцию за занятые жесткие диски.

Если вы предоставите возможность регулировать местоположение файлов журналов, администратор сможет присвоить данному свойству нужное ему значение. В противном случае файлы журналов все равно будут перемещены в другое место, но способ этого перемещения может вам не понравиться.

В операционных системах семейства UNIX наиболее распространенным обходным путем являются символьные ссылки. Такая ссылка ведет из папки **logs** к актуальному местоположению файлов. Небольшой платой за это становятся операции ввода-вывода при каждом открытии файла, но конкуренция за занятые жесткие диски обходится куда дороже. Еще мне доводилось видеть выделенные под файлы журналов файловые системы, которые монтировались сразу под установочным каталогом.

К счастью, все известные платформы протоколирования поддерживают настраиваемые маршруты (что является еще одной причиной *отказа* от собственных систем протоколирования).

Уровни протоколирования

Читая (или даже просто сканируя) файлы журналов новой системы, люди учатся понимать, что для этой системы является «нормальным». Некоторые приложения, особенно новые приложения для электронной коммерции, крайне «зашумлены»; они генерируют множество ошибок в своих файлах журналов. Бывают спокойные приложения, вообще не посылающие отчетов в процессе нормального функционирования. В любом случае, приложение дает понять человеку, как выглядит нормальный режим.

Большинство разработчиков реализуют протоколирование так, как будто они сами являются основными потребителями файлов журналов. При этом администраторы и инженеры из группы поддержки проводят за анализом этих журналов куда больше времени, чем любой разработчик. Протоколирование должно делаться

с ориентировкой на эксплуатацию системы, а не на разработку или тестирование. А значит, любые события, зарегистрированные с уровнем «ERROR» или «SEVERE», требуют действий со стороны обслуживающего персонала. Далеко не каждое исключение нужно фиксировать как ошибку. Тот факт, что пользователь ошибся при вводе номера кредитной карты и в результате компонент, отвечающий за ее проверку, выбросил исключение, не означает, что следует предпринимать какие-то действия. Ошибки в бизнес-логике или при пользовательском вводе имеет смысл фиксировать как предупреждения (если их вообще нужно фиксировать). Оставьте уровень «ERROR» для серьезных системных проблем. Например, предохранитель, застрявший в положении «открыт», — это ошибка. В нормальных обстоятельствах подобного не могло бы произойти, и скорее всего, с другой стороны соединения должны быть произведены некие действия. Сбой при подключении к базе данных — это ошибка, причиной которой является проблема в сети или на сервере базы данных. Исключение `NullPointerException` автоматически ошибкой не становится.

ОТЛАДКА ЖУРНАЛОВ В ПРОЦЕССЕ ЭКСПЛУАТАЦИИ

Раз уж мы говорим об уровнях протоколирования, упомяну про свою большую моль: «отладку» журналов в процессе эксплуатации. Этого практически никогда делать не стоит, так как возникает такое количество помех, что реальные проблемы тонут под многочисленными результатами трассировки методов и обычными контрольными точками. При этом оставить отладочные сообщения включенными очень легко. Достаточно в процессе разработки отправить фрагмент кода в `svn` или `svn` при включенном уровне отладки в конфигурации системы протоколирования. Я рекомендую добавить к процессу сборки шаг, который будет автоматически удалять все настройки, позволяющие включать отладку или трассировку уровней протоколирования.

Каталог сообщений

Распространенным запросом от отдела эксплуатации является получение списка всех сообщений в файле журнала, которые может сгенерировать система. Такие запросы всегда вызывают у меня стон. Впрочем, сейчас для экспорта всего содержимого журналов достаточно нескольких часов работы с утилитами интернационализации в таких средах разработки, как Eclipse, IDEA и NetBeans. Как только все собирается в единый пакет ресурсов, пересылка его в службу поддержки трудностей не вызывает.

Кроме того, к каждому сообщению можно добавить однозначно трактуемые коды. Коды сообщений обладают двумя замечательными свойствами. Во-первых, они обеспечивают точность взаимодействия между отделом эксплуатации и разработчиками. Если вам когда-нибудь говорили: «Я не уверен, что означает сообщение, но там упоминается о неустранимой системной ошибке», значит, имеется необходимость

в кодах сообщений. Во-вторых, значения этих кодов легко найти в технической документации или в базе знаний. Представьте, что в вашем коде много раз появляется следующий паттерн:

```
try {
    ...
} catch (TimeoutException e) {
    LOGGER.log(Level.SEVERE, "Timeout failure connecting
        to fulfillment system.", e);
}
```

Применение в среде разработки Eclipse команды **Externalize Strings** дает нам новый файл `messages.properties` и новый класс `Messages`. Вот что Eclipse помещает в файл `messages.properties`:

```
transparency/messages.properties
FulfillmentClient.0=Connection refused.
FulfillmentClient.1=Authorization failed. Credentials refused.
FulfillmentClient.2=Timeout failure connecting to fulfillment system.
```

Доступ к этим сообщениям осуществляется через метод `getString(String key)`, примененный к новому служебному классу `Messages`, который среда разработки Eclipse создала в том же пакете, что и исходный класс:

```
transparency/Messages.java
public static String getString(String key) {
    // Вставить автоматически генерируемую заглушку метода
    try {
        return RESOURCE_BUNDLE.getString(key);
    } catch (MissingResourceException e) {
        return '!' + key + '!';
    }
}
```

Хотя команда **Externalize Strings** была добавлена для упрощения интернационализации, ее можно использовать и для другого типа интерфейса. Как только среда разработки Eclipse соберет все строки в один файл, а служебный метод их вернет, у вас появится замечательная отправная точка для внесения изменений. Например, после небольшого редактирования класса `Messages` параметр `key` в методе `getString(String key)` создает прекрасный код сообщений:

```
transparency/Messages.java
public static String getString(String key) {
    // Вставить автоматически генерируемую заглушку метода
    try {
        return "(" + key + ") " + RESOURCE_BUNDLE.getString(key);
    } catch (MissingResourceException e) {
        return '!' + key + '!';
    }
}
```

Человеческий фактор

Самое главное, что файлы журналов могут читаться людьми. То есть они формируют человеко-машинный интерфейс и должны рассматриваться с учетом такого фактора, как присутствие человека. Это может прозвучать тривиально или даже смешно, но в стрессовой ситуации, например в случае инцидента уровня Severity 1, неверная интерпретация информации о состоянии системы может продлить или усугубить проблему. Операторы АЭС Три-Майл-Айленд неверно интерпретировали показания датчиков давления охлаждающей жидкости и температуры, что спровоцировало целую череду ошибочных действий (см. книгу Джеймса Р. Чайлза *Inviting Disaster*, с. 49–63). Конечно, отказ систем, с которыми мы работаем, не станет причиной радиоактивных выбросов в окружающую среду, но обойтись вам в тысячи долларов вполне может. Поэтому важно гарантировать, чтобы файлы журналов предоставляли четкую, точную и конкретную информацию тем, кто будет их читать.

Если файлы журналов представляют собой интерфейс для взаимодействия «человек–машина», они должны быть написаны таким образом, чтобы человек мог максимально быстро распознать и интерпретировать их содержимое. Нужен максимально читабельный формат. Зрительная система человека — это беспрецедентно быстрый и сложный механизм распознавания шаблонов. Почему же многочисленные производители делают все возможное, чтобы помешать этой способности? Рисунок 35 демонстрирует пример формата журнала, прочитать который сможет компьютер или, возможно, марсианин, но уж никак не человек. Это фрагмент загрузки сервера WebLogic 9.2. Вы можете быстро найти в нем предупреждающее сообщение?

```
<Aug 13, 2006 7:24:53 PM CDT> <Notice> <Log Management> <BEA-170027> <The server
<Aug 13, 2006 7:24:54 PM CDT> <Notice> <WebLogicServer> <BEA-000365> <Server stat
<Aug 13, 2006 7:24:54 PM CDT> <Notice> <WebLogicServer> <BEA-000365> <Server stat
<Aug 13, 2006 7:24:57 PM CDT> <Notice> <Security> <BEA-090171> <Loading the ident
A-000331> <Started WebLogic Admin Server "examplesServer" for domain "wl_server"
<Aug 13, 2006 7:25:00 PM CDT> <Warning> <WorkManager> <BEA-002919> <Unable to fir
map to the default WorkManager for the application bea_wls9_async_response>
<Aug 13, 2006 7:25:00 PM CDT> <Warning> <EJB> <BEA-014014> <The message driven be
chPolicy" that refers to an unknown work manager. The default work manager will b
Could not invoke browser, command=netscape -remote openURL(http://192.168.1.98:70
can open from the cmd-line.
java.io.IOException: java.io.IOException: netscape: not found
<Aug 13, 2006 7:25:00 PM CDT> <Notice> <WebLogicServer> <BEA-000365> <Server stat
<Aug 13, 2006 7:25:00 PM CDT> <Notice> <WebLogicServer> <BEA-000360> <Server star
```

Рис. 35. Этот формат понять невозможно

На рис. 36 показан журнал сервера приложения WebSphere 6.1. В нем используется тот же самый прикладной программный интерфейс `java.util.logging`, что и в предыдущем примере, но ужасный формат, предлагаемый по умолчанию, заменен более приемлемым. Этот формат уже доступен для сканирования

человеческим глазом. Если знать, что буквы *I*, *W* и *A* обозначают разные уровни серьезности («information», «warning» и «audit»), поиск предупреждений и ошибок становится тривиальной задачей. Разделение на столбцы помогает человеку прочитать данные. Обратите внимание на поле с кодами сообщений, помогающее автоматическому анализу файла. Такой формат журнала понятен как человеку, так и компьютеру.

```
[8/14/06 8:22:14:653 CDT] 0000000a SSLComponentI I CWPKI0001I: SSL service is
[8/14/06 8:22:14:813 CDT] 0000000a WSKeystore W CWPKI0041W: One or more key
[8/14/06 8:22:14:848 CDT] 0000000a SSLConfigMana I CWPKI0027I: Disabling defau
[8/14/06 8:22:24:639 CDT] 0000000a WorkSpaceMana A WKSP0500I: Workspace config
[8/14/06 8:22:25:508 CDT] 0000000a FileRepositor A ADMR0010I: Document cells/t:
[8/14/06 8:22:25:961 CDT] 0000000a SSLDiagnostic I CWPKI0014I: The SSL compone
[8/14/06 8:22:26:325 CDT] 0000000a FileRepositor A ADMR0010I: Document cells/t:
[8/14/06 8:22:26:670 CDT] 0000000a SSLComponentI I CWPKI0002I: SSL service ini
```

Рис. 36. Понятный формат

Рисунок 37 демонстрирует уже знакомый нам по первой иллюстрации вывод загрузочной последовательности сервера WebSphere, но на этот раз этот вывод записан в формате, который по умолчанию предлагает JDK-пакет `java.util.logging`.

```
Aug 19, 2006 7:13:25 PM com.example.server.SSLComponentIdentity emit
INFO: SSL service is initializing the configuration
Aug 19, 2006 7:13:25 PM com.example.server.WSKeystore emit
WARNING: One or more key stores are using the default password.
Aug 19, 2006 7:13:25 PM com.example.server.SSLConfigManager emit
INFO: Disabling default hostname verification for HTTPS URL connections.
Aug 19, 2006 7:13:25 PM com.example.server.WorkSpaceManager emit
INFO: Workspace configuration consistency check is false.
Aug 19, 2006 7:13:25 PM com.example.server.FileRepository emit
INFO: Document cells/trozNode01Cell/security.xml is modified.
Aug 19, 2006 7:13:25 PM com.example.server.SSLDiagnostic emit
INFO: The SSL component's FFDC Diagnostic Module com.ibm.ws.ssl.core.SSLDiagnosti
Aug 19, 2006 7:13:25 PM com.example.server.FileRepository emit
INFO: Document cells/trozNode01Cell/security.xml is modified.
Aug 19, 2006 7:13:25 PM com.example.server.SSLComponentIdentity emit
INFO: SSL service initialization completed successfully
```

Рис. 37. Формат, предлагаемый по умолчанию в `java.util.logging`

Я не знаю, кто придумал этот формат из двух строк, но он делает визуальное восприятие журнала совершенно невозможным. Более того, анализ журнала другими программами в этом случае тоже затруднен. Утилита `grep` с таким форматом работать не умеет. Вам придется найти какого-нибудь UNIX-хакера старой школы, который сумеет применить один из `awk`-трюков. Перед этим форматом бессильны и человек, и компьютер.

ШАМАНСКИЕ ОПЕРАЦИИ

Как я уже говорил, люди очень хорошо выявляют шаблоны. По сути, у нас есть естественная склонность к поиску шаблонов даже там, где они отсутствуют. В своей книге «Why People Believe Weird Things» Майкл Шермер обсуждает влияние эволюции на распознавание шаблонов. Те древние люди, которые не могли распознать реальный шаблон, например сочетание света и тени, в реальности оказывавшееся леопардом, с меньшей вероятностью передавали свои гены, чем те, кто распознавал даже несуществующие шаблоны и бежал прочь от кустов, которые выглядели как леопард.

Другими словами, цена ложноположительного срабатывания — «распознавания» отсутствующего шаблона — была минимальной, в то время как за ложноотрицательное срабатывание — неумение распознать реально существующий шаблон — приходилось дорого платить. Шермер утверждает, что именно давление эволюционного отбора способствует склонности к суевериям. Мне приходилось видеть это на практике. Когда система находится на грани отказа, администратору приходится очень быстро проходить этапы наблюдения, анализа, выдвижения гипотез и действия. Если действия приводят к положительному результату, они становятся частью статистики, а возможно, даже частью документированной базы знаний. Но откуда известно, что эти действия были корректными? А что, если это просто совпадение?

Однажды я столкнулся с ситуацией, когда действия группы, занимающейся обслуживанием одного из моих ранних коммерческих приложений, были ничем не лучше вульгарного шаманства. Я стоял рядом с администратором, когда на его пейджер пришло сообщение. Он немедленно авторизовался на рабочем сервере базы данных и начал его аварийное переключение. Из любопытства я спросил, что происходит. Мне рассказали, что пришедшее сообщение информирует о скором отказе сервера базы данных, поэтому нужно как можно быстрее переключиться на резервный узел и перезагрузить основную базу данных. Я посмотрел на пейджер, и по спине у меня побежали мурашки. Сообщение гласило: «Достигнут предел времени жизни канала данных. Требуется сброс».

Разумеется, я узнал это сообщение, ведь я сам его написал. И оно не имело никакого отношения к базе данных. Это было отладочное сообщение (см. предыдущую врезку «Отладка журналов в процессе эксплуатации»), информирующее меня о том, что зашифрованный канал связи с внешним поставщиком работает уже достаточно долго для того, чтобы ключ шифрования скоро стал уязвимым просто из-за объема обслуживаемых этим каналом зашифрованных данных. Такое случалось примерно раз в неделю. Частично проблема заключалась в том, что в словосочетании «требуется сброс» ничего не говорилось о том, кто должен его выполнить. Если посмотреть код, становилось понятным, что приложение само осуществит сброс канала сразу же после отправки сообщения, — но у адресатов сообщения не было возможности увидеть код. В свое время я оставил включенным режим вывода этих отладочных сообщений, чтобы понять, насколько часто возникает такая ситуация при нормальных объемах данных. А потом просто забыл его отключить.

Мне удалось выявить момент зарождения этого мифа о необходимости аварийного переключения базы данных. Примерно шесть месяцев назад сразу после обеда случился системный отказ. И сообщение «Требуется сброс» оказалось последним, зафиксированным перед падением сервера Sybase. Причинно-следственная связь между этими вещами отсутствовала, но они произошли одновременно. Никакого предупреждающего сообщения об отказе базы данных не было — ей требовалось исправление от Sybase, которое мы применили почти сразу же после отключения. Но совпадение двух событий во времени в сочетании с нечетко сформулированным сообщением заставило администраторов в течение шести месяцев в часы пиковых нагрузок раз в неделю выполнять переключение базы данных.

Заключительные замечания

Сообщения должны содержать идентификатор, позволяющий проследить этапы транзакции. Это может быть идентификатор пользователя, идентификатор сеанса, идентификатор транзакции или даже случайное число, присвоенное в момент прихода запроса. Когда наступает время погрузиться в чтение журнала из 10 000 строк (например, после отказа), возможность указать утилите **grep** номер строки изрядно экономит вам время.

Следует фиксировать интересные переходы между состояниями, даже если вы планируете использовать SNMP-ловушки или JMX-уведомления для информирования системы мониторинга. Написание дополнительного кода для фиксации смены состояния занимает всего несколько секунд, но выводимые этим кодом данные существенно упрощают эксплуатацию системы. Кроме того, эта запись крайне важна при анализах причин сбоя.

17.5. Системы мониторинга

Даже самая лучшая система протоколирования может помочь только при работающем приложении. Мертвые процессы следов не оставляют. Как и зависшие процессы. Поэтому требуется нечто, наблюдающее за процессом извне, — какой-то инструмент, работающий по принципу черного ящика и следящий за здоровьем и благополучием приложения и компьютера, на котором оно запущено.

