

O'REILLY®



Site Reliability Engineering

надежность и безотказность как в Google

Бетси Бейер, Крис Джоунс,
Дженнифер Петофф,
Нейл Ричард Мёрфи

Делитеर

O'REILLY®



Site Reliability Engineering

НАДЕЖНОСТЬ И БЕЗОТКАЗНОСТЬ КАК В GOOGLE

Бетси Байер, Крис Джоунс,
Дженнифер Петофф,
Найл Ричард Мёрфи



**Бетси Бейер, Крис Джоунс, Дженнифер Петофф, Нейл Ричард
Мёрфи**

Site Reliability Engineering. Надежность и безотказность как в Google



2018

Переводчики *A. Ананич, Е. Зазноба, И. Пальти*

Технический редактор *Н. Гринчик*

Литературные редакторы *Н. Гринчик, Н. Рошина*

Художники *А. Барцевич, С. Заматевская*

Корректоры *О. Андриевич, Е. Павлович*

Верстка *Г. Блинов*

**Бетси Бейер, Крис Джоунс, Джениффер Петофф, Нейл
Ричард Мёрфи**

Site Reliability Engineering. Надежность и безотказность как
в Google. — СПб.: Питер, 2018.

ISBN 978-5-4461-0976-0

© [ООО Издательство "Питер"](#), 2018

*Все права защищены. Никакая часть данной книги не может
быть воспроизведена в какой бы то ни было форме без
письменного разрешения владельцев авторских прав.*

Предисловие Марка Берджеса

История Google — это история роста. Это одна из величайших историй успеха в компьютерной индустрии, ознаменовавшая переход к ИТ-ориентированному бизнесу. Google в числе первых определила, что на практике означает союз бизнеса и ИТ-технологий, а также популяризовала концепцию DevOps — сращивание процессов разработки продуктов и информационно-технологического обслуживания. Данную книгу написали люди, воплотившие этот переход в реальность.

Компания Google развивалась в те времена, когда пересматривалась традиционная роль системного администратора. Google поставила под вопрос саму идею системного администрирования, говоря: «Мы не можем позволить себе подчиняться традициям, нам нужно придумать что-то новое, и у нас нет времени ждать, пока остальные нас догонят». Во введении к книге *Principles of Network and System Administration* [Burgess, 1999] я утверждал, что системное администрирование — человеко-машинная технология¹. Некоторые критики резко отреагировали на эту идею, заявив, что «мы еще не достигли уровня развития, позволяющего называть это технологией». Уже тогда у меня было чувство, что вся эта отрасль зашла в тупик, заперта в заколдованным круге своей культуры, и я не видел пути вперед.

Компания Google сделала решительный шаг, разорвав этот круг и превратив грядущее в настоящее. Она изменила взгляд на саму роль системного администратора. Теперь название этой должности звучит так: «инженер по обеспечению надежности информационных систем» (Site Reliability Engineer, SRE). Некоторые мои друзья были в числе первых инженеров нового поколения; они формализовали эту роль с помощью

специального ПО и средств автоматизации. Поначалу они не распространялись о том, чем занимаются, а SRE-процессы, происходившие внутри Google и вовне, сильно отличались друг от друга: опыт Google был уникален. Но со временем методы работы Google становились известны остальным. Эта книга — шаг к тому, чтобы приподнять завесу над образом мышления SR-инженеров.

Из книги вы узнаете не только о том, как Google построила свою легендарную инфраструктуру, но и о том, как эта компания развивалась, училась и постепенно меняла свой взгляд на инструменты и технологии. Если мы будем открыты новому, то тоже сможем с легкостью справляться с самыми нетривиальными вызовами. Клановость IT-культуры зачастую тормозит всю отрасль. Если Google смогла преодолеть эту инерцию, то сможем и мы.