Здесь мы вступаем в область систем мониторинга. Рисунок 38 демонстрирует изрядно упрощенное представление такой системы. Важными компонентами подобного «наблюдателя со стороны» являются агенты, собирающие информацию, а также надежный механизм передачи, отображения и обработки информации.

У большинства систем мониторинга агенты запускаются на тех хостах, за которыми ведется наблюдение. Эти агенты выполняют периодическую проверку, в том числе проверку статистики производительности операционной системы, состояния процессов, совпадения файлов журналов с шаблонами и слушателей портов. В случае хостов UNIX они сканируют журнал **syslog** и любые другие файлы журналов, настроенные администратором. На хостах Windows сканируется также журнал событий Windows. В показанном примере агент на сервере базы данных замечает, что база прекратила свою работу. Он создает событие для брокера сообщений — встроенного компонента, функционирующего как часть системы мониторинга. Брокер рассылает предупреждение о возможной ошибке всем активным клиентам. В данном случае консоль мониторинга у администратора — клиентское приложение, обращающееся к системе мониторинга, — отображает предупреждение в виде текстового сообщения и меняет цвет значка базы данных на красный.

Эти агенты отлично распознают события, на поиск которых их запрограммировали: шаблоны в файлах журналов, SNMP-ловушки, процессы, которые прекратили свою работу или потребляют больше ресурсов, чем им было выделено, файловые системы, близкие к переполнению, и т. п. Неожиданное или новое поведение, не приводящее к одному из этих событий, может остаться нераспознанным. К примеру, при едином формате фиксации ошибок в журнале может быть обнаружена неожиданная ошибка. Если же ошибки какого-то типа не фиксируются или фиксируются в другом формате, система мониторинга не в состоянии их выявить.

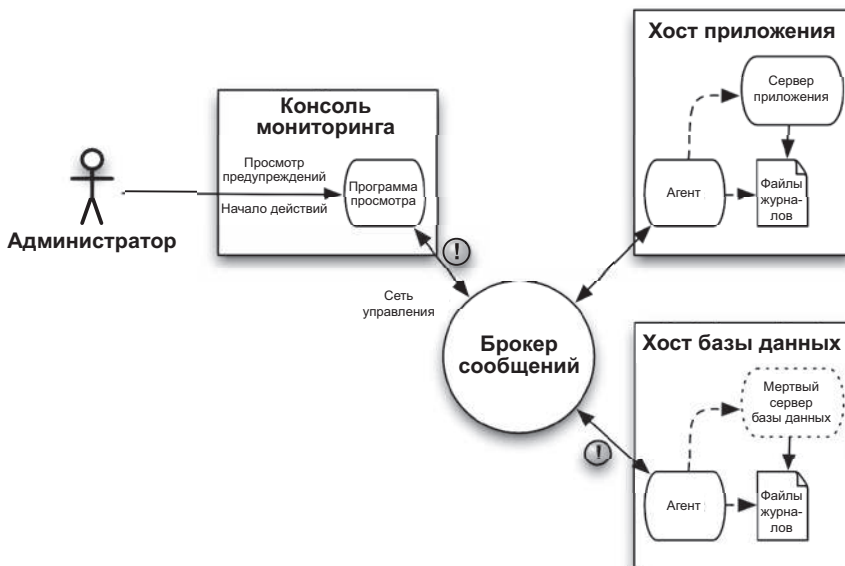


Рис. 38. Схема системы мониторинга

Агенты подобны внедренным в военную часть репортерам. Они находятся рядом с местом действия, то есть в зоне риска. Если хост прекращает работу, агент также выходит из строя. Поэтому системы мониторинга всегда пользуются контрольными сигналами для распознавания неработающего агента или отказа ведущей к агенту сети.

Внедренные журналисты должны иметь возможность отправлять свои репортажи в газету; аналогично обстоят дела с агентами мониторинга. Системы мониторинга крайне зависимы от бесперебойной доставки сообщений. Во внедренных корпоративных системах малого и среднего размера трафик мониторинга иногда проходит те же самые сегменты сети, что и рабочий трафик. Это не очень хорошая ситуация. Ведь в подобных случаях проблемы с рабочей сетью — например, сетевой червь, DDoS-атака или старая добрая ошибка в конфигурации — автоматически

отключают систему мониторинга. Связанный с этой системой трафик часто содержит конфиденциальную информацию. Это не обязательно могут быть пароли. Зачастую там передаются имена хостов, имена пользователей, внутренние IP-адреса, фрагменты файлов журналов, имена и идентификаторы процессов и пр. Ничто из этого не должно оказаться в общем доступе, поэтому данный трафик не должен пересекать границ виртуальной локальной сети, по которой идет общий трафик.

МОНИТОРИНГ БЕЗ АГЕНТОВ

Некоторые системы мониторинга предлагают варианты установки без агентов. Их разработчики утверждают, что, избавившись от агента на хосте, за которым ведется наблюдение, мы сокращаем непроизводительные издержки процессора. По большому счету, это маркетинговая уловка. Даже если данные мониторинга собираются с другого хоста, нам все равно не обойтись без взаимодействия с операционной системой наблюдаемой машины. К примеру, программа Operations Manager от Microsoft может работать без агента. В этой конфигурации сервер управления использует встроенный журнал системных событий и средства удаленного администрирования наблюдаемого хоста. То есть сбор информации по-прежнему происходит и потребление ресурсов этого хоста продолжается.

Коммерческие системы мониторинга

Ранее в больших коммерческих системах мониторинга были заинтересованы только крупные предприятия. Такие производители, как IBM, Computer Associates и Hewlett-Packard, после многомиллионных сделок вошли в список Fortune 500. После того как деловой мир стал работать по расписанию 24/7, системы стали больше. В это же время производители обнаружили возможность новых сегментов рынка. Ураган Катрина в 2005 году и события 11 сентября 2001 года продемонстрировали острую необходимость в надежном аварийном восстановлении и бесперебойной работе. (Многие руководители ИТ-отделов пробовали поднять тревогу перед приходом урагана Катрина, но планирование аварийного восстановления никогда особо не привлекало руководителей и директоров, пока ряду компаний не пришлось прекратить свою работу просто из-за отсутствия плана действий в критических ситуациях.) Строгая нормативно-правовая база, возникшая после тотального внедрения раздела 404¹ закона Сарбейнза — Оксли, также повысила важность контроля безопасности и мониторинга. В результате этого взаимодействия разных факторов коммерческие системы мониторинга стали привычной частью любого бизнеса.

¹ Меня до ужаса забавляет, что раздел 404 требует контроля конфиденциальной информации и возможности проверять источник каждого фрагмента финансовых данных, в то время как «ошибка 404» в Интернете означает «не найдено». Впрочем, на большинстве вечеринок, в которых я принимал участие, эта шутка не производила нужного эффекта.

УПРАВЛЕНИЕ КОРПОРАТИВНЫМИ ПРИЛОЖЕНИЯМИ

По большому счету, рассматриваемые системы ориентированы на управление сетями и серверами. В конце 1990-х годов возникло изрядное количество мелких компаний, нацеленных на работу с приложениями. В отличие от своих предшественников они считали доступность и производительность приложений определяющими факторами для удобства пользователя. Однако большие рыбы всегда едят мелких, и эти мелкие фирмы с рынка исчезли. Основные производители: HP, EMC, BMC, IBM, CA (в этой области невозможно выжить фирмам, масштаб которых недостаточно велик, чтобы их узнавали всего по двум или трем буквам) — сейчас объединяют эти продукты, нацеленные на работу с приложениями, со своими основными продуктами, чтобы создать перспективные комплексы управления корпоративными приложениями.

В этих комплексах сочетается осведомленность о состоянии сети, серверов и приложений, чтобы (в теории) обеспечить полную видимость систем. Они поддерживают автоматическое обнаружение зависимостей путем прослушивания сетевых пакетов и получения базовых характеристик производительности. Это перспективные комплексы, которые могут дать нам новые, замечательные вещи, особенно если начнут поддерживать функциональные возможности, характерные для жизненного цикла приложений, такие как управление конфигурацией, автоматизированное развертывание и нагрузочное тестирование. Это быстро развивающийся сегмент рынка.

Недостатки коммерческих систем

Наборы технических средств от крупных производителей стабильны, развиты и хорошо поддерживаются. Если использовать выражения из книги Джеффри Мура *Crossing the Chasm*, эти системы уже вошли в число «новейших из широко распространенных» — один из самых жестких рынков для продажи технологий. Если у них и есть недостаток, то это ориентация на ИТ (хотя вряд ли: см. врезку «Управление корпоративными приложениями» в предыдущем подразделе). В настоящее время уже неприемлемо — да и в принципе это была не самая лучшая идея — отделять функционирование производственных систем от их бизнес-логики.

Объединение функционирования с коммерческими результатами требует умения сопоставлять «системную» и «деловую» информацию. К примеру, можете ли вы в типичной современной распределенной системе однозначно идентифицировать, какой из серверов предоставляет конкретную функцию? Мне кажется, что нет. Скорее всего, одна функция поддерживается группой серверов на разных уровнях. Соответственно, система мониторинга должна быть осведомлена не только о системах, но и о предоставляемых ими бизнес-сервисах. По сути, требуется, чтобы она была в состоянии определить влияние на эти сервисы любого системного события — как проблемы, так и просто отклонения показателей от нормального значения. До сих пор большинство коммерческих систем, вероятно, считало, что приложение не обеспечивает собственную видимость. В эти системы начали внедрять поддержку данного типа осведомленности, но работы в этом направлении предстоит еще много.

Существует еще один серьезный недостаток рассматриваемых систем. Дело в том, что они могут сообщить только свою точку зрения на происходящее. Поэтому возможна ситуация, когда все компоненты окружения сами по себе функционируют, но в целом конечный пользователь видит совершенно неудовлетворительный результат. Подобное часто происходит в случае заблокированных потоков или каскадных отказов. Даже при корректном обслуживании большинства пользователей некоторые из них могут испытывать проблемы из-за своих профилей, браузеров, файлов cookie или других внешних факторов. Основные системы до сих пор не представляют взгляд на систему с позиции пользователя, они показывают, как система видит сама себя. Однако появился ряд новых продуктов (пока не съеденных более крупными фирмами), которые ставят себя на позицию пользователя.

Проектирование для систем мониторинга

Выбор системы мониторинга практически всегда делается за вас. Это дорогие системы, требующие повсеместного внедрения для принесения прибыли. Один раз выбранная и внедренная система будет много лет внедряться во все ИТ-функциональное. Это решение переживет корпоративную ориентированность на операционные системы, языки программирования, производителей аппаратного обеспечения и организационные диаграммы. Другими словами, вероятность того, что система мониторинга войдет «в игру» именно в тот момент, когда вы проектируете новую программу, крайне мала. А значит, она становится частью среды, для которой вы проектируете.

В такую среду крайне легко добавить поддержку конкретной системы мониторинга, получив в результате сильную взаимозависимость и замкнутость на одного производителя. Этой ловушки можно избежать, при проектировании ориентируясь как официально, так и фактически на небольшой набор применимых стандартов.

ИНФОРМИРОВАННОСТЬ О ВНЕШНЕМ ОКРУЖЕНИИ

В статье «In Ambient Displays: Turning Architectural Space into an Interface between People and Digital Information» группа исследователей из медиалаборатории Массачусетского технологического института приводит доводы в пользу дисплеев, отражающих окружающую обстановку. В понятие окружающей обстановки входят негромкие звуки, освещение, шаблоны движения, такие как рябь на воде, — каждый проецируется на какой-то фрагмент данных. Например, детская вертушка на палочке могла бы отражать входящий трафик веб-сайта. А звуковая гамма могла бы представлять внутреннее состояние систем.

В нормальной ситуации восприятие звуков человеком отходит на второй план, как стрекотание сверчков летним вечером. Часть великолепной системы распознавания образов в человеческом мозге подавляет однообразную входящую информацию, по крайней мере в случае, когда она не является угрожающей. И что еще лучше, через день-другой в такой среде человек автоматически начинает распознавать «нормальный» звук, совсем как судовой инженер.

При изменении базовых данных дисплей, проецирующий эти данные, меняет свой звук или вид. Представьте, что все сверчки внезапно замолчали. Их стрекотание может сливаться с фоном, но не должно совершенно исчезать. У человека до какой-то степени остается осознание этих звуков, и внезапно наступившая тишина начинает казаться ему оглушительно громкой.

Попасть на работу в медиалабораторию не просто, но существует проект с открытым исходным кодом, нацеленный на предоставление информации об окружении через звуковую гамму. Приложение Peep¹ следит за состояниями и событиями в сети и превращает их в приятные звуки, такие как журчание ручья и пение птиц. Новая версия этого проекта не выходила уже более четырех лет. Кажется, создатели его совсем забросили и ждут кого-нибудь, способного принять эстафету.

На сайте Pragmatic Project Automation (см. <http://www.pragmaticautomation.com/cgi-bin/pragauto.cgi/Monitor/Devices>) есть множество записей, посвященных физическим устройствам, которые отслеживают состояние здоровья системы (как правило, серверы сборки). Именно этот сайт вдохновлял мою группу во время работы над предыдущим проектом. Мы приспособили светофор следить за нашим сервером сборки: «когда свет зеленый, сборка проходит нормально».

17.6. Стандарты де юре и де факто

В мире систем мониторинга господствуют стандарты. Как ни странно, они не дают нам той совместимости, которую можно было бы ожидать. Давайте посмотрим на протокол SNMP, 800-фунтовую гориллу от стандартов мониторинга, а также на его настоящих и будущих конкурентов.

Простой протокол сетевого управления

Дедушкой стандартов, связанных с мониторингом и системами управления, без сомнения, можно считать простой протокол сетевого управления (Simple Network Management Protocol, SNMP). Его первая версия появилась в 1988 году. Хотя изначально он был написан специально для управления сетевыми устройствами (именно поэтому в его имени есть слово «Network»), его функциональность вышла далеко за исходные границы. Сейчас он применяется для чего угодно, от маршрутизаторов Cisco операторского класса, обрабатывающих половину трафика в Интернете, до USB-периферии.

Даже в первой версии слово «простой» в названии этого протокола было большой натяжкой. В нем присутствовала концептуальная элегантность, которая делала его простым в том же самом смысле, в каком прост язык LISP: из единственной идеи извлекается ее суть и затем применяется ко всем возможным проблемам. Усвойте

¹ См. <http://peep.sourceforge.net/intro.html>.

эту идею, разберитесь в ней, и все остальное станет четким и простым. Но до этого момента она будет выглядеть крайне запутанной. Основная концепция протокола SNMP звучит так: «Все является переменной». Все, о чем может отчитаться или что может сделать узел, является переменной. Команд не существует, есть только операции присваивания переменных.

Протокол SNMP не дает устройству команду выполнить определенное действие, а посылает «запрос», предлагающий устройству установить переменную в нужное состояние. Например, информирование формирователя трафика Packeteer о новой ловушке прерываний через протокол SNMP выглядит как запись в переменную 1.3.6.1.4.1.2334.2.1.6.2.0, также известную как `trapDestAdd`. (Нужно быть членом комитета по стандартам, чтобы предлагать такие имена, как 1.3.6.1.4.1.2334.2.1.6.2.0.)

Хотя SNMP называют протоколом, все его стандарты описывают гораздо большую функциональность, чем функциональность простого сетевого протокола. Посвященный SNMP документ RFC также определяет информационную модель и своего рода метастандарт, касающийся способов задания новых модульных порций информационной модели.

Эта информационная модель называется структурой управляющей информации. Она задает базу управляющей информации (Management Information Base, MIB). Организация по назначению интернет-номеров¹ (Internet Assigned Numbers Authority, IANA) перечисляет основных владельцев модулей на странице <http://www.iana.org/assignments/smi-numbers>. В другом представлении перегруженной системы обозначений модульные определения обычно называют MIB. Производители называют MIB для описания управляющего интерфейса их устройств и программного обеспечения. Вот MIB-фрагмент от компании Packeteer²:

transparency/packeteer.mib

```
trapDestAdd OBJECT-TYPE
    SYNTAX DisplayString
    ACCESS write-only
    STATUS mandatory
    DESCRIPTION
        "A shortcut for adding a host to trapDestTable. If the name is not
        in n.n.n.n IP address form, then the agent attempts to look it up
        via DNS. If the operation fails with BADVALUE, try again with
        an IP address. The associated
        IP address is added to the trapDestTable if the operation succeeded.
        The table should be queried afterward to
        insure that the action took place. "
    ::= { psAdmin 2}
```

¹ См. <http://www.iana.org>.

² Со страницы <http://nagios.manubulon.com/mibs/Packeteer/packeteer.mib>.

Как видите, для описания MIB используется машинно независимый язык¹. Благодаря MIB система мониторинга умеет общаться с устройством или приложением. (Но обычно MIB при импорте в систему мониторинга сначала требуется скомпилировать в специальный формат.) Вооружившись взятыми из MIB определениями таблиц, типов и переменных, система мониторинга может задавать пороговые значения, триггеры и предупреждения об ошибках, отталкиваясь от текущего состояния наблюдаемой системы. Кроме того, протокол SNMP позволяет «агенту» — демону или библиотеке на сервере, который управляет самим протоколом SNMP, — асинхронно уведомлять систему мониторинга о возникновении интересных или опасных событий. Эти уведомления называются *ловушками* (traps). Ловушки менее универсальны, чем переменные; в основном они служат для привлечения внимания системы мониторинга.

С SNMP начали работать многие производители программного обеспечения. В Microsoft предложили SNMP-службу, которая позволяет получить доступ к компьютеру с любой операционной системой, начиная с Windows 2000. Существует SNMP-модуль для Apache. Он встроен в серверы WebSphere и WebLogic. Несколько баз управляющей информации, как стандартных, так и частных, поддерживает Oracle.

Что все это означает для ваших систем? Использование любой из этих платформ автоматически обеспечивает высокую степень поддержки SNMP, а значит, мгновенную прозрачность. Система мониторинга на базе SNMP в комбинации с платформой, поддерживающей SNMP, немедленно делает видимыми тысячи переменных. Еще такая система мониторинга позволяет задавать пороговые значения и политики уведомлений как о текущем поведении, так и об аномальном состоянии.

Но для кода вашего приложения картина выходит далеко не такой радужной. Проблема в том, что создание MIB для разрабатываемого своими силами программного обеспечения представляет собой гигантское свершение совершенно особой природы. Даже если у вас получится написать эту базу, администраторы систем мониторинга крайне неохотно идут на использование MIB от «ненадежного» источника. Мне не понять, почему группе разработчиков можно доверить написание приложений, отвечающих за миллионы долларов дохода, но нельзя доверить написание подключаемой базы управляющей информации для системы мониторинга. Тем не менее я часто наталкивался на странное сопротивление со стороны администраторов.

Кроме того, после написания MIB в ваше приложение нужно будет встроить SNMP-агента, который, как показано на рис. 39, будет служить мостом между SNMP-переменными и объектами вашего приложения. (Хотя в соответствии со стандартами для агента выделяется порт 161, существует возможность запустить несколько SNMP-агентов, каждый на своем порту.) Несмотря на то что SNMP считает MIB-переменные *объектами*, на самом деле они больше напоминают

¹ Если интересно, то это язык ASN.1. См. <https://ru.wikipedia.org/wiki/ASN.1>.

С-структуры. По сути, MIB определяет кучу глобальных переменных — как массивов, так и структур, — которые вовсе не обязаны точно проецироваться на объекты.

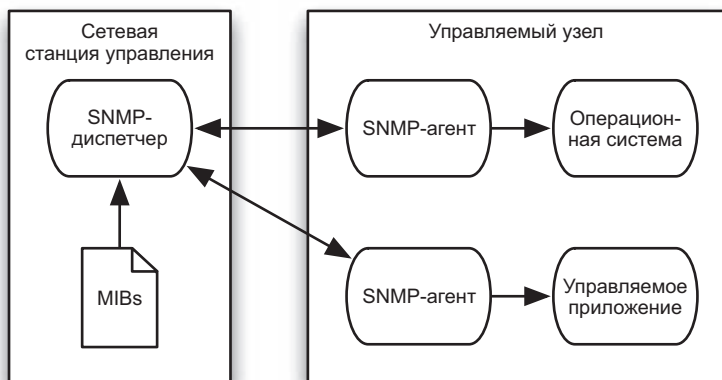


Рис. 39. Структура SNMP-взаимодействия

В Java-сообществе совместимость с SNMP лучше всего достигается написанием кода приложений с поддержкой управляющих расширений для Java (Java Management eXtensions, JMX) и применением коннектора JMX-SNMP.

CIM

В 1996 году организация Distributed Management Task Force¹ (DMTF) представила преемника и конкурента протокола SNMP. Так называемая общая информационная модель (Common Information Model, CIM) заменила неудобный язык ASN.1 и MIB-структуру протоколом метаобъектов и структурой-посредником, дававшими возможность динамической регистрации и обнаружения ресурсов.

Во многих отношениях CIM превосходит SNMP. Эта объектно-ориентированная модель намного лучше вписывается в современные стили программирования, а динамическое обнаружение различных ресурсов избавляет от необходимости приобретать дорогостоящие платформы управления.

В настоящее время общая информационная модель внедрена не так широко, как SNMP, хотя большинство производителей операционных систем, в том числе Microsoft, Sun, HP и IBM, поддерживают CIM. У Linux поддержка присутствует, но, как правило, в дистрибутивы не включается. Многие производители аппаратного обеспечения добавляют поддержку CIM к продуктам корпоративного класса. Поддерживают его и некоторые производители приложений, в частности Oracle.

¹ См. <http://www.dmtf.org>.

Однако поддержка на уровне приложений пока далека от масштабной, поэтому я считаю CIM моделью будущего, а не настоящего.

JMX

Управляющие расширения для Java (Java Management Extensions, JMX) исторически являются одним из самых недооцененных прикладных программных интерфейсов в Java. Появившиеся в 1998 году как JSR 3, они изначально задумывались в качестве части платформы Java Enterprise Edition. Но со временем ценность JMX оказалась столь значительной, что они как JDK 1.5 вошли и в вариант Standard Edition. Любая виртуальная Java-машина, работающая с Java 5 или выше, автоматически включает JMX.

Там, где протокол SNMP напоминает процедурные языки с их глобальными переменными, структурами и таблицами, управляющие расширения для Java предлагают объектно-ориентированное представление для управления приложениями. Основной управляющей единицей являются объекты **MBean**, своего рода «управляющие представители» некоторых базовых системных объектов, как показано на рис. 40. Открытые свойства — в том смысле, в каком их понимает JavaBean, — и методы класса **MBeans** могут активироваться средствами удаленного доступа.

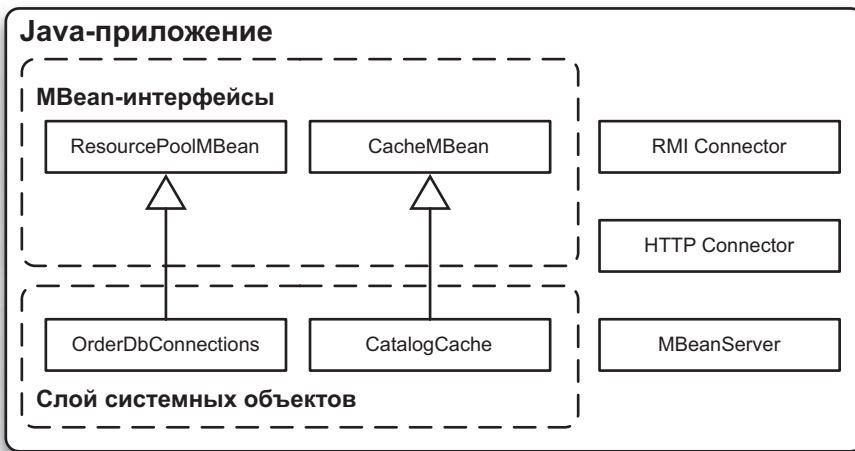


Рис. 40. Объекты MBean в роли представителей

У объектов **MBean** есть имена, определенные интерфейсом **MBeanServer**. Этот интерфейс располагается между внешними коннекторами, обеспечивающими удаленный доступ, и внутренними компонентами **MBean**. Внешние коннекторы адаптируют такие протоколы, как HTTP и RMI, к вызовам самого интерфейса **MBeanServer**.

(Обратиться к интерфейсу `MBeanServer` как к управляемому ресурсу можно из той же самой виртуальной Java-машины. Код приложения этого делать не должен. Это окольный способ взаимодействия с самим собой.) RMI-коннектор позволяет получить удаленный представитель управляющего компонента. Методы, вызванные для этого представителя, вызываются и для самого компонента. Это могут быть простые методы доступа к свойствам или методы управления.

Листинг 1. Пример интерфейса MBean

```
transparency/CircuitBreakerMBean.java
package com.example.util;

import java.io.IOException;

public interface CircuitBreakerMBean {
    public void setResetTime(long timestamp) throws IOException;
    public long getResetTime() throws IOException;

    public CircuitBreakerState getState() throws IOException;

    public void reset() throws IOException;
}
```

Листинг 2. Реализованный интерфейс MBean

```
transparency/CircuitBreaker.java
package com.example.util;

public class CircuitBreaker implements CircuitBreakerMBean {
    private CircuitBreakerState state = CircuitBreakerState.CLOSED;
    private long resetTime = -1;

    public void setResetTime(long timestamp) {
        this.resetTime = timestamp;
        checkForReset();
    }

    public long getResetTime() {
        return resetTime;
    }

    public CircuitBreakerState getState() {
        return state;
    }

    public void reset() {
        setState(CircuitBreakerState.CLOSED);
    }
    ...
}
```

Листинг 1 демонстрирует простой интерфейс управления для компонента предохранителя. Методы `setResetTime(long timestamp)` и `getResetTime()` определяют

свойство класса **JavaBeans**. Метод **reset()** вызывает изменение состояния в управляемом объекте. Это всего лишь интерфейс, значит, он должен реализовываться каким-нибудь классом. На деле этот интерфейс, как правило, реализует управляемый объект. В листинге 2 показан пример управляемого ресурса. Это означает, что прямое применение JMX уведомляет код о его будущей роли в качестве приложения, которым управляют и за которым пристально следят.

Другой тип компонентов **MBean** — «динамические объекты **MBean**» — не требует информации, получаемой на этапе компиляции. Аналогично интерфейсу динамического обращения в CORBA или API-отражению в Java динамические объекты **MBean** имеют возможность определять собственные методы, а не пользоваться статическими определениями, полученными на этапе компиляции. Это означает, что типичные динамические компоненты **MBean** должны уметь функционировать как управляющие представители любого старого Java-объекта. Разумеется, интерфейс **StandardMBean** делает ровно это. При наличии объекта и интерфейсного класса он превращается в компонент **MBean**, делающий этот интерфейс видимым объекту. Это самый быстрый способ добавить к вашему приложению поддержку JMX, одновременно избавившись от влияния JMX на системные объекты.

До Java 5 (JDK 1.5) управляющие расширения для Java были дополнением. Чтобы его использовать, приложениям приходилось включать реализацию JMX-спецификации. Шире всего были распространены расширения **Reference Implementation** от Sun и проект с открытым исходным кодом **MX4J**. Начиная с Java 5 JMX интегрируется в виртуальную Java-машину, и любое приложение может применять интерфейс **MBeanServer**.

Все еще редко используемые разработчиками приложений управляющие расширения для Java получили широкую поддержку у разработчиков платформ. Ядро сервера **JBoss** сократилось до реестра на базе JMX и загрузчика модулей. Продукты **WebSphere** от IBM используют JMX для обеспечения своей видимости. Где-то посередине оказывается **WebLogic** от BEA; значительная часть **WebLogic** активируется и управляется через JMX, но не до такой степени, как **JBoss**. Начиная с версии 1.2 в среду разработки **Spring** включается автоматический генератор динамических компонентов **MBean**. В типичной манере **Spring** он требует изрядного объема идиотской XML-разметки и не требует кода. (Пусть, в конечном счете, применение XML имеет смысл, но, как и в случае пророчеств Оракула, этот смысл становится понятным только в ретроспективе.)

Поддержка JMX предоставляет этим платформам одно из самых больших преимуществ: возможность написания сценариев. Как Oracle, так и IBM создали оболочки командной строки, обращающиеся к компонентам **MBeans** через удаленные коннекторы. Программы **wlst** от Oracles и **wsadmin** от IBM предоставляют серверам приложений с включенными объектами **MBean** интерфейсы оболочки с возможностью написания сценариев. (См. мое выступление о графических административных интерфейсах в разделе 14.4.) В **JBoss** есть «дополнение», которое,

не являясь полноценной оболочкой, позволяет отдавать команды произвольным компонентам MBean.

Добавляйте к административным интерфейсам возможность написания сценариев.

Получить похвалу от людей, занимающихся эксплуатацией систем, на удивление непросто. Но добавьте к административным функциям приложения возможность написания сценариев, и они будут благословлять ваше имя. Ценность административных интерфейсов, допускающих написание сценариев, невозможно описать. Добавление средствами JMX полезного набора компонентов MBeans — самый простой способ наделения подобной функциональностью Java-приложения.

УМОПОМРАЧИТЕЛЬНЫЕ УПРАВЛЯЮЩИЕ РАСШИРЕНИЯ ДЛЯ JAVA

Полная спецификация JMX позволяет получать крайне сложные варианты поведения, которые вряд ли когда-нибудь потребуются большинству разработчиков. Существует возможность динамически создавать экземпляры компонентов MBean, которые могут привести к активации целых подсистем. Собственными компонентами MBean, динамически возникающими и исчезающими, могут обладать и доменные объекты. Модель MBean является настолько динамической, что даже не существует во время компиляции — эти компоненты собираются путем программного определения интерфейса и присоединением любого объекта.

С точки зрения создания прозрачного приложения компоненты MBean лучше всего присоединять к долгоживущим компонентам архитектуры этого приложения: пулам ресурсов, кэшам, репозиториям, интерфейсам для внешних систем и т. п.

Эти динамические впечатляющие функциональные возможности приносят реальную пользу разработчикам платформ — команде, работающей над проектом с открытым исходным кодом JBoss, ордам ревностных разработчиков продуктов WebSphere в IBM, — но для разработчиков приложений они не имеют особой ценности. Скорее всего, при попытке углубиться в JMX вы потеряетесь в огромном количестве информации. Сделайте резервную копию, глубоко вздохните и поищите другие способы.

Что делать видимым

Если бы заранее можно было угадать, какие показатели будут ограничивать вычислительную мощность системы, показывать проблемы стабильности или другие системные неполадки, можно было бы следить только за ними. Но есть две проблемы. Во-первых, скорее всего, ваша догадка окажется неверной. Во-вторых, даже в случае правильного попадания ключевые показатели имеют свойство меняться со временем. Меняется код, меняются режимы нагрузок. И показатель, который станет ключевым через год, сейчас может попросту не существовать.

В идеале в приложении должны быть видимы все переменные состояния, счетчики и показатели. Так как вы не знаете, что вам может потребоваться в будущем, отображайте все. Кроме того, поскольку невозможно предсказать, каким должно быть пороговое значение и как реагировать на выход за его пределы, сосредоточьтесь просто на том, чтобы обеспечить видимость. Политики оставьте «на потом». Изначально обеспечьте общую видимость, а политики можно будет сформулировать позже.

Разумеется, можно приложить гигантские усилия и создать вообще для всего компоненты MBean или SNMP-переменные. Но так как система должна заниматься не только сбором данных, но еще и функционировать, я разработал ряд эвристических правил, помогающих решить, какие из переменных или показателей имеет смысл сделать доступными для наблюдения. Некоторые из них будут отображаться сразу. К другим может потребоваться добавить код, предназначенный в первую очередь для сбора данных. Вот что с моей точки зрения бывает полезно во всех случаях:

❑ Индикаторы трафика

Общее число запросов страницы, число запросов страницы, число транзакций, число одновременных сеансов.

❑ Состояние пула ресурсов

Включенное состояние, всего ресурсов¹, выгруженных ресурсов, наивысший предел, количество созданных ресурсов, количество уничтоженных ресурсов, сколько раз выгружался каждый ресурс, количество потоков, заблокированных в ожидании ресурса, сколько раз поток был заблокирован для ожидания.

❑ Состояние подключений базы данных

Количество выброшенных исключений `SQLExceptions`, количество запросов, среднее время ответа на запрос.

❑ Состояние точек интеграции

Состояние предохранителя, количество таймаутов, количество запросов, среднее время ответа, количество успешных ответов, количество сетевых ошибок, количество ошибок протокола, количество ошибок приложения, реальный IP-адрес удаленной конечной точки, текущее количество одновременных запросов, максимальное количество одновременных запросов.

¹ Это применимо к пулам подключений, пулам рабочих потоков и любым другим пулам ресурсов.

□ Состояние кэша

Количество элементов в кэше, сколько памяти использует кэш, частота обращений к кэшу, количество элементов, удаленных сборщиком мусора, настраиваемый верхний предел, время, потраченное на создание элементов.

Все эти показатели подразумевают временной компонент. Их следует читать, добавляя после «за последние *n* минут» или «с момента последнего запроса».

Комбинация JMX и SNMP

Технология JMX создавалась для поддержки различных коннекторов и адаптеров протоколов. Возможность открывать Java-приложения предоставляется протоколу SNMP через JMX. Более того, доступны несколько коннекторов разного качества для перехода JMX-SNMP. Лидером рынка в этой (правда, небольшой) области, по-видимому, является AdventNet (<http://www.adventnet.com>).