Эта книга представляет собой сборник статей, написанных специалистами из одной компании, разделяющими в общем одну точку зрения. Характерно, что все изложение строится вокруг единой цели компании. Некоторые темы и «персонажи» (программные системы) фигурируют в нескольких главах. При этом мы стараемся рассмотреть тему с разных сторон и в соответствии с различными интересами. Статьи (из них и состоит книга) похожи на личные блоги, они написаны в разных стилях и отражают взгляды разных специалистов, каждый из которых обладает своим набором навыков. Они написаны смело и честно, что необычно для большинства книг нашей отрасли. В некоторых статьях говорится «никогда не делайте так, всегда делайте так», другие же более абстрактны и философичны, что отражает разнообразие позиций их авторов в рамках IT-культуры, и это также играет свою роль в истории. Мы, в свою очередь, читаем их как скромные сторонние наблюдатели, которые не преодолевали этот путь и не

располагают всей информацией о множестве противоречивых проблем и компромиссов. Именно вопросы таких наблюдателей — то, что прежде всего будет вынесено из этой книги. Почему они не сделали X? Что было бы, сделай они Y? Как мы посмотрим на эту книгу многие годы спустя? Сравнивая наши собственные идеи с информацией, предоставленной в книге, мы можем оценить собственные мысли и опыт.

Самое впечатляющее в этой книге — ее жизненность. Сегодня мы часто слышим безапелляционное: «Просто покажите мне код». Вокруг открытого исходного кода выросла культура «не задавай вопросов», где сообщество и принцип совместной разработки берут верх над профессионализмом². Google — это компания, которая рискнула пересмотреть самые устои. Она нанимает самых талантливых людей, многие из которых имеют степень PhD. Инструментарий — лишь одно звено в цепи наряду с ПО, сотрудниками и данными. В книге нет универсальных решений задач, но в этом и вся суть. Подобные истории ценятся гораздо больше, чем конечный результат в виде кода или готовых программных продуктов. Реализация эфемерна, а задокументированное обоснование бесценно. Мы редко имеем доступ к подобным откровениям.

Эта книга — история успеха одной компании. Многие пересекающиеся истории показывают нам, что рост — это нечто гораздо большее, чем механическое увеличение классической вычислительной архитектуры. Это масштабирование бизнес-процессов, а не просто наращивание парка техники. Уже сам по себе этот урок стоит своего места на бумаге.

Мы не хотим втягиваться в самокритический обзор мира ИТ. Собственно, миру ИТ вообще свойственно повторять и заново изобретать решения. Многие годы проводилась лишь одна

конференция, на которой обсуждалась инфраструктура ИТ, — USENIX LISA, и несколько конференций по операционным системам. И даже сейчас, когда ситуация значительно изменилась, эта книга остается бесценным даром: в ней детально задокументирован путь Google в переломную эпоху. Описанное не нужно копировать (хотя оно и может быть образцом для подражания) — книга должна вдохновить нас сделать свой следующий шаг. На ее страницах вы увидите множество правдивых примеров, демонстрирующих как превосходство, так и скромность. Вы увидите истории о надежде, страхе, успехах и провалах. Я приветствую мужество авторов и редакторов, позволивших рассказывать истории настолько откровенно, чтобы все, кто не является частью этого процесса, тоже смогли извлечь пользу из уроков, полученных в стенах Google.

*Марк Берджес, автор книги *In Search of Certainty* Осло, март 2016*

[1](#) В исходном тексте книги *human-computer engineering*. — Примеч. пер.

[2](#) Это слишком категорично и однобоко. Инициатива Open Source, сообщества разработчиков, качество программного продукта — связь между ними, конечно, есть, но она не столь простая и однозначная. Закрытые «фирменные» разработки тоже, к сожалению, не всегда гарантируют качество. — Примеч. пер.

Предисловие авторов

Программное обеспечение (ПО) во многом подобно ребенку: для его *появления* на свет необходимо пройти через трудности и боль, но *после* его рождения усилий потребуется еще больше. Сегодня в обсуждении разработки ПО первому периоду уделяется намного больше внимания, несмотря на то что от 40 до 90 % затрат приходится на второй — *после* его выпуска³. Распространенное мнение о том, что развернутое и работающее ПО «стабильно» и, как результат, не требует пристального внимания разработчиков, неверно. Следовательно, если разработка программного обеспечения обычно сосредоточена на проектировании и построении программных систем, должна существовать и другая дисциплина, предметом которой будет *весь* жизненный цикл программных продуктов: их создание, развертывание, функционирование, обновление и в свое время — безболезненный вывод из эксплуатации. Эта дисциплина обращается — и должна обращаться — к широкому спектру навыков, но цель их применения иная, чем у других технических дисциплин. Такую дисциплину в Google называют *обеспечением надежности информационных систем* (Site Reliability Engineering, SRE).

Что же включает в себя понятие «обеспечение надежности информационных систем» (SRE)? Мы признаем, что название недостаточно хорошо отражает суть работы, — практически у каждого инженера, обеспечивающего надежность в Google, периодически спрашивают, что это такое и чем он занимается.