Самая большая проблема, с которой приходится бороться коннектору, это совмещение протоколов, работающих с разными типами данных. Преобразование операций протокола между SNMP и JMX происходит легко. В конце концов, протокол SNMP выполняет всего три типа операций: получить переменную, задать переменную и перечислить переменные. Напоминаю, что информационная модель SNMP базируется на схеме из переменных, таблиц и структур, в то время как модель JMX объектно-ориентирована. Коннектор JMX-SNMP нивелирует эту разницу интерфейсов. Он должен определять, какие переменные проецируются на свойства компонентов **MBeans**, какие иницируют вызовы методов и как превратить коллекцию объектов в записи таблицы. Наконец, коннектор JMX-SNMP должен проецировать JMX-уведомления (асинхронные события) на SNMP-ловушки.

В результате мы получаем полностью настроенный SNMP-агент, который используется интерфейсом **MBeanServer** виртуальной Java-машины и MIB для импорта в систему мониторинга. Как только все эти затруднения будут устранены, интеграция Java с поддержкой JMX в систему мониторинга на базе SNMP становится тривиальной.

17.7. База данных функционирования системы

Протоколирование и мониторинг позволяют обнаружить и понять мгновенное поведение приложения или системы. Но для получения информации о прошлом системы и для прогнозирования ее будущего эти механизмы не очень подходят. Рисунок 41 демонстрирует пригодность разных инструментов при

рассмотрении системы с разных точек зрения. Конечно, можно узнать текущее состояние системы только по файлам журналов, но это крайне сложная задача, так как требуется найти информацию о последнем изменении каждой значимой переменной состояния. (Конечно, анализ может упростить вывод переменных состояния по циклическому таймеру.)

Системы мониторинга дают нам превосходное представление о текущем поведении. Они великолепно демонстрируют текущее состояние системы, но, как я упоминал в разделе 17.5, говоря о недостатках коммерческих систем, они показывают происходящее только с одного ракурса. Когда дело доходит до прошлых или будущих тенденций, они мало чем могут помочь. И пока пакеты программ для управления корпоративными приложениями не разовьются в достаточной степени, этот недостаток нужно учитывать.

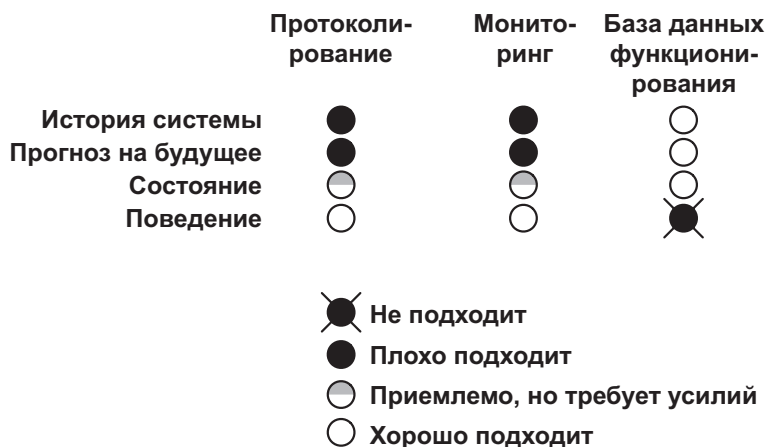


Рис. 41. Пригодность для целей технологии прозрачности

На рис. 42 представлена дополнительная технология, ориентированная на представление системы во времени. Эта «база данных функционирования системы»¹ (Operations DataBase, OpsDB) собирает данные о состоянии и показатели от всех серверов, приложений, пакетных заданий и источников данных, из которых состоит система в целом. Она предоставляет нам «одно окошко» на панели инструментов, содержащее коммерческие показатели, статистику системы и, что самое важное, корреляцию между первым и вторым. В то время как протоколирование дает информацию об одном приложении, более широкое OpsDb-представление объединяет в себе отчеты о состоянии и показателях системы в целом.

База данных функционирования системы является источником информации для неизменно популярной панели инструментов, показывающей текущее состояние.

¹ Я предпочел бы назвать ее «хранилищем рабочих данных», но это название уже используется для других целей.

Так как она предоставляет показатели спроса и системные показатели одновременно, небольшие упражнения по анализу данных позволят вам получать коэффициенты корреляции для планирования вычислительной мощности. Содержащиеся в этой базе данные за прошлые периоды автоматически дадут отправную точку для определения «нормальных» показателей по сайту в целом.

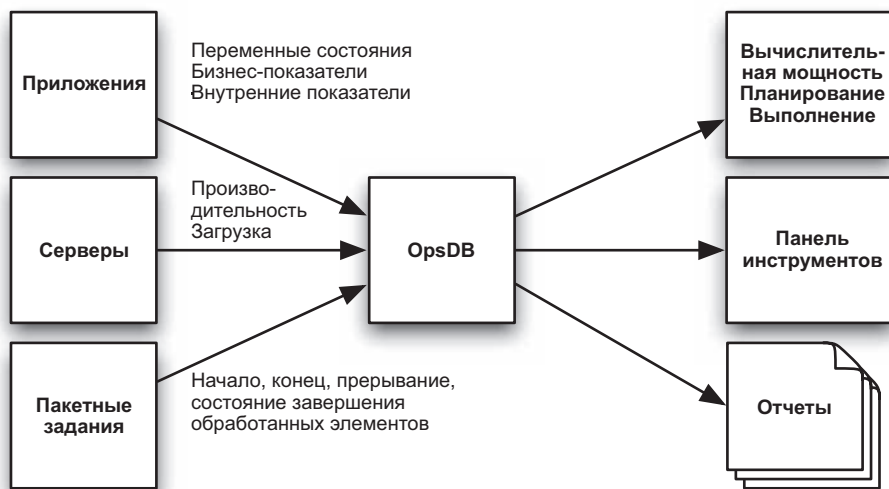


Рис. 42. Роль базы данных функционирования

Записи выполненных заданий ускоряют поиск и устранение неисправностей при возникновении проблем на уровне приложения, например устаревших данных от торгового партнера. Анализ обычного времени начала и окончания и соотнесение их с количеством обработанных позиций дает возможность прогнозировать время завершения заданий и вероятность не уложиться в эти сроки.

Высокоуровневая структура базы данных функционирования

На рис. 43 показана диаграмма UML-классов, представляющая информацию в базе данных¹. Проецирование этой объектной модели на SQL-схему я оставляю в качестве

¹ Те, кто читал книгу Мартина Фаулера *Analysis Patterns*, сразу обнаружат сходство с паттерном наблюдения (Observation). Этот паттерн описывает систему медицинской диагностики состояния здоровья пациента. Просто в данном случае вы имеете дело не с человеком, а с сетевой компьютерной системой — гораздо более простым созданием.

упражнения для читателей (в частности, потому, что форма этого проецирования зависит от конкретного ORM-инструмента и сервера базы данных).

Элемент **Feature** представляет собой единицу функциональности, влияющую на коммерческую составляющую. В него входят именно те функциональные возможности, доступность которых оговаривается в SLA и которые продемонстрируют, корректно ли была спланирована вычислительная мощность.

Источником данных для элемента **Feature** является не сервер. Один такой элемент может быть реализован по меньшей мере на трех серверах (веб-сервер, сервер приложений и сервер базы данных), на приложениях каждого уровня, на паре фаерволов, на сетевом коммутаторе и в массиве хранения или SAN.

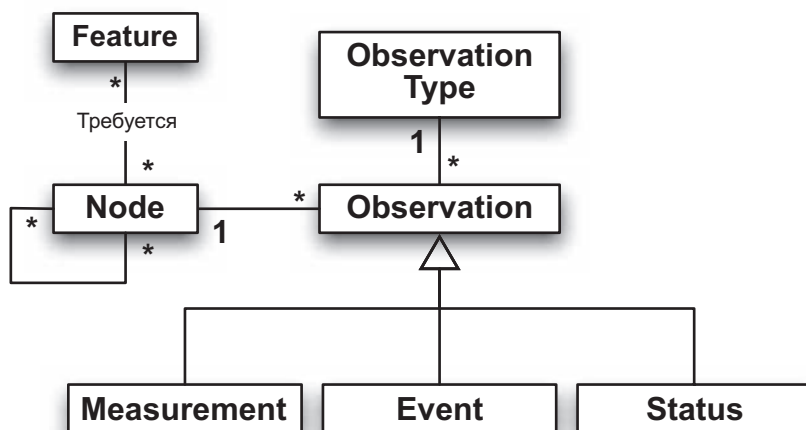


Рис. 43. Объектная модель OpsDb: наблюдения

Класс **Node** представляет любой из этих трех активных узлов. На практике обычно достаточно смоделировать хосты и приложения. Если система включает в себя аппаратные фаерволы, SSL-ускорители, серверы кэширования контента или другое сетевое оборудование, играющее активную роль в доставке приложения (в основном относительно 7-го слоя взаимодействий), скорее всего, имеет смысл включить их в число узлов. Каждый элемент **Node** имеет уникальный идентификатор. Отдельные серверы, приложения и пакетные задания используют этот идентификатор в отчетах о своих наблюдениях, поэтому за этим аспектом нужно пристально следить. Я обнаружил, что выделение блоков ID конкретным группам и подгруппам прекрасно работает. При этом сохраняется согласованность идентификаторов, что позволяет пользоваться ими, не дожидаясь выделения новых. Элементы **Node** используют другие элементы **Node**, но фиксация этих зависимостей порой оказывается слишком сложным делом, поэтому без нее можно обойтись. Тем не менее если вы решите составить полную схему, зависимости элемента **Node**

смогут оказать вам неоценимую помощь в деле поиска и устранения неисправностей, поскольку именно по этим зависимостям происходит распространение каскадных отказов.

Сердцем рабочей базы данных являются элементы **Observation**. Каждый из них представляет собой один замер элемента **Node**. Для таких узлов, как серверы, эти элементы предоставляют главным образом статистику производительности, записанную в элементах **Measurement**. Последние, как правило, являются периодическими, и все записи сохраняются. В случае приложения в этот элемент входят статистика производительности и состояние важных системных объектов. Например, сервер приложений должен фиксировать максимальные показатели пула соединений базы данных и выгруженные ресурсы (кроме прочих параметров), но вдобавок он должен фиксировать изменение состояния с «включено» на «выключено». Аналогичным образом все переходы между состояниями должен фиксировать предохранитель. Эти переходы записываются в объекты **Status**. Панели инструментов (показывающей текущее состояние) интересна только последняя запись **Status**. А вот для поиска и устранения неисправностей или для определения прошлых тенденций частота и тип изменений состояния имеют большое значение.

Чтобы элемент **Observation** имел смысл, обязательно требуется элемент **ObservationType**. Набор всех элементов **ObservationTypes** определяет информацию, охватываемую рабочей базой данных. Именно элементы **ObservationType** задают имя и тип каждого элемента **Observation**.

Передача данных в базу

Для передачи данных о наблюдениях в базу я рекомендую создать API на стороне клиента. Выберите для API написания язык, на котором написана большая часть системы. Но главное, этот прикладной программный интерфейс должен быть простым. Отказ может случиться где угодно, в том числе в базе данных функционирования. Ее наличие не влияет на коммерческий успех системы, поэтому нужно сделать так, чтобы ее отказы не отражались на основных системных функциях.

Присутствующие в системе сценарии оболочки или командные файлы для записи информации в нашу базу данных должны пользоваться какой-то программой командной строки.

Сценарий должен вызывать командную строку во время своего запуска, фиксируя, сколько элементов ему требуется обработать. В момент своего завершения он должен записать, сколько элементов удалось обработать (в противовес изначально поставленным перед ним заданиям). При нулевом количестве таких элементов следует сделать запись об аварийном прекращении работы.

Если это Java-система и вы ощущаете приступ инициативности, можно написать обобщенный компонент **MBean**, который будет отвечать за запись

периодической выборки. Более того, он может получать уведомления об изменении состояния и фиксировать их. После этого для оснащения любого приложения измерительными инструментами достаточно будет создать и настроить экземпляр этого MBean.

Применение базы данных функционирования

После того как массив данных становится частью OpsDb, он оказывается более полезным. На рис. 44 показаны новые классы, добавленные к модели, представленной на рис. 43. (На этой диаграмме не показаны элементы **Feature** и **Node**, но они по-прежнему существуют.)

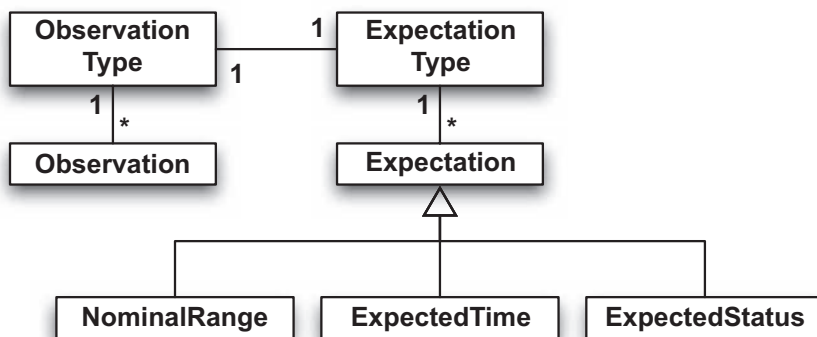


Рис. 44. Перспективные оценки базы данных функционирования

Элемент **ExpectationType** функционально эквивалентен элементу **ObservationType**. Он определяет имена и свойства элементов **Expectation**. Последние же представляют допустимый диапазон показателей, временные рамки, в которых событие должно (или не должно) происходить, и допустимое состояние. Нарушение любой из этих перспективных оценок должно вызывать сигналы о возможной ошибке в системе мониторинга.

Оптимальным источником информации для перспективных оценок являются уже содержащиеся в базе данные за прошедшие периоды. Перспективные оценки должны отражать реальное положение дел, чтобы избежать ложных срабатываний.

ОПАСНОСТЬ ЛОЖНЫХ СРАБАТЫВАНИЙ

В своей книге «Inviting Disaster» Джеймс Р. Чайлз рассказывает историю начальника производства, который увидел, как оператор рефлекторно отключил звуковой сигнал на диспетчерском пульте. Когда об этом зашла речь, оператор стал категорически отрицать, что он это сделал, но не из страха последствий, а потому что у него не отложилось осознанное воспоминание об отключении предупреждения. Система

настолько приучила его не обращать внимания на этот сигнал, что оператор выключал его автоматически.

Аналогичные истории я то и дело слышал от разработчиков и руководителей, носящих пейджеры. Одна женщина рассказывала мне, что ее пейджер пищал по три раза за ночь. И она знала, что это — нормальная ситуация. Если третий сигнал не приходил в определенный час, значит, возникла какая-то проблема. Если пейджер начинал пищать в четвертый раз, значит, опять же возникла какая-то проблема. Конечно, подобная форма ситуационной осведомленности лучше, чем ничего, но в качестве образа жизни я бы ее не рекомендовал.

На начальном этапе перспективные оценки могут быть достаточно общими. Но по мере усиления контроля над процессами они должны становиться более жесткими и сложными. К примеру, сначала приемлемый «коэффициент использования процессора веб-сервера» может находиться в диапазоне от 0 до 80 %. После получения дополнительной информации о взаимосвязях в системе диапазон может сузиться до меньше 5 и больше 50 %. На следующем уровне к расчетам может добавиться периодичность, связанная с коммерческой составляющей, как непрерывная огибающая или как ступенчатая функция. Например, в случае ступенчатой функции оценка может потребовать низкой загрузки процессора в ночное время, средней в утренние часы и высокой после полудня. Отклонения от этой оценки в меньшую или большую сторону будут вызывать сообщения о возможной ошибке.

Со временем OpsDb позволит системе стать более зрелой и самоуправляемой. Система лучше узнает саму себя и будет знать, как реагировать на внешние стимулы.

Для крупных систем OpsDb со временем может накопить много данных. Не забывайте об упоминаемом в разделе 5.4 паттерне стабильного состояния (Steady State). Без механизмов сжатия эти данные могут заполнить всю систему. Очевидно, что не стоит ставить под угрозу вычислительную мощность, сохраняя слишком много ископаемых данных о производительности. Когда данные становятся ископаемыми? Это зависит от вашей системы, но могу сказать, что с каждой минутой от данных, возраст которых превышает неделю, становится все меньше проку.

17.8. Вспомогательные процессы

Как часто вы сталкивались с ситуацией, когда система генерирует отчет, передает его в список рассылки, а у половины подписчиков программа Microsoft Outlook настроена на автоматическое удаление таких отчетов? Это классическая ситуация. И каждый раз, сталкиваясь с ней, я понимаю, что эти отчеты не имеют смысла. Даже

хуже того. Не только их генерация отнимает время и силы, есть те, кто поддерживает существование этих отчетов в новых версиях системы. Ужаснее всего то, что раз в сто лет в таком отчете может появиться действительно серьезная информация, но к тому времени все предполагаемые адресаты давным-давно прекратят эти отчеты читать.

В данном случае мы имеем дело с примером прозрачности без замкнутой обратной связи. На внедрение и поддержку прозрачности тратятся средства, но пользы она не приносит. Зато дает иллюзию безопасности. Самые лучшие данные в мире не помогут, если никто не будет их читать. В этом разделе я отступлю от систем мониторинга, чтобы обсудить более крупные динамические образования — организации, создающие эти компьютерные системы и управляющие ими.

Эффективную обратную связь можно описать как «ответные действия на значимую информацию». Прозрачность системы всего лишь обеспечивает доступ к данным. Но при этом принимающий участие в этом цикле человек должен увидеть и интерпретировать эти данные.

Существуют различные формы циклической обратной связи, например цикл Деминга¹ «планирование-действие-проверка-корректировка» или нелинейный цикл «О-О-D-A»² Джона Бойда. В конечном счете все эти формы сводятся к следующему:

- ☐ Проанализируйте систему: текущее состояние, ранее возникавшие паттерны, прогнозы на будущее.
- ☐ Проинтерпретируйте данные. Этот процесс всегда происходит в контексте ментальной модели человека, занимающегося интерпретацией.
- ☐ Оцените потенциальные действия, в том числе их стоимость, и, как вариант, вообще ничего не предпринимайте.
- ☐ Выберите план действий.
- ☐ Внедрите выбранный план действий.
- ☐ Наблюдайте за новым состоянием системы.

ПОЛКОВНИК ДЖОН БОЙД И ЦИКЛ О-О-D-A

Этот вариант процесса обратной связи возник непосредственно из работ полковника Джона Бойда, пилота ВВС и стратега. Ключевой концепцией полковника Бойда был «цикл О-О-D-A», что расшифровывается как Observe-Orient-Decide-Act (наблюдай-ориентируйся-решай-действуй). Бойд утверждал, что полностью понять ситуацию невозможно. И даже если в какой-то момент получается это сделать, информация

¹ См. <http://deming.ru/TeorUpr/PDSA.htm>.

² См., например, монографию А. А. Ивлева «Основы теории Бойда». — *Примеч. пер.*

быстро устаревает. Он рассматривал ситуацию на поле боя, как постоянно хаотически меняющуюся. В таких условиях победа доставалась стороне, которая лучше оценивала и контролировала меняющиеся тактические и стратегические взаимодействия.

Цикл O-O-D-A требует корректных наблюдений, не затуманенных попытками выдать желаемое за действительное или необъективностью, — это крайне сложная задача. Ориентация представляет собой процесс обновления ментальной карты возможностей и вариантов в соответствии с предыдущей картой и новыми результатами наблюдений.

Хорошая ориентация дает понять, что возможно, а что нет. В военных и корпоративных условиях влияние политики на наблюдения и ориентацию приводит к фатальным последствиям. «Подтасовка фактов» противоречит циклу O-O-D-A, который требует взаимодействовать с реальным окружением, а не с идеализированной или приукрашенной картиной этого окружения.

Цикл O-O-D-A включает множество цепей обратной связи. Наблюдения, решения и действия влияют на ориентацию действующего лица. В процессе действия меняется как внешняя среда, так и ваше собственное понимание изменяющихся условий. Соответственно результат итерации цикла O-O-D-A влияет на следующую итерацию двумя способами: через саму среду и через ваше восприятие этой среды.

На каждом витке цикла вы знаете больше, чем раньше, что формирует усиливающую обратную связь. При этом каждый раз, когда вы видите циклы усиливающейся обратной связи, можно подозревать, что перед вами нелинейная и хаотичная система в современном смысле этих слов.

Эта способность к хаосу отличает цикл O-O-D-A от простой схемы «стимул-реакция», в которой усиливающийся стимул приводит к пропорционально усиливающейся реакции. Вспомните окопную войну в период Первой мировой или попытки получить более быстрое программное обеспечение за счет выполнения большего количества операций. Вы можете получить огромную созидательную силу, позволяющую победить оппонентов, пройдя через цикл O-O-D-A быстрее, чем они.

Бойд называет это проникновением «внутрь» цикла решений врага: одновременно превосходя и ограничивая его действия¹.

Ключ к наблюдению

В первую очередь наблюдатели должны отслеживать тенденции и аномальные значения. Оба этих аспекта позволяют проникнуть в суть происходящего. Они помогают создать рабочий ритм, который делает улучшения обычным явлением,

¹ Еще я считаю цикл O-O-D-A мощным аргументом в пользу гибких методов разработки. Каждая итерация разработки должна проецироваться на итерации цикла O-O-D-A. Появляющийся в результате контроль среды позволяет компании взять инициативу на себя и выявить конкурентов. См. статью Стива Адольфа по адресу <http://www.iohai.com/iohai-resources/agile-lessons-ooda.html>, в которой связь между O-O-D-A и гибкими методами разработки раскрывается более подробно.

а не плодом судорожных усилий. Вот несколько вспомогательных процедур, помогающих наблюдениям:

- ❑ Каждую неделю просматривайте присланные пользователями на прошлой неделе уведомления о неисправностях. Обращайте внимание на повторяющиеся проблемы и проблемы, решение которых требует больше всего времени. Обращайте внимание на вызывающие много проблем подсистемы или группы разработчиков (если имеется несколько групп). Ищите проблемы, связанные с конкретным сторонним производителем или точкой интеграции.
- ❑ Каждый месяц проверяйте общее количество проблем. Учитывайте их распределение по типам. По мере исправления серьезных проблем общие тенденции будут смягчаться. Сократится и их количество. (Но при этом общая зависимость будет иметь пилообразную форму, так как появление новых версий кода будет сопровождаться новыми проблемами.)
- ❑ Ежедневно или еженедельно ищите в файлах журналов исключения и результаты трассировки стека. Сопоставляйте найденные данные, чтобы определить, откуда чаще всего возникают исключения. Проанализируйте, что является их причиной — серьезные проблемы или просто отсутствие кода, обрабатывающего эти ошибки.
- ❑ Рассмотрите вопросы, с которыми чаще всего обращаются в службу поддержки. Они могут дать вам представление как о путях совершенствования пользовательского интерфейса, так и об узких местах, в которых следует повысить надежность системы.
- ❑ Если отчетов о неисправностях и обращений в службу поддержки слишком много для детального изучения, проанализируйте только самые частые случаи. Также случайным образом просматривайте заявки, чтобы обнаружить те, которые заставляют вас задуматься.
- ❑ Каждые четыре-шесть месяцев проверяйте корректность старых корреляций.
- ❑ Хотя бы раз в месяц смотрите на объемы данных и статистику запросов.
- ❑ Проверяйте самые ресурсоемкие запросы на сервере базы данных. Влияет ли на них каким-то образом план запросов? Попадает ли новый запрос в число самых ресурсоемких? Любое из этих изменений может указывать на накопление данных. Вызывают ли самые распространенные запросы сканирование таблиц? Это может означать отсутствующий индекс.
- ❑ Анализируйте ежедневную и еженедельную диаграммы спроса (задающие переменные) и системные показатели. Меняются ли шаблоны трафика? Внезапное уменьшение трафика в популярное время может означать, что система в эти часы работает слишком медленно. Есть ли плато у задающих переменных? Это указывает на некий ограничивающий фактор, возможно, на скорость отклика системы.

Помните, что со временем центр внимания будет смещаться. Сначала вы в основном столкнетесь с проблемами, требующими реакции. Основной интерес будут представлять обзоры уведомлений о неисправностях, заключительные отчеты об отказах и последние тренды. По мере устранения основных причин неисправностей, выхода новых версий и изменения шаблонов трафика акцент сместится с реагирования на упреждающий анализ. Что случится в следующем квартале? Как все должно работать в это же время через год? В процессе этого смещения следует перейти от обзора старых данных к обзору новых трендов. Как только показатель перестанет давать вам полезную информацию, прекратите обращать на него внимание. Отчеты, жизненно необходимые через месяц после запуска, через два года станут бесполезными или даже сбивающими с толку.

Для каждого анализируемого показателя учитывайте следующие факторы. Как он выглядит по сравнению с исторически установившейся нормой? (При наличии в OpsDb достаточного количества данных легко приступить к составлению прогнозов.) Если показатель укладывается в тренды последнего времени, что происходит с другими связанными с ним показателями? Как долго может сохраняться тренд и какой ограничивающий фактор может его нарушить? Какие явления может обусловить этот ограничивающий фактор?

К примеру, рассмотрим соотношение между числом полученных заказов и загрузкой процессора сервера приложений. На любом сайте интернет-магазина эти показатели в той или иной степени связаны друг с другом. (Количество полученных заказов является производной от количества уникальных посетителей. И связывает эти два показателя величина, которую маркетологи называют *коэффициентом обращаемости*.) Соответственно, при увеличении количества заказов сразу возникает вопрос: как долго может сохраниться тенденция к увеличению? Ответ на этот вопрос подталкивает к интерпретации и принятию решения. Это может быть решение ничего не делать, увеличить количество ресурсов, сократить число покупателей или оптимизировать код приложения. Любое из этих решений представляет собой успешное завершение цикла обратной связи.

17.9. Заключение

При столкновении с реальным миром в системах начинают происходить странные вещи. При этом любые отклонения и любые события могут остаться непонятными или же из них можно извлечь ценные уроки. Именно прозрачность отличает систему, которая совершенствуется со временем, от системы, топчущейся на месте или даже деградирующей.

Широко распространены системы управления. Самой популярной в настоящий момент является SNMP версии 3. Системы CIM/WBEM лишены многих

недостатков SNMP и в ближайшем будущем обещают выйти на первый план. Java-программисту возможность работать как с SNMP, так и с CIM/WBEM предоставляет технология JMX.

Прозрачность требует доступа к внутреннему устройству компьютеров и программного обеспечения. Обязательным условием для этого является обеспечение видимости этих внутренних структур. Затем требуются средства сбора и анализа контрольных данных. Для этого подходят и коммерческие системы мониторинга, и OpsDb. Наконец, необходима реакция на собранные таким способом сведения.

В следующей главе мы поговорим о том, что делать, когда наблюдения сигнализируют о необходимости изменений. *Адаптация* требует анализа динамики показателей за длительный период с прицелом на построение систем, способных корректно меняться со временем.

Из слов всевозможных, что сходят с пера,
Грустней быть не может, чем
«Эх, знал бы тогда...»

Джон Уиттьер

18

Адаптация

Какие бы оптимистические прогнозы вы ни строили или насколько серьезного подхода ни требовали бы обстоятельства, запускаемая система всегда будет не такой прекрасной, как вам бы того хотелось. Совсем новая система — вещь несовершенная и менее масштабная, чем вы предполагали, и слабо подходящая для решения задач, которые изначально перед ней ставились. Останется ли она на том же уровне или вырастет до уровня, задуманного ее создателями?

Настоящее рождение системы происходит не в день ее проектирования и разработки и даже не в момент появления идеи проекта, а в момент ее запуска в эксплуатацию. Это начало, а не конец. Со временем система будет расти и развиваться. Она будет получать новые функциональные возможности. Она будет избавляться от дефектов и, возможно, получать новые. Она станет такой, какой задумывалась, или, что еще лучше, такой, какой должна быть. Но прежде всего она обязана меняться. Система, которая не может адаптироваться к окружающей среде, является мертворожденной.

18.1. Адаптация со временем

Новая система должна корректно делать некоторые вещи, иначе ее запуск попросту не состоялся бы, кроме того, она могла бы делать некоторые другие вещи так, как это задумывали проектировщики. Тем не менее определенные программные компоненты могут работать не так, как изначально предполагалось, и даже могут оказаться более сложными, чем должны были быть. По сути, это означает несоответствие между формой системы и занятым этой формой пространством решений.

Новая система не может автоматически совершенствоваться в процессе эксплуатации. Большую часть времени ее приходится осваивать пользователям — трудно и мучительно. Систему можно изменить, подогнав под пространство решений, даже если это пространство продолжает меняться с течением времени, но это может быть только преднамеренное действие.

В своей книге *Evolution of Useful Things* Генри Петровски¹ утверждает, что старое высказывание «форма определяется функцией» неверно. Вместо этого в качестве правила развития дизайна он предлагает постулат «форма определяется отказом». То есть движущей силой изменения конструкции таких банальных вещей, как вилка или скрепка, были те аспекты, в которых эти вещи функционировали плохо. Даже скромная канцелярская скрепка изначально не имела привычного для нас вида. И каждая ее новая версия была, по сути, работой над ошибками.

В сфере разработки программного обеспечения необходимость каждой следующей версии обуславливается новыми конструктивными особенностями (заполнением пробелов) или исправлением ошибок (сглаживанием шероховатостей). Точно так же, как со временем менялась форма канцелярской скрепки, архитектурные и проектные решения систем ПО на каждой итерации постепенно адаптируются к реальным и предполагаемым отказам.

Но эта адаптация имеет свою цену. Любое действие по изменению системы стоит денег: проектирование, разработка, тестирование плюс стоимость внедрения. Физики и химики называют это «энергией активации». И если затраты на предполагаемые изменения не превысят доходов, которые будут получены благодаря заполнению пробелов или сглаживанию шероховатостей, рациональным будет *отказ от изменений*. Возмещение может выражаться исключительно в терминах денежных потоков. А может лежать в сфере защиты крупных инвестиций, ранее сделанных в систему.

Денежные потоки — жизненные соки любого бизнеса. Деньги приходят и уходят. Из-за изменения стоимости денег с течением времени нынешние денежные потоки ценнее будущих.

¹ Не могу не выразить восторг автору, который на восемнадцати страницах увлекательно рассказывает о том, как появилась обычная вилка.

Наконец, практика показывает, что от 40 до 90 % затрат на разработку возникает после выхода первой версии. Иногда они проходят по категории «техническое обслуживание», а иногда рассматриваются как новая разработка. В любом случае на данный вид издержек приходится идти долгое время после версии 1.0. Это старая, всеми признанная проблема с проектными решениями и архитектурой. Намного хуже обстоят дела с пониманием сложности и стоимости внесения изменений.

В этой главе мы поговорим о том, как компактные и крупномасштабные архитектурные решения влияют на способность системы к адаптации. Это часть энергии активации, включающая в себя «затраты на внедрение». Сочетание потребностей и энергии активации дает нам стоимость внесения изменений — скрытые затраты, которые могут вырасти до удивительных высот. Наша конечная цель состоит в том, чтобы гарантировать, что изменения нашей системы — заполнение пробелов и устранение шероховатостей — принесут больше денег, чем потребуются на их реализацию. Химические реакции, в процессе которых выделяется больше энергии, чем требуется для их возникновения, называются *экзотермическими*. Нам нужно, чтобы вносимые в наши системы изменения были экзoeкономическими, то есть приносили больше денег, чем требуется на их инициирование.

18.2. Проектирование адаптируемого ПО

Проектированию программного обеспечения, способного к адаптации, посвящены тысячи страниц. Десятки известных методик направлены на создание программ, идеально вписывающихся в функциональное пространство. Многие новые методики включают в себя даже принципы изменения со временем — это так называемые гибкие методы. Перечислить все попросту невозможно.

Подавляющее большинство этих методологий концентрируется на реализации корректной функциональности или на предоставлении этой функциональности возможности меняться со временем. Но в проектировании программного обеспечения есть и другие аспекты, относящиеся к способности системы к адаптации без прекращения ее эксплуатации. Именно эти аспекты мы и рассмотрим. Их можно считать «верхним слоем», который должен располагаться поверх выбранных вами методов проектирования.

Внедрение зависимостей

Имена обладают собственной силой. Стоило Мартину Фаулеру присвоить относительно распространенной методике собственное имя, в мире стали появляться среды разработки, поддерживающие «внедрением зависимостей»¹ (см. также AJAX).

¹ <http://www.martinfowler.com/articles/injection.html>.

Принцип внедрения зависимостей состоит в том, что компоненты должны взаимодействовать через интерфейсы и не должны напрямую создавать экземпляры друг друга. Вместо этого обязан присутствовать некий агент, «собирающий» приложение из слабо связанных компонентов.

Корректно выполненное внедрение зависимостей способствует ослаблению взаимозависимости, до такой степени мощным является его влияние. Заодно оно облегчает модульное тестирование. Главным ключом к достижению гибкости при внедрении зависимостей является определение и использование интерфейсов. Взаимодействие объектов посредством интерфейса позволяет безболезненно заменить любую из конечных точек. В качестве замены может выступать как реальная конечная точка с новой функциональностью, так и подставной объект, используемый для модульного тестирования. Внедрение зависимостей с применением интерфейсов сохраняет возможность внесения локальных изменений.

ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ

Когда на горизонте только появилась технология J2EE, некоторые создатели сред разработки начали строить такие вещи, как Pico, Spring и Apache Avalon. Эти продукты заметно отличались по качеству от типичных приложений J2EE. В частности, написанные в этих «облегченных» средах приложения, как правило, делали акцент на небольших автономных компонентах, реализующих интерфейсы. Они не создавали экземпляры других компонентов, а вызывали через интерфейсы некие службы и «предполагали», что реализация указанного интерфейса будет осуществляться указанным контейнером.

Сначала это описывалось как инверсия управления, общая характеристика сред разработки приложений. Но эта техника используется и в более универсальной практике расширения сред разработки путем добавления кода приложения в виде подклассов или обратных вызовов, в результате чего зависимой оказывался среда разработки, а не код приложения. (Среда разработки обращается к приложению, что является типичной инверсией, отличающей «среду разработки» от «библиотеки классов».)