Объясняя термин, стоит сказать, что специалисты SRE прежде всего *инженеры*. Мы используем принципы информатики и инженерии, чтобы проектировать и

разрабатывать компьютерные системы, как правило, крупные и распределенные. Иногда нам приходится писать ПО для таких систем в тесном контакте с разработчиками программных продуктов. Иногда мы должны реализовать все служебные модули данной системы, вроде резервного копирования или балансировки нагрузки, при этом в идеале эти фрагменты должны быть пригодны для использования и в других системах. А иногда наша задача — выяснить, как применять существующие решения для решения новых задач.

Далее, мы заботимся об обеспечении *надежности* системы. Бен Трейнор Слосс, вице-президент Google и автор термина SRE, утверждает, что надежность — это наиболее важная характеристика любого продукта: система не будет успешной, если с ней невозможно работать! И поскольку надежность⁴ настолько важна, специалисты SRE трудятся над тем, чтобы найти способы улучшить дизайн и функционирование систем в попытке сделать их более масштабируемыми, надежными и эффективными. Однако мы работаем в этом направлении только до определенного момента: когда система считается «достаточно надежной», мы переносим свое внимание на добавление новых функций или создание новых продуктов⁵.

Наконец, SR-инженеры занимаются поддержанием работы сервисов, построенных на базе наших распределенных вычислительных систем, независимо от того, являются они хранилищами планетарных масштабов, сервисом электронной почты для сотен миллионов пользователей или поисковиком, с которого и началась история Google. Слово «сайт» в названии поначалу говорило, что мы поддерживали работу сайта **google.com**, однако сейчас у нас запущено гораздо больше сервисов, многие из которых не являются сайтами, — начиная с внутренней инфраструктуры вроде сервиса Bigtable и

заканчивая продуктами для внешних разработчиков наподобие Google Cloud Platform.

Хотя мы преподносим SRE как универсальную дисциплину, ее появление именно в быстрорастущем мире веб-сервисов вполне закономерно. Возможно также, что ее зарождению способствовали и особенности нашей инфраструктуры. Неудивительно, что из всех характеристик ПО, которым мы можем уделить внимание после развертывания, надежность является одной из основных⁶. Веб-сервисы — как раз та сфера, где наиболее актуален наш подход, поскольку процесс обновления и совершенствования серверного ПО относительно самодостаточен, а процесс управления этими изменениями тесно связан со сбоями всех видов.

Несмотря на то что эта дисциплина появилась в Google и в веб-сообществе в целом, мы считаем, что наш опыт применим и в других компаниях и организациях. В этой книге мы пытаемся объяснить свой подход как для того, чтобы другие организации смогли использовать наши наработки, так и для того, чтобы лучше определить роль и значение SRE. Для этого мы структурировали книгу таким образом, чтобы общие принципы и более конкретизированные практики были по возможности отделены друг от друга. Кроме того, там, где это уместно, при рассмотрении тех или иных тем мы приводим примеры из практики Google. Мы верим, что читатель встретит это с пониманием и сможет сделать полезные для себя выводы.

Мы также предоставили вспомогательный материал — описание производственной среды Google и соответствия между нашим внутренним ПО и общедоступным, что должно помочь вам воспринимать весь изложенный материал в понятном вам контексте и применять свои знания на практике.

Наконец, нужно сказать, что создание ПО и разработка систем с учетом повышенной надежности — однозначное

благо. Однако мы понимаем, что в небольших организациях могут задаваться непростым вопросом: как им лучше применить знания, полученные из этой книги? Это похоже на вопрос безопасности — чем раньше вы им займитесь, тем лучше. Даже если ваша небольшая организация имеет множество насущных проблем и ваше ПО отличается от выбранного в Google, вам все равно стоит заранее позаботиться о команде SRE, поскольку в этом случае последующее расширение структуры обойдется дешевле, чем построение ее с нуля. В части IV приведены практические рекомендации для обучения, коммуникации и проведения совещаний, опробованные у нас. Многие из них вы могли бы применить и для своей компании.

В компаниях среднего размера, скорее всего, уже имеется сотрудник, выполняющий функции SR-инженера (хотя, возможно, его должность называется как-то иначе). Поэтому путь к улучшению надежности ПО в такой организации — официально обозначить эту работу или найти людей, уже выполняющих ее, и побуждать их к ее дальнейшему выполнению, вознаграждая соответствующим образом. Эти люди стоят на границе разных взглядов на мир, подобно Ньютону, которого иногда называют не первым в мире физиком, а последним алхимиком.

И раз уж мы заговорили об истории, давайте подумаем, кто мог бы быть первым SR-инженером.

Мы думаем, что Маргарет Гамильтон, работавшая над программой «Аполлон» во время учебы в MIT, первой продемонстрировала все основные черты SR-инженера⁷. По ее словам, «частью технической культуры является обучение всему и отовсюду, включая вещи, от которых меньше всего ждешь возможности чему-то научиться».

Однажды она взяла на работу свою маленькую дочь Лорен. В то время ее коллеги моделировали сценарии миссии на гибридном компьютере. Как и другие маленькие дети, Лорен принялась исследовать новое место и спровоцировала крах миссии, введя в DSKY (интерфейс компьютера «Аполлона», сокращение от display and keyboard) команды, не предусмотренные в текущем режиме. Таким образом, разработчики узнали, что произойдет, если реальный астронавт во время реальной миссии выполнит программу предстартовой подготовки P01. (Случайный запуск программы P01 во время реальной миссии стал бы серьезной проблемой, поскольку это привело бы к удалению навигационных данных и компьютер без них не смог бы управлять кораблем.)

Прислушавшись к своему чутью SR-инженера, Маргарет отправила запрос на изменение программы — добавление специального проверяющего кода на случай, если астронавт во время полета нечаянно выберет программу P01. Но эту инициативу верхушка NASA посчитала ненужной — ведь такого, конечно же, просто не может быть!⁸ Поэтому вместо того, чтобы добавить код, Маргарет обновила документацию для миссии, поместив там предупреждение вида «Не запускайте программу P01 во время полета». (Судя по всему, это позабавило многих участников проекта, поскольку им много раз говорили, что астронавты имеют безукоризненную подготовку и поэтому не ошибаются.)

Эти меры безопасности казались ненужными только до следующей миссии — на «Аполлоне-8», — которая стартовала спустя всего несколько дней после обновления спецификации. Экипаж состоял из трех человек: Джима Ловелла, Уильяма Андерса и Фрэнка Бормана. Во время прохождения среднего участка траектории, на четвертый день полета, Джим Ловелл нечаянно выбрал программу P01 — по случайному совпадению

это произошло на Рождество, — что создало кучу проблем всем участникам. Проблемы были более чем серьезными, ведь если бы решение не нашлось, то астронавты не смогли бы вернуться домой. К счастью, в обновленной документации такая ситуация была описана, и это помогло разобраться, как достаточно быстро загрузить необходимые данные и восстановить управление миссией.

По словам Маргарет, «даже досконального понимания того, как работают системы, бывает недостаточно для предотвращения человеческих ошибок», и вскоре запрос на добавление механизма обнаружения ошибки и восстановления работоспособности на случай запуска программы предстартовой подготовки P01 был одобрен.

Хотя инцидент с «Аполлоном-8» произошел полвека назад, этот опыт весьма поучителен для современных инженеров и продолжит оставаться таковым в будущем. Следите ли вы за системами, работаете в группе или создаете свою компанию, пожалуйста, держите в голове принципы SRE: доскональность и самоотдача, убежденность в важности подготовки и документирования, предусмотрительность в том, что может пойти не так, вкупе со стремлением это предотвратить. Добро пожаловать в нашу развивающуюся профессию!

Как читать эту книгу

Эта книга представляет собой сборник очерков, написанных сотрудниками службы по обеспечению надежности Google. Она больше похожа на сборник докладов конференции, чем на обычную книгу, написанную одним автором или группой авторов. Каждая глава должна рассматриваться как часть целого, но пользу принесет также и чтение отдельных

интересующих вас глав. (Если существуют какие-то другие статьи, которые дополняют текст, мы оставим на них ссылку, чтобы вы могли с ними ознакомиться.)

Вам не обязательно читать книгу в определенном порядке, однако мы советуем начать с глав 2 и 3 (часть I «Введение»), где описывается производственная среда Google и подчеркиваются подходы SRE к рискам. (Риск во многом является ключевой особенностью нашей профессии.) Вы можете прочесть книгу от корки до корки, это тоже будет полезно; главы в ней сгруппированы по темам: «Принципы» (часть II), «Практики» (часть III) и «Управление» (часть IV). Каждая из них начинается небольшим вступлением: из него вы узнаете, о чем рассказывается в этой части, а также найдете ссылки на другие статьи, опубликованные SR-инженерами Google (там некоторые темы рассматриваются более подробно). Помимо этого, сопутствующий книге сайт, <https://g.co/SREBook>, содержит немало полезных ресурсов.