Мартин Фаулер ввел понятие **внедрения зависимостей** (dependency injection), пытаясь описать вставку ссылок в компоненты. Во время выполнения контейнер связывает компоненты друг с другом в соответствии с конфигурационным файлом или определением приложения. При этом самим компонентам вовсе не обязательно знать, с чем они взаимодействуют. Внедрение зависимостей позволяет получить хорошо адаптируемый код, прекрасно приспособленный к модульному тестированию и легко «перестраиваемый» при изменении требований.

Объектное проектирование

Удивительно, сколько эмоций выражение «хорошее объектно-ориентированное проектирование» вызывает даже сейчас, спустя двадцать лет после начала «объектной революции». Я заявляю, не утруждая себя дальнейшими доказательствами, что такая вещь, как хорошее объектно-ориентированное программирование, действительно

существует. Те, кто придерживается подобной точки зрения, поверят мне и без доказательств, а ее противников никакие доказательства не убедят.

В контексте адаптации лучше всего следовать старым добрым принципам: слабая взаимозависимость и сильная связанность. Изначально они применялись к исходным модулям. Слабая взаимозависимость означает «Не трогай глобальные переменные в других файлах». Сильная связанность означает: «Большинство или все подпрограммы в этом файле должны использовать большинство или все глобальные переменные из этого файла». Но, кажется, эти правила пора обновить.

Современное объектно-ориентированное определение взаимозависимости больше относится к вариантам поведения, чем к переменным. Каждый раз, когда один класс использует поведение другого, он становится зависимым от него через это поведение. Количество вариантов поведения, которые один класс требует от другого, определяет «ширину» интерфейса между этими классами.

Очевидно, что некоторая степень взаимозависимости нужна. Не обращающиеся друг к другу объекты могут сделать не много. Что же в этом контексте означает «слабая взаимозависимость»? Предположим, что класс экспонирует все открытые методы. И существует подмножество этих методов, вызываемое другими классами. При идеальной связанности класс будет иметь всего один такой набор, включающий в себя все методы, вызываемые всеми пользователями класса. Это практически невероятная ситуация. Куда чаще встречается малое число таких наборов. Различные пользователи класса обращаются к разным наборам методов. Чем сильнее отличаются друг от друга такие наборы, то есть чем меньше методов присутствует одновременно в нескольких наборах, тем проще впоследствии изменить поведение такого класса. Он будет более адаптируемым. Не случайно и то, что в нем будет проще создавать бесклассовые интерфейсы для группирования, именования и обобщения этих наборов методов.

При наличии объединенных в подмножества методов связанность описывает, насколько сильно внутреннее состояние объекта влияет и попадает под влияние этих подмножеств. Другими словами, если набор методов затрагивает только подмножество состояний объекта, но при этом не попадает под влияние других аспектов этого состояния, объект не обладает связанностью. Это может означать, что внутри находится еще один объект, ожидая преобразования в собственный класс.

Взаимозависимость влияет на адаптацию сильнее, чем связанность. Чтобы сильно взаимозависимые классы стали полезными, требуется более сильный *внешний контекст*. К примеру, рассмотрим простой геометрический класс, представляющий собой точку в двух измерениях. Для полноценного функционирования такому классу вряд ли понадобится какой-либо внешний контекст. Он прекрасно работает сам по себе, без приложения, добавляющего ему какие-то взаимосвязи или глобальное состояние (например, объекты-одиночки или реестры объектов). В некотором смысле это кирпичик; он используется одним и тем же способом всякий раз, когда требуется соединить друг с другом нечто массивное.

Также рассмотрим чрезмерно интеллектуальный доменный объект, например обычный объект `Customer`. Скорее всего, он взаимодействует с тремя или четырьмя другими классами, например `Account`, `Address`, `CreditCard` и, возможно, каким-то еще. Если объект `Customer` знает что-либо о своем постоянном состоянии, он может также взаимодействовать с классами `PersistenceManager`, `TableDataGateway` и подобными им. Каждое из этих взаимодействий необходимо, и каждое создает проблемы. Без них объект бесполезен; с ними он зависит от наличия и корректного поведения крупных внутренних структур. Это уже не кирпичик, а скорее многомерный фрагмент головоломки. У него имеются ручки, подключения и сочленения, которые для обеспечения его работоспособности должны точно совместиться с другими фрагментами.

Кластер объектов, которые могут существовать только в рамках плотного взаимодействия, напоминает кристаллическую решетку в металлах. Объекты сильно связаны друг с другом, как атомы в такой решетке. Чем меньше размер кристаллов металла, тем выше его способность к деформации. Более пластичные металлы лучше восстанавливаются после нагрузки. Крупные кристаллы способствуют формированию трещин. В программном обеспечении крупные «кристаллы» затрудняют редактирование. Объекты одного кристалла, принимающие участие во множестве вариантов взаимодействия, соединяют между собой несколько кристаллов, формируя более крупную структуру и сильно снижая пластичность программного обеспечения.

Область распространения таких тесно связанных кристаллов неограниченна. В самых экстремальных случаях кристаллы разрастаются до границ приложения. При этом каждый объект уже служит всего одной цели, для которой он в высшей степени приспособлен. Он идеально подходит к своему месту и, по большому счету, связан со всеми остальными объектами. Такие кристаллические дворцы порой даже по-своему красивы. Они не признают улучшений, отчасти из-за невозможности поэтапной доработки, отчасти потому, что не существует фрагмента, перемещение которого не приводило бы к перемещению всех остальных частей. Как правило, это мертвые структуры. Разработчики ходят вокруг них на цыпочках, разговаривая шепотом и пытаясь ничего не трогать.

ВАРИАНТЫ ВЗАИМОДЕЙСТВИЯ И ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Может показаться, что я не очень разбираюсь в такой теме, как взаимодействие. Но это не так. Более того, в сфере проектирования объектов я признанный специалист по поведению. Взаимодействие крайне важно. Но я настаиваю на том, что при проектировании вариантов взаимодействия следует учитывать связанность и уменьшать степень «кристаллизации» программного обеспечения. Объект, присутствующий в нескольких паттернах проектирования, каждый из которых взаимодействует с разными группами объектов, я воспринимаю как гордиев узел, который крайне сложно или даже невозможно изменить.

Там, где возникает необходимость связать различные кристаллы, количество этих связей, пересекающих границы кристаллов, должно быть по возможности сведено к минимуму.

Кстати, именно тут многие новички сворачивают не туда, пытаясь применять знания, полученные из великолепной книги «банды четырех» «Паттерны проектирования». Эта книга предлагает взглянуть на взаимодействие объектов больше с позиции их совместного использования и вариантов поведения. У проектировщиков-новичков появляется соблазн оценивать качество проекта по количеству задействованных в нем паттернов, хотя Гамма и его соавторы прямым текстом предостерегают против этого. Чрезмерная роль паттерна для одного объекта приводит к появлению кристаллов все большего размера, пока весь код не превратится в один гигантский жесткий кристалл.

Приемы экстремального программирования

О религиозных войнах программистов ходят легенды: emacs против vi, знак табуляции против пробела, VMS против UNIX, Java против .NET. В этот список в настоящее время следует добавить войну сторонников гибкого программирования против приверженцев традиционных методов.

К сожалению, любая религиозная война заставляет полностью отвергнуть новый подход пылкое меньшинство, возглавляющее оппозицию, и не определившееся со своими пристрастиями большинство. Именно это случилось с экстремальным программированием (Extreme Programming, XP). Многие разработчики и руководители проектов отвергли его из-за отсутствия дисциплины и отрыва от реальности. (Как выяснилось, они были не правы, но это тема для другой книги.) И это обидно, ведь они отрезали себя от двух ценнейших практик написания кода, разработанных с момента появления объектного подхода: рефакторинга и модульного тестирования.

В книге *Refactoring* Мартин Фаулер собрал и документировал случаи из своей растущей практики. Он назвал рефакторинг «улучшением структуры имеющегося кода без изменения его функциональности». Это вовсе не пустая трата времени, а ключ к адаптивности. Рефакторинг действует как постоянная сила, поддерживающая сложность проектного решения на минимальном уровне, необходимом для поддержки функциональных возможностей программного обеспечения. Он препятствует усилению кристаллизации и способствует увеличению уровня обобщения базовых классов ПО.

Модульное тестирование стоит в одном ряду с рефакторингом. Более того, многие утверждают, что без первого не было бы второго. Изменение проектного решения без подушки безопасности в виде модульных тестов больше напоминает случайные переделки кода и скорее приведет к появлению новых ошибок, чем к усовершенствованию конструкции.

Экстремальной формой модульного тестирования является разработка через тестирование (Test-Driven Development, TDD). В TDD сначала пишется модульный тест, который служит функциональной спецификацией. Затем пишется фрагмент кода,

позволяющий пройти этот тест, и ни строкой больше — см. также принцип YAGNI (You Ain't Gonna Need It! — Вам это не понадобится!). После этого вы получаете возможность провести рефакторинг кода, улучшающий проектное решение при гарантированном прохождении тестирования. Комбинация TDD, рефакторинга и принципа YAGNI естественным образом превращается в код, хорошо поддающийся адаптации.

Второй менее выраженный эффект достигается путем последовательного модульного тестирования. Объект никогда не следует называть «допускающим многократное использование», пока им не воспользуются хотя бы еще один раз. Подвергнутый модульному тестированию объект задействуют в двух контекстах: как код готового приложения и как собственно тест. Значит, такой объект с большей вероятностью допускает многократное применение. Тестирование означает, что вам потребуются заглушки, или так называемые мок-объекты. Для этого зависимости объекта должны проявлять себя как свойства, обеспечивая свою доступность для внедрения в итоговый код программы. Как только объекту требуется глубокая настройка в его внешнем контексте (как в случае упоминавшегося объекта *Customer*), его модульное тестирование сильно затрудняется. Распространенной — и не самой удачной — реакцией является прекращение модульного тестирования таких объектов. Куда лучше сократить объем необходимого внешнего контекста. В случае доменного объекта *Customer*, убрав ответственность за сохранение состояния, мы сократим объем внешнего контекста, требующего поддержки. Это облегчит модульное тестирование, заодно уменьшив размер кристалла объекта *Customer* и сделав этот объект более гибким. Накопительный эффект от множества таких небольших изменений оказывается очень глубоким.

Гибкие базы данных

Хотя термин *гибкие базы данных* (agile databases) не является оксюмороном, случайно подобные вещи не появляются. Если в области ИТ и есть что-то, сопротивляющееся изменениям больше, чем определения логической структуры базы данных, это разве что CICS-транзакции на мейнфреймах, то есть то, что недоступно для редактирования, потому что их создатель умер в конце 70-х годов прошлого века.

Но если поставить во главу угла поведение, то становится очевидным, что изменение функциональности кода приложения требует соответствующих изменений логической структуры базы данных. Почему же базы данных так часто противостоят изменениям? Прежде всего, не следует забывать, что далеко не все согласны ставить поведение во главу угла! Многие организации явно или неявно утверждают, что данные намного важнее функциональности приложений¹. В других организа-

¹ Распознать такие организации с первого взгляда можно по схемам потоков данных или гигантским средам Захмана.

циях архитекторы баз данных отделены от разработчиков приложений. Каждая группа думает, что им нужно превзойти соперников. Разработчики хотят, чтобы схема менялась в соответствии с нуждами приложений. Архитекторы считают себя хранителями данных — не только их представления, но и самого смысла. Более того, у базы может быть несколько потребителей, каждый со встроенной информацией о логической структуре и ее интерпретации.

Это неверное противопоставление. Ни у поведения, ни у данных не может быть абсолютного приоритета. Программисты могут и будут искать обходные пути любых препятствий, в том числе жесткой схемы. Они будут перегружать определения столбцов, добавлять индикаторы типов, а в CLOB-объектах упаковывать данные в XML-строки. Они будут реализовывать соотношения в коде, не поддерживающем проверку ссылочной целостности данных в логической схеме, или изобретать алгоритмы кодирования на основе справочных таблиц, существующих в коде только как тип «enum». Изменение поведения при неизменной схеме приведет только к более изощренным злоупотреблениям этой схемой, в конечном счете демонстрируя такие варианты применения, которые заставят архитекторов в ужасе отскочить. Это семантическое загрязнение распространяется и на других пользователей данных, затрудняя работу с данными и внося в нее ошибки.

Итак, я показал, что логическая структура данных должна меняться. Теперь нужно подумать, как сделать эти изменения по возможности максимально безболезненными. Первым делом в схеме следует позаботиться о собственных зависимостях прикладного кода. Объектно-реляционное проецирование (ORM) позволяет легко обновить приложение. Но существуют некоторые ограничения, в частности, касающиеся версий и их контроля. Запуск приложения при наличии схемы базы данных, не совпадающей с ORM-метаданными, дает непредсказуемые результаты. К примеру, среду Hibernate можно настроить на проверку файлов проецирования и метаданных базы при запуске, но эта процедура будет занимать время и зависеть от уровня поддержки метаданных производителем базы. Не обнаруженные при запуске несовпадения логической схемы и метаданных приведут к странным ошибкам выполнения, вызывающим откат транзакции (с отправкой пользователям отчетов об ошибках) или, что еще хуже, повреждение данных.

Каждая логическая схема должна содержать таблицу с текущими результатами пересмотра структуры. Таблица может состоять из одной строки и одного столбца — номера версии. По крайней мере, при запуске приложения нужно проверять номер версии на совместимость. И в соответствии с принципом быстрого отказа прекращать работу, если не в состоянии пользоваться указанной схемой базы данных. Кроме того, номер версии может применяться для автоматических обновлений схемы, таких как миграции в Ruby on Rails¹. Он будет иметь значение и позже, при обновлениях «на лету». Семантика данных может меняться и без изменения логической структуры базы, поэтому указывайте номер версии также и при изменениях в интерпретации.

¹ См. ActiveRecord::Migration в проекте RDoc.

18.3. Адаптируемая архитектура предприятия

Некоторые архитекторы стремятся создать современный аналог *Колосса: проекта Форбина* (или *Трона*, если вы принадлежите к моему поколению)¹. Они работают над представлением эффективно интегрированного предприятия, функционирующего как единая машина. Каждый фрагмент является необходимой частью целого, идеально подходящей для своей роли. Программы и программисты обитают внутри машины, служа ее нуждам. Я смотрю на такие утопии с глубоким подозрением.

Подобная архитектура развивается сверху вниз, обычно начинаясь с гигантской среды разработки, например среды Захмана или среды TOGAF. Архитекторы рассматривают ее как самостоятельную сущность с простыми системами, заполняющими ячейки матрицы архитектуры предприятия. При наличии власти они приостанавливают проекты до определения архитектуры предприятия. В противном случае им остается терзаться от того, что один некондиционный проект за другим запускается в эксплуатацию, не используя преимуществ такой архитектуры.

Такая точка зрения основывается на двух ошибочных предположениях. Во-первых, считается, что архитектура может быть завершена. Утверждение, что архитектура предприятия «сформирована», означает конец изменений. Но прекращение изменений в данном случае означает, что организация навсегда застревает во времени.

Состояние бездействия — это смерть. Во-вторых, развитие сверху вниз предполагает, что организация умеет останавливать время, отказываясь от изменений до момента, пока не будет определена архитектура предприятия. В данном случае речь идет как о реальных издержках, так и о стоимости упущенных возможностей.

Я до некоторой степени преувеличиваю, чтобы четче донести основную мысль. Целью в данном случае является не искажение аргументов, а подчеркивание диктаторских тенденций приверженцев нисходящего проектирования. Представление о предприятии как о полностью интегрированном, четко определенном целом предполагает механистический взгляд на системы. Но большие, сложные машины предполагают множество нежелательных режимов отказа. Они часто ломаются. Они могут выйти из строя из-за поломки одного фрагмента. Проектирование и построение подобной архитектуры *требует* такой иерархии управления и контроля, которую попросту невозможно получить.

Самым убийственным является требование, чтобы изменения одновременно возникали в далеко отстоящих друг от друга группах, — еще одно применение закона

¹ Если ни одно из этих названий ни о чем вам не говорит, см. <http://www.kinopoisk.ru/>. — Примеч. пер.

Конвея. По мере роста количества этих привязанных к определенной версии систем предприятие начинает испытывать все большие сложности: каждую систему приходится менять при изменении любой из смежных систем (потому что такова природа сильной взаимозависимости). В экстремальном случае, например при изменении протокола сервисной шины предприятия (Enterprise Service Bus, ESB), все входящие в нее системы должны быть обновлены одновременно. Представьте себе риски, связанные с одновременным развертыванием новой версии каждой из критически важных систем предприятия! По большому счету, сопутствующие затраты и риски настолько велики, что крупномасштабные изменения протокола становятся попросту невозможными. В результате ESB либо «закостеневает», либо деградирует, совсем как логическая структура статической базы данных. Как только технология ESB достаточно устаревает, ее вытесняет более новая технология, не требующая затрат, связанных со сложностью.