Мы надеемся, что эта книга будет для вас интересной и полезной настолько, насколько ее создание было интересным и полезным для нас.

Редакторы

Условные обозначения

В книге используются следующие условные обозначения.

Курсив

Применяется для обозначения новых понятий и терминов, которые авторы хотят особо выделить.

Шрифт без засечек

Используется для обозначения адресов электронной почты и URL-адресов.

Моноширинный шрифт

Используется для текста (листингов) программ, а также внутри абзацев для выделения элементов программ: имен переменных или функций, названий баз данных, типов данных, имен переменных среды, инструкций и ключевых слов, имен файлов и каталогов.

Курсивный моноширинный шрифт

Показывает в тексте те элементы, которые должны быть заменены значениями, подставляемыми пользователем или определяемыми контекстом.



Этот знак отмечает совет или предложение.



Этот знак указывает на примечание общего характера.



Этот знак отмечает предупреждение.

Использование примеров кода

Сопутствующий материал вы можете найти по адресу <https://g.co/SREBook>.

Эта книга призвана помочь вам в работе. Примеры кода из нее вы можете использовать в своих программах и документации. Если объем кода несущественный, связываться с нами для получения разрешения не нужно. Например, для написания программы, использующей несколько фрагментов кода из этой книги, разрешения не требуется. А вот для продажи или распространения компакт-диска с примерами из книг издательства O'Reilly нужно получить разрешение. Ответы на вопросы с использованием цитат из этой книги и примеров кода разрешения не требуют. Но для включения объемных примеров кода из этой книги в документацию по вашему программному продукту разрешение понадобится.

Мы приветствуем указание ссылки на источник, но не делаем это обязательным требованием. Такая ссылка обычно включает название книги, имя автора, название издательства и ISBN. Например: Site Reliability Engineering, edited by Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy (O'Reilly). Copyright 2016 Google, Inc., 978-1-491-92912-4.

Если вам покажется, что использование кода примеров выходит за рамки оговоренных выше условий и разрешений, свяжитесь с нами по адресу permissions@oreilly.com.

³ Само по себе наличие такого большого разрыва может кое-что сказать нам о программной инженерии как о дисциплине. Для получения более подробной информации вы можете обратиться к [Glass, 2002].

⁴ В данном контексте надежность — это «вероятность того, что система будет выполнять требуемые функции без сбоев при заданных условиях в заданный период времени», согласно определению, приведенному в [O'Connor, 2012].

⁵ Как правило, мы работаем над сайтами и другими подобными сервисами; мы не рассматриваем вопрос надежности ПО для атомных электростанций, авиационных

систем, медицинского оборудования или иных систем, для которых критически важна безопасность. Однако в главе 33 мы сравниваем свой подход с подходами, используемыми в других областях.

[6](#) В этом мы отличаемся от направления DevOps: хотя мы рассматриваем инфраструктуру как код, нашей главной целью является надежность. Вдобавок мы стремимся к тому, чтобы избавляться от необходимости в отдельной службе операционной поддержки — для получения более подробной информации прочтите главу 7.

[7](#) Помимо этого, она внесла значительный вклад в популяризацию термина software engineering — «программная инженерия, технология разработки ПО».

[8](#) Были и объективные причины минимизировать объем и сложность программ. Во-первых, ресурсы бортовых компьютеров, созданных для программы «Аполлон», были весьма ограничены (см., например, <https://history.nasa.gov/computers/Ch2-5.html>), причем на первых порах программа поглощала более половины всех производимых в США микросхем. Во-вторых, любая модификация кода требовала бы длительного повторного тестирования (эта проблема в книге также рассматривается).
— Примеч. пер.

Благодарности

Этой книги не было бы, если бы авторы и технические писатели не прикладывали все усилия для ее создания. Мы также хотим поблагодарить следующих научных редакторов за весьма ценные замечания и отзывы: Алекса Мэйти, Дермота Даффи, Джей Си ван Винкеля, Джона Т. Риза, Майкла О'Райли, Стива Карстенсена и Тодда Андервуда. Бен Латч и Бен Трейнор Слосс стали спонсорами этой книги в самой компании Google; их вера в этот проект и в то, что мы должны поделиться своими знаниями о своих крупномасштабных сервисах, стала залогом успешного появления данного издания.

Мы хотели бы отдельно поблагодарить Рика Фэрроу, редактора журнала *;login;*, за его участие в предпубликациях в конференции USENIX.

Несмотря на то что авторы для каждой главы указаны отдельно, мы хотели бы перечислить всех, кто также внес вклад в создание каждой главы, предоставив полезную информацию, вступив в дискуссию или сделав обзор.

- Глава 3: Эйб Рэй, Бен Трейнор Слосс, Брайан Столер, Дейв О'Коннор, Дэвид Бесбрис, Джилл Алвидрес, Майк Кертис, Нэнси Ченг, Тэмми Кэпистрент, Том Лимончелли.
- Глава 5: Коди Смит, Джордж Сэдлер, Лоренс Берланд, Марк Алвидрес, Патрик Сталберг, Питер Дафф, Пим ван Пелт, Райан Андерсон, Сабрина Фармер, Сет Хеттич.
- Глава 6: Майк Кертис, Джейми Уилкинсон, Сет Хеттич.
- Глава 8: Дэвид Шнур, Джей Ти Голдстоун, Марк Алвидрес, Маркус Лара-Рейнхольд, Ноа Максвелл, Питер Динджес,

Сумитран Рагунатан, Ютонг Чо.

- Глава 9: Райан Андерсон.
- Глава 10: Джулс Андерсон, Макс Люббе, Микель Макдэниел, Рауль Вера, Сет Хеттич.
- Глава 12: Чарльз Стивен Ганн, Джон Хеддитч, Питер Наттал, Роб Иващук, Сэм Гринфилд.
- Глава 13: Елена Ортел, Крипа Кришнан, Серджио Сальви, Тим Крейг.
- Глава 14: Эми Жоу, Карла Гейссер, Грэйн Шерин, Хилдо Бирсма, Елена Ортел, Перри Лорье, Рун Кристиан Викен.
- Глава 15: Дэн Ву, Хизер Шерман, Джаред Брик, Майк Луэр, Степан Давидович, Тим Крейг.
- Глава 16: Эндрю Стриблхилл, Ричард Вудбери.
- Глава 17: Исаак Клеренсиа, Марк Алвидрес.
- Глава 18: Ульрик Лонгийир.
- Глава 19: Дебашиш Чаттерджи, Перри Лорье.
- Главы 20 и 21: Адам Флетчер, Кристоф Фистерер, Лукаш Йезек, Маниот Пава, Миша Ризер, Ноа Фидел, Павел Херманн, Павел Зузельски, Перри Лорье, Ральф Вильденхьюс, Тюдор-Иона Саломи, Витольд Бариллик.
- Глава 22: Майк Кертис, Райан Андерсон.
- Глава 23: Анант Шринивас, Майк Берроуз.

- Глава 24: Бен Фрид, Дерек Джексон, Гейб Краббе, Лаура Нолан, Сет Хеттич.
- Глава 25: Абдулрахман Салем, Алекс Перри, Арнар Мар Храфнкельссон, Дитер Пирси, Дилан Керли, Эйвинд Эклунд, Эрик Вич, Грэхэм Паултер, Ингвар Мэттсон, Джон Луни, Кен Грант, Мишель Даффи, Майк Хохберг, Уилл Робинсон.
- Глава 26: Кори Викри, Дэн Арделеан, Дисней Луангсисонгхам, Гордон Приореши, Кристина Беннет, Лианг Лин, Майкл Келли, Сергей Иванюк.
- Глава 27: Вивек Рай.
- Глава 28: Мелиssa Бинде, Перри Лорье, Престон Ешиока.
- Глава 29: Бен Латч, Карла Гейссер, Dzevad Trumic, John Turek, Мэтт Браун.
- Глава 30: Чарльз Стивен Ганн, Крис Хейсер, Макс Люббе, Сэм Гринфилд.
- Глава 31: Алекс Келенбек, Джероми Карье, Джоэл Бекер, Совмия Виджайярагаван, Тревор Мэттсон-Гамильтон.
- Глава 32: Сет Хеттич.
- Глава 33: Эдриан Хилтон, Брэд Кратеквил, Чарльз Баллоу, Дэн Шеридан, Эдди Кеннеди, Эрик Гросс, Гас Хартманн, Джексон Стоун, Джейф Стивенсон, Джон Ли