Меня настораживают механистические метафоры, когда речь заходит о предприятии. Механические системы демонстрируют именно те признаки, которых лучше избегать; они одновременно жесткие и хрупкие. Избежать этих признаков можно, черпая вдохновение в биологических и экологических сравнениях. Я рассматриваю организацию как экосистему. Люди и системы занимают в ней свои ниши. Они обмениваются ресурсами с окружающей средой, главным образом, в виде информационных потоков. Отдельные ниши в экосистеме могут быть населены сразу несколькими сущностями. Например, мне доводилось видеть компании с не менее чем семью независимыми, частично совместимыми реализациями ERP-систем от SAP. Каждая обладала собственной генеалогией в зависимости от того, с каким из корпоративных приобретений она появилась. Каждая обладала собственными компонентами, источниками питания (системами, предоставлявшими данные) и симбионтами (системами, которые использовали результаты ее работы). Как это часто бывает, все это было ужасающе неэффективным, но крайне надежным. Каждая реализация могла модифицироваться независимо от остальных, позволяя отдельным фрагментам адаптироваться куда быстрее, чем если бы они были разработаны архитекторами с «единственным и неповторимым» представлением о корпоративных системах.

На реальном предприятии все всегда куда более запутанно, чем допускает архитектура. Новые технологии никогда не вытесняют старые до конца. Обнаруживается мешанина интеграционных технологий, начиная от передачи с пакетной обработкой неструктурированных файлов и заканчивая передачей сообщений публикации/подписки. Любая стратегия, сформулированная с претензией на создание монокультуры — будь то единая технология интеграции или один язык программирования, — обречена стать дорогостоящим провалом. Представьте компанию, которой удалось остаться в рамках единственного языка. Все ПО предприятия написано на Ada, Smalltalk, Pascal, C или каком-то другом старом языке. Учитывая время, которое требуется языку на то, чтобы получить достаточное распространение, можно утверждать, что кандидатам будет не менее десяти лет. В качестве подтверждения можно взять язык Java, в настоящее время считающийся «корпоративным

стандартом» в ряде крупных интернет-магазинов. Повторюсь: эта искусственная жесткость не будет служить нуждам предприятия.

По сути, самым полезным критерием оценки архитектуры является вопрос: начинают ли информационные технологии лучше отвечать нуждам пользователей? В большинстве случаев при разработке архитектуры предприятия подобной цели не ставится, скорее при этом учитываются нужды ИТ-группы. («Что хорошо для ИТ, хорошо для фирмы».) С учетом этого можно сделать так, чтобы архитектура предприятия вытекала из паттернов взаимодействия отдельных систем. Я не предлагаю полную анархию; она тоже не оптимизирует ИТ-сферу под нужды пользователей. Скорее в данном случае работает комбинация разных сил: выделенный бюджет, утвержденное расписание, пожелания к конструктивным особенностям, прямые и не прямые затраты на реализацию этих пожеланий, господствующая технология, потери в процессе совмещения всех этих факторов. Успешная архитектура предприятия требует динамического решения всех этих задач — паттернов взаимодействия, способствующих работоспособности организаций.

Спрашивайте себя: «Сможет ли эта архитектура сделать так, чтобы информационные технологии стали лучше отвечать нуждам пользователей?»

Зависимости внутри системы

Кластеризация в системах должна быть свободной. В несвязанном кластере потеря отдельного фрагмента значит для всей совокупности не больше, чем потеря одного дерева для леса. Например, удаление одного из десятка экземпляров Apache практически никак не повлияет на общую функциональность¹. Аналогичным образом исчезновение одного экземпляра сервера приложений не будет иметь особого значения для работоспособности приложения или службы.

Это означает, что отдельные серверы не исполняют разноплановые роли или, по крайней мере, любая роль доступна в более чем одной службе. Серверы WebLogic и WebSphere этот принцип нарушают. Они требуют наличия уникальных узлов, управляющих кластерами. Утрата этих узлов ведет к отказу всей службы.

Члены свободного кластера могут загружаться и останавливаться независимо друг от друга. Требований упорядочивания активации членов кластера во времени просто не существует.

Члены одного кластера или уровня не должны иметь зависимостей с отдельными членами другого уровня и даже не должны знать об их наличии. Зависимости должны замыкаться на виртуальный IP-адрес или имя службы, представляющее

¹ При условии адекватной общей вычислительной мощности системы (см. главу 8).

кластер как целое. Прямые зависимости отдельных членов создают жесткие связи, мешающие индивидуальному редактированию конечных точек. Это представлено иллюстрирует рис. 45. Экземпляры вызывающего приложения обращаются только к имени службы, предоставленному кластером 2. Члены кластера 1 не должны ничего знать о конкретных хостах и приложениях, обеспечивающих работу службы.

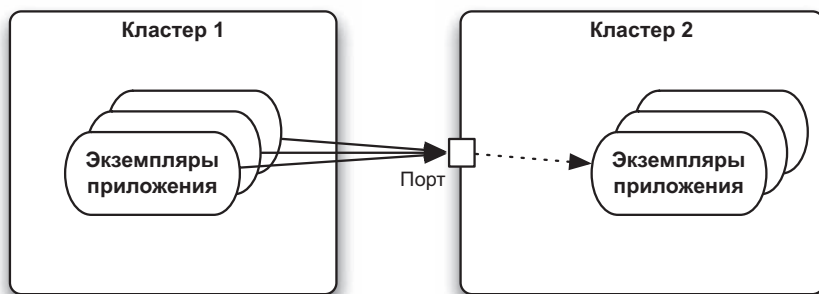


Рис. 45. Зависимости от служб, а не от отдельных экземпляров

Члены кластера никогда не должны знать идентификаторы всех членов другого кластера. В противном случае возникает $O(N^2)$ требований к изменениям, которые затрудняют добавление и удаление членов кластера. Также появляются предположения к паттернам взаимодействия «один ко многим», негативно влияющим на вычислительную мощность. Рассылка уведомлений, например сообщений о недействительности кэша, должна происходить путем передачи по типу публикации/подписки или очереди команд.

Зависимости между системами: протоколы

Как и в случае взаимозависимости между кластерами объектов, взаимозависимость между системами в рамках предприятия вызывает аналогичное окостенение. Сильная взаимозависимость систем затрудняет изменение с каждой из сторон интерфейса. Еще раз напомним, что в хорошей архитектуре необходимость изменений считается фундаментальным принципом — механизмом, ведущим к улучшению, а не чудовищем, которое следует взять под контроль.

Любой интерфейс определяется протоколом. Это может быть низкоуровневый протокол, передающий пакеты в сокетах, высокоуровневый, такой как CORBA, DCOM или RMI; или промежуточный вариант, например XML поверх HTTPS. Но в любом случае обе стороны интерфейса должны говорить на одном языке и понимать этот язык. И рано или поздно этот язык неизбежно приходится менять. При этом возможны два варианта развития событий. На рис. 46 обе системы

меняются одновременно. При этом обязательно появляется время простоя, так как развертывание не может произойти мгновенно. Подобная ситуация может быть приемлемой, если системы связаны только друг с другом (хотя при этом становятся невозможными обновления «на лету»). Но как только у систем появляется обычная паутина интеграции с другими системами, календарь изменений становится бессмысленным, потому что невозможно обеспечить требуемое для поддержания всех развертываний время простоя.

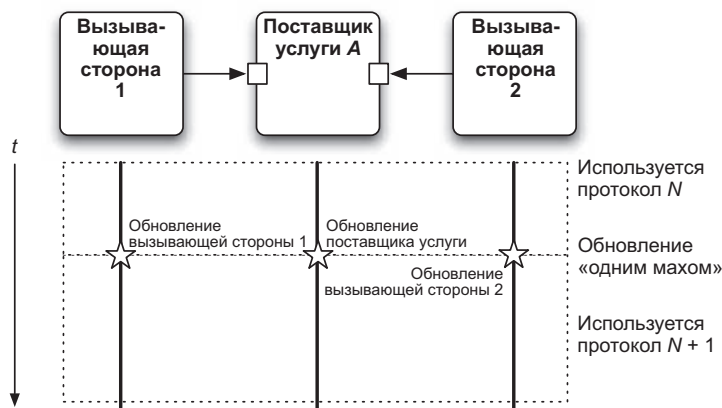


Рис. 46. Одновременные изменения в обеих конечных точках

Еще хуже то, что каждая новая версия протокола требует достаточного времени опережения на прохождение циклов разработки и тестирования. Предположим, что с системой А на рис. 46 работает быстрая группа разработчиков, применяющая гибкие методы с двухнедельными итерациями. Они смогут адаптироваться к новому протоколу примерно за две недели. Но если группа разработчиков, занимающаяся второй системой, по какой-то причине не может справиться за две недели, группа А должна замедлиться, чтобы сохранить общий темп. При этом у второй группы может быть кварталный цикл выпуска! Экстраполируя эту ситуацию на другие системы, с которыми интегрирована система А, можно сказать, что быстрый двухнедельный цикл в итоге будет растянут до самого медленного цикла разработки в рамках данного предприятия¹.

Очевидно, что подобной взаимозависимости лучше избегать. Она оказывает нежелательное влияние на время выхода на рынок, стоимость развертывания и доступность системы. Протоколы следует проектировать таким образом, чтобы любая конечная точка могла меняться независимо от остальных. Решение лежит в управлении версиями протоколов. Рисунок 47 демонстрирует, каким образом

¹ Именно это явление описывается в книге *The Goal* Элияху Голдрата, когда Алекс сталкивается с идущими в поход бойскаутами. Независимо от полученных распоряжений самый быстрый член группы достигнет цели вместе с самым медленным.

изменений формата требует определения способов интерпретации более старых версий.

Если перекрытие сроков, показанное на рис. 47, возникает только во время развертывания одной из систем, его продолжительность может измеряться минутами или часами. Если же у разработчиков второй системы применяется квартальный (или еще более долгий) цикл выпуска, этот период может растянуться надолго.

Зависимости между системами: базы данных

Интеграция баз данных — это то, чего делать никогда не следует! Даже при наличии представлений. Даже с хранимыми процедурами. Поднимитесь на уровень вверх и включите базу данных в веб-службу. Затем сделайте эту службу избыточной и доступной по виртуальному IP-адресу. Создайте тестовую систему, проверяющую, что происходит, когда веб-служба перестает работать. *Так выглядит* технология интеграции предприятия. Проникновение в другую базу данных выглядит попросту... неприлично.

Ничто не мешает способности системы к адаптации больше, чем другая система, заглядывающая в ее внутренности. «Интеграции» баз данных — это однозначное зло. Они нарушают инкапсуляцию и сокрытие данных, демонстрируя самые интимные подробности внутренней работы системы. Они способствуют ненужной взаимозависимости как на структурном, так и на семантическом уровне.

Хуже того, система, которая выставляет базу данных на обозрение посторонним, не может доверять данным из этой базы. Другие субъекты могут добавлять и модифицировать строки, даже если у владельца в памяти есть объект, спроецированный из этих строк. Возможен обход жизненно важной логики приложения, приводящий к запрещенным или недостижимым состояниям¹.

База данных, доступ к которой имеется у нескольких систем, становится связующим звеном жесткости. Все такие системы должны обновляться одновременно при любом изменении базовой схемы. (Еще раз упомяну, что наиболее вероятной в этом случае видится ситуация, когда базовая схема вообще не будет меняться.)

Конечно, иногда бывают случаи, когда системам действительно требуется доступ к большим объемам чужих данных. Это может быть необходимо с целью

¹ Настоящие фанаты баз данных сразу начинают говорить, что сама база должна быть хранилищем всей бизнес-логики. Мне уже доводилось слышать подобное. В принципе, звучит убедительно. Доступны базы данных, написанные на тьюринг-полных языках. На практике мы имеем некорректную логику в неверном месте с неверными инструментами и процессами, которые следует поменять.

составления отчетности, применения инструментов бизнес-аналитики или, к примеру, для доставки торговым партнерам или финансового анализа. В этих случаях можно найти ряд альтернатив, не предполагающих проникновения внутрь другой системы.

Исключая торги на бирже, для достижения большинства целей не требуется немедленный доступ к производственным данным. Если вы усматриваете необходимость в подобной срочности, почитайте про задержку, присущую процессу анализа и принятия решений (см. раздел 17.8).

Скорее всего, к моменту, когда будут предприняты какие-либо действия, если это вообще произойдет, данные уже окажутся устаревшими на часы (по меньшей мере). Насколько важно в этом случае получить именно производственные данные? Снимок состояния, сделанный час назад, будет таким же своевременным. (Совсем другая история, когда обратная связь занимает всего несколько секунд.)

Иногда требуется передача больших объемов производственных данных. В этих случаях для физического и семантического проецирования желательно использовать инструменты извлечения, преобразования, загрузки (Extract-Transform-Load, ETL). После определения преобразований эти инструменты можно настроить и указать предельные значения, чтобы не ставить под удар производственные мощности. Идентификатор, обеспечивающий функционирование извлеченных данных, можно настроить таким образом, что допустимой окажется только операция выбора, но не вставки, удаления или обновления. Для передачи больших объемов данных также хорошо подходят «материализованные представления» Oracle. В зависимости от архитектуры систем хранения данных вам могут также оказаться доступными операции с так называемыми BCV-томами, позволяющими перемещать данные из одной сети хранения данных в другую.

18.4. Безболезненный переход к новой версии

В одной из моих любимых торговых фирм процесс перехода к новой версии напоминал пусковую последовательность в NASA. Он начинался в полдень и продолжался, пока не наступали предрассветные утренние часы. В былые времена в этот процесс вовлекалось более двадцати человек. Сейчас их меньше дюжины. Как несложно догадаться, любой процесс, включающий в себя такое количество исполнителей, требует детального планирования и координации. Так как каждый переход к новой версии происходит сложно, количество подобных мероприятий в год невелико. Малое число новых версий приводит к тому, что каждая оказывается уникальной. Эта уникальность требует дополнительного планирования, что еще сильнее затрудняет появление новых версий, вообще отбивая желание часто проводить подобные мероприятия.

Появления новых версий должны стать столь же обыденным событием, как новая стрижка (или компиляция нового ядра для UNIX-хакера). Литература, посвященная гибким методам, упрощенная разработка и увеличенное финансирование должны создавать предпосылки для частых обновлений с прицелом на удобство пользователя и коммерческую ценность¹. А в случае промышленной эксплуатации проявляется еще одно преимущество частого появления новых версий. Вы становитесь настоящим профессионалом по части обновления и разворачивания.

Как отмечалось в главе 17, для улучшений важен быстрый цикл обратной связи. Чем быстрее этот цикл работает, тем более точными станут улучшения. А для этого необходимы частые обновления. Кроме того, частое появление новых версий с дополнительной функциональностью позволяет компании опережать конкурентов и задавать тон на рынке.

Как часто бывает, новые версии стоят слишком дорого и приносят слишком большой риск. Как уже отмечалось, из-за объема прилагаемых физических усилий и необходимости координации с трудом удастся обеспечить три-четыре новые версии в год. Но произвести таким способом двадцать обновлений попросту невозможно. Остается простое, но пагубное решение: замедлить график обновлений. Но реагировать на проблему таким образом — все равно что реже ходить к стоматологу, потому что это больно. Проблема при этом только усугубляется. Куда правильнее сократить количество необходимых усилий и число исполнителей, а также сделать процесс автоматизированным и стандартным.

ТОЧКА ЗРЕНИЯ ОБСЛУЖИВАЮЩЕГО ПЕРСОНАЛА

Один из моих любимых руководителей проекта часто напоминает, что люди «имеют позитивные намерения». То есть когда между сотрудниками или группами возникает конфликт, исходить следует из того, что все его стороны движимы одним и тем же желанием принести пользу фирме. В процессе самых ужасных конфликтов «мы» против «них» может показаться, что противники просто пытаются нам досадить, но подобное случается редко. Конфликты обычно возникают потому, что разные группы руководствуются разными мотивами.

При обсуждении частоты обновлений также важно исходить из позитивных намерений. Инвесторы и разработчики смотрят на новые версии как на генератор положительных изменений. Но при этом разработчики первым делом концентрируются на выполненных требованиях, исправленных ошибках и улучшениях архитектуры. Инвесторов же в первую очередь интересуют конструктивные особенности, позволяющие получить доход, удержать клиентов, догнать и перегнать конкурентов. С их точки зрения изменение означает улучшение.

¹ Если вас не убедят книги *Agile Software Development* Алистера Коуберна, *Lean Software Development* Мэри и Тома Поппендигов, *Software by Numbers* Марка Денни и Жана Келланд-Хванг и *Extreme Programming Explained* Кента Бека, то вас не убедит ничто.

Когда о новой версии речь заходит в отделе эксплуатации, там в первую очередь рассматривается потенциальная возможность новых режимов отказа, необходимость новых процедур и новых систем мониторинга. Они видят, что старая система мониторинга становится неприменимой. Они видят, что для новой версии, скорее всего, потребуется большее количество «четырёх ключевых» ресурсов: процессора, памяти, хранилища данных и полосы пропускания. И прежде всего, они видят риск. Факторы риска подразумеваются самим процессом развертывания, новой версией и тем, что эта версия представляет собой нечто неизвестное в сравнении с существующей базой кода.

И все они — инвесторы, разработчики и отдел эксплуатации — по-своему правы. Новая версия означает сразу все перечисленное.

Слишком высокая стоимость развертывания

Везде, за исключением производственных компаний, стоимость обновлений практически никогда не учитывается при планировании бюджета. Учитываются планирование, анализ, разработка и тестирование, и это при том, что прямые и косвенные затраты на обновления часто оказываются весьма значительными.

Прямые затраты в основном связаны с работами, предшествующими обновлению. Помимо тестирования перед нами стоят задачи в пяти областях.

❑ Управление конфигурациями

Создание ветви новой версии, метки базы кода, создание сборки новой версии.

❑ Документирование

Примечания к новой версии и новые или обновленные учебные материалы.

❑ Маркетинговые коммуникации

Обновление маркетинговых материалов и объявления внутри и вне компании.

❑ Развертывание

Планирование, выполнение и проверка развертывания.

❑ Поддержка

Аварийное исправление ошибок, обновления системы мониторинга, обновления документации по заданию.

Самые большие косвенные расходы возникают из-за простоя в процессе обновления. Системы, действующие только в «рабочее время», встречаются крайне редко. Или, если взглянуть с другой стороны, теперь «рабочее время» распространилось на весь циферблат. Времена, когда ночью наступало «время обслуживания системы», давно прошли.

Отделы эксплуатации, скорее всего, рассчитывают доступность, исходя из незапланированного времени простоя, поэтому уровень доступности 99,5 % означает, что за месяц система может преподносить сюрпризы в течение менее чем 216 минут. Но это всего лишь уловка. В период обновлений система может выйти из строя на пять часов. Волнует ли пользователей системы, запланирован или нет данный простой? Нет! Для пользователя система попросту не работает. Если человек не может закончить свою работу, значит, система не функционирует! Всегда помните, что пользователи, как правило, понятия не имеют о графике работы групп разработки и эксплуатации. И их заботит только собственный график.

Даты выхода новых версий

Почему мы придаем такое значение датам выхода новых версий? Для производителей готовой продукции, таких как Microsoft или Adobe, они действительно важны. О них объявляют за год, устраивают масштабные мероприятия по запуску, рассказывают в СМИ, проводят массированную подготовку клиентов и каналов сбыта. Но никто не будет занимать очередь в полночь, чтобы получить новейшую версию программы от 99,9 % остальных производителей.

В большинстве случаев наши клиенты даже не знают о запланированных датах выхода новых версий. Программное обеспечение, работающее на базе интернет-технологий, просто в один прекрасный день меняется, не причиняя клиентам особого беспокойства. Если это произойдет позже на день или на неделю, клиент не придаст этому никакого значения и даже об этом не узнает. Его заботит только доступность системы и относительная безошибочность ее работы.

Не ставьте клиентов под удар, запуская новую версию в день, выбранный случайным образом.

Тем не менее я встречал организации, где пытались запускать в эксплуатацию новые версии еще до их полной готовности. Один раз я принимал участие в собрании, посвященном вопросу «запускать или нет» и происходившем в 4 часа вечера в день выхода новой версии. Отдел контроля качества говорил, что тестирование еще не пройдено, но через час у них появится дополнительная информация. Несмотря на это, собравшиеся проголосовали за запуск. Я возражал, что

без прохождения тестов продукт не может считаться готовым. Можем ли мы создавать рискованную ситуацию для клиентов только ради того, чтобы новая версия появилась именно в день, случайным образом выбранный несколько месяцев назад? Ответ оказался очевидным и не имел ничего общего с соображениями о благе клиента!

Развертывание «на лету»

Серверы и архитектура высокой доступности сделали неприемлемым простой на время обслуживания оборудования, почему же мы до сих пор лояльно относимся к простоям, связанным с обновлениями ПО?

Но так или иначе, практически каждое появление новой версии кода требует простоя оборудования на время ее запуска. Это может быть час или день, но тот факт, что данный период называют «запланированным простоем», не избавляет от издержек. Если стоимость простоя составляет 10 000 долларов в час, четырехчасовое развертывание обойдется в 40 000 долларов, вне зависимости от того, по графику оно происходит или нет.

Почему же для запуска новых версий приложений требуется останавливать работу системы? Как ни парадоксально, но причиной является та самая архитектурная особенность, которая призвана увеличивать время безотказной работы: избыточность. То, что входящие запросы обрабатываются набором серверов, означает, что в процессе развертывания всегда будут как серверы с новой версией кода, так и серверы со старой. В это время одновременно действуют версии N и $N+1$. В случае полностью автономного программного обеспечения такая ситуация не составляет проблемы. Но корпоративные приложения крайне редко бывают автономными. Они пользуются базами данных, веб-службами и поисковыми машинами. Для сайтов они генерируют URL-адреса, связанные с таблицами стилей, JavaScript-файлами и файлами мультимедиа. Все эти многочисленные ссылки создают массу предпосылок для конфликта версий. И проще всего на время развертывания остановить работу системы.

Впрочем, развертывание можно выполнять в течение длительного периода времени — дней или даже недель, — избегая конфликта версий и обеспечивая совместное существование двух вариантов ПО. Для структурирования развертывания «на лету» требуется сотрудничество отделов разработки и эксплуатации, вот почему оно реализуется так редко. Тем не менее добавленные заблаговременно дополнения к архитектуре позволяют избежать простоя на время развертывания.

Развертывание «на лету»: подробная временная шкала

□ Расширение

- Развертывание новых статистических файлов (изображения, JavaScript-файлы, таблицы стилей)

- Создание новых сервисных пулов, если требуется
- Добавление новых таблиц
- Добавление новых столбцов
- Запуск сценариев переноса данных
- Добавление активаторов перенесения программ
- Применение «рекурсивных ZDD-диаграмм» для подготовки вторичных кластеров
- Выгрузка
 - Для каждого сервера
 - Распаковка кода на сервере
 - Прекращаем принимать новые запросы
 - Остановка работы сервера
 - Указатель на новый код
 - Запуск сервера
 - Подтверждение успешного запуска
- Очистка
 - Удаление активаторов перенесения программ
 - Удаление устаревших отношений целостности данных
 - Удаление устаревших столбцов
 - Удаление устаревших таблиц
 - Добавление новых отношений целостности данных
 - Добавление ограничений NOT NULL
 - Удаление устаревших статических файлов
 - Удаление старого кода
 - Удаление старых сервисных пулов

Это ключ к разбиению процесса развертывания на отдельные фазы. Вместо того чтобы сразу добавлять, менять и удалять различные фрагменты, например столбцы и таблицы базы данных, ограничители и службы, добавляйте новые элементы на ранней стадии, обеспечивая прямую совместимость со старой версией кода. После развертывания новой версии удаляйте то, что больше не имеет отношения к делу, и добавляйте новые ограничители, которые могли помешать работе старой версии. Мы рассмотрим эти фазы более подробно.

Расширение

Первым делом нужно добавить новые «фрагменты». Фрагменты состоят из ресурсов на базе URL, конечных точек веб-служб, таблиц и столбцов баз данных и т. п. И при определенных условиях все это можно добавить, не мешая работе старой версии ПО.

Ресурсам на базе URL, таким как таблицы стилей, изображения, анимация или JavaScript-файлы, следует в каждой новой версии присваивать новые URL-адреса. Если приложение генерирует страницы, ссылающиеся на адрес `/static/styles.css`, любые изменения файла `styles.css`, скорее всего, приведут к конфликту версий. Если же приложение ссылается на файл `/static/1.1/styles.css`, ничто не мешает развернуть файл `/static/1.2/styles.css`, не вызывая никаких конфликтов.

В случае веб-служб каждая версия интерфейса должна получить новое имя конечной точки. Аналогичным образом, задание новых имен (например, при помощи цифры, указываемой после имени) интерфейсов удаленных объектов для каждой версии гарантирует, что как старая, так и новая версия ПО получают нужный вариант интерфейса. Один класс реализации может вместить определения нескольких интерфейсов, если изменения состоят только в добавлении или удалении предоставляемых методов. Изменение семантики существующих методов требует нового определения класса (обычно через адаптер или извлеченный суперкласс).

Сокет-ориентированные протоколы должны содержать идентификатор версии. Это, безусловно, требует, чтобы принимающие приложения были обновлены раньше, чем приложения, посылающие запросы. Более того, предполагается, что первые поддерживают несколько версий протокола. При наличии других вызывающих сторон у принимающих приложений это помогает ослабить взаимозависимость систем. Развертывание новой версии кода на принимающих эти вызовы серверах может быть осуществлено как их собственное «рекурсивное развертывание на лету».

Если поддерживать несколько версий протокола нецелесообразно, можно определить набор сервисных пулов в распределителе нагрузки на разных портах. В этом случае старая версия вызывающего приложения будет использовать номер порта, проецируемый на исходную версию принимающего приложения, в то время как новая версия будет пользоваться портом, проецируемым на новую версию.

До сих пор большинство конфликтов, причем представляющих наибольшие трудности, возникало вокруг базы данных. Изменения логической структуры редко имеют восходящую совместимость и никогда не являются случайными. Тем не менее даже эти изменения можно разбить на фазы. Во время фазы расширения добавляются таблицы и столбцы. Любые столбцы, в которых в итоге будет значение NOT NULL, добавляются как допускающие это значение, так как старая версия не знает, каким образом их следует заполнять. Позднее, в фазе очистки добавляются ограничители. Это касается и правил ссылочной целостности. Их невозможно добавить в фазе расширения, так как старая версия немедленно нарушит соотношение.

В итоге даже если старая версия приложения не прекратит свою работу из-за новых таблиц и столбцов, она все равно не сможет сделать для них ничего полезного. А новая версия, скорее всего, ожидает наличия здесь неких значимых данных. В нее может входить даже сценарий перемещения, заполняющий новые таблицы и столбцы данными из старой версии. Приложение для создания строк новой базы с большой вероятностью пользуется инструкцией `INSERT`¹. Для передачи данных в новые таблицы и столбцы можно пользоваться таким средством, как триггеры. При условии, что новая структура допускает заполнение существующими данными и что в сценарии переноса есть пример этого действия, триггер может выполнять эту операцию построчно. Таким образом, старая версия приложения будет создавать данные, которыми сможет пользоваться новая версия. В конечном счете новая версия тоже начнет создавать данные. Но она будет пользоваться для этого новой структурой; и если не предпринять никаких мер, старая версия увидит эти данные как поврежденные или неполные. Триггеры могут сформировать мост и в этом направлении, заполняя старую структуру на основе новых данных.

Разумеется, чтобы к этому подготовиться, любые SQL-команды `INSERT` или `UPDATE` должны четко задавать столбцы и значения. Команда `SELECT *` также вряд ли будет вам полезна в данном случае. На помощь снова придут инструменты ORM. Они механически генерируют SQL-команды, включающие в себя конкретные столбцы, которые следует выделить, вставить или обновить. Любые вспомогательные запросы, например интеллектуальный анализ данных или отчеты, должны четко задавать интересующие их столбцы. Это легко можно организовать, если делать постоянно с самого начала. Попытка заменить каждую команду `SELECT *`, чтобы выпустить новую версию, вряд ли будет хорошо воспринята.

Выгрузка

После подготовки, проведенной в фазе расширения, реальная выгрузка нового программного обеспечения на серверы приложений становится тривиальным мероприятием. Она может занять от нескольких часов до нескольких дней в зависимости от того, насколько аккуратно вы хотите к ней подойти. Например, можно попробовать дать паре серверов поработать с новой базой кода день или два. (Пока используются обе версии, имеет смысл создать два сервисных пула в распределителях нагрузки, чтобы запросы или переключения сеансов попадали в ту же группу серверов, что и оригинальный сеанс.)

Если в процессе развертывания потребуется остановить системы, на вас будет давить необходимость закончить работу в минимально возможный срок. Как только это будет сделано, у вас останется достаточно времени для аккуратной остановки

¹ Если приложение для добавления записей в базу данных пользуется хранимыми процедурами, их тоже следует обновить на этом этапе, чтобы обеспечить заполнение данными, иницируемое старой версией приложения.

системы. Следует сформировать привычку выполнять чистую перезагрузку, избавляя пользователей от риска столкнуться с внезапной остановкой системы¹ (см. раздел 14.3).

Очистка

После установки новой версии приходит время уборки. В эту процедуру входит удаление активаторов перенесения программ и дополнительных сервисных пулов. Можно удалить и все более не используемые столбцы и таблицы. А также старые версии статических файлов.

К этому моменту все серверы приложений уже используют новую версию кода. Теперь пора добавить тем столбцам, которым это нужно, атрибут NOT NULL, а также проверку ссылочной целостности (хотя добавленные к базе данных ограничители могут стать причиной больших проблем в слое ORM). Также нужно удалить все более не используемые столбцы и таблицы.

18.5. Заключение

Изменение является определяющей характеристикой программного обеспечения. Это изменение, точнее адаптация, начинается с момента выпуска основной версии. Это начало настоящей жизни программы; до этого система лишь созревает. После этого система либо развивается с течением времени, приспосабливаясь к меняющемуся окружению, либо постепенно деградирует, пока затраты на ее эксплуатацию не превысят приносимую ею прибыль, после чего система умирает.

¹ Главным образом, это потерянное состояние сеанса. Я никогда не видел идеально выполненного аварийного переключения сеансов.

Список литературы

- Kent Beck and Martin Fowler. Planning Extreme Programming. Addison-Wesley, Reading, MA, 2001.
- Kent Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley Longman, Reading, MA, 2000.
- James R. Chiles. Inviting Disaster: Lessons From the Edge of Technology. Harper Business, New York, NY, USA, 2001.
- Mike Clark. Pragmatic Project Automation. How to Build, Deploy, and Monitor Java Applications. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2004.
- Alistair Cockburn. Agile Software Development. Addison-Wesley Longman, Reading, MA, 2001.
- Mark Denne and Jane Cleland-Huang. Software by Numbers: Low-Risk, High-Return Development. Prentice Hall, Englewood Cliffs, NJ, 2003.
- Tom DeMarco. Why Does Software Cost So Much? Dorset House, New York, NY, USA, 1995.
- Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, MA, 1999.
- Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Longman, Reading, MA, 2003.
- Martin Fowler. Analysis Patterns: Reusable Object Models. Addison-Wesley Longman, Reading, MA, 1996.
- Justin Gehtland, Ben Galbraith, and Dion Almaer. Pragmatic Ajax: A Web 2.0 Primer. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2006.

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- Eliyahu Goldratt. *The Goal*. North River Press, Great Barrington, MA, Third, 2004.
- Charles Kozierok. *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*. No Starch Press, San Francisco, CA, 2005.
- Domenico Lepore and Oded Cohen. *Deming and Goldratt: The Theory of Constraints and the System of Profound Knowledge*. North River Press, Great Barrington, MA, 1999.
- Barbara Liskov and J. Wing. *Family Values: A Behavioral Notion Of Subtyping*. citeseer.ist.psu.edu/liskov94family.html. [MIT/LCS/TR-562b]:47, 1993.
- Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading, MA, Second, 2000.
- Geoffrey A. Moore. *Crossing the Chasm*. Harper Business, New York, NY, USA, 1991.
- Donald A. Norman. *The Design of Everyday Things*. Doubleday, New York, NY, USA, 1988.
- Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley, Reading, MA, 2003.
- Mary Poppendieck and Tom Poppendieck. *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley, Reading, MA, 2006.
- Henry Petroski. *The Evolution of Useful Things*. Alfred A. Knopf, Inc, New York, NY, 1992.
- Chet Richards. *Certain To Win*. Xlibris Corporation, Philadelphia, PA, 2004.
- Peter M. Senge. *The Fifth Discipline: The Art & Practice of the Learning Organization*. Doubleday, New York, NY, USA, 1994.
- Michael Shermer. *Why People Believe Weird Things*. W.H. Freeman and Company, New York, NY, 1997.
- W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, MA, 1993.
- John Vlissides, James O. Coplien, and Norman L. Kerth. *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, MA, 1996.
- Craig Wisneski, Hiroshi Ishii, Andrew Dahley, Matt Gorbet, Scott Brave, Brygg Ullmer, and Paul Yarin. *Ambient Displays: Turning Architectural Space into an Interface between People and Digital Information*. *Lecture Notes in Computer Science*. 1370:22, 1998.

Майкл Нейгард
**Release it! Проектирование и дизайн ПО для тех,
кому не всё равно**

Перевела на русский И. Рузмайкина

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Н. Римицан</i>
Литературный редактор	<i>А. Жданов</i>
Художник	<i>В. Шимкевич</i>
Корректоры	<i>С. Беляева, Н. Викторова</i>
Верстка	<i>Л. Панич</i>

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 01.07.15. Формат 70×100/16. Усл. п. л. 25,800. Тираж 500. Заказ

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



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

-  Наложным платежом с оплатой при получении в ближайшем почтовом отделении.
-  С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
-  Электронными деньгами. Мы принимаем к оплате: Яндекс.Деньги, Webmoney и Kiwi-кошелек.
-  В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com)
- Можно оформить доставку заказа через почтоматы, (адреса почтоматов можно узнать на нашем сайте www.piter.com)

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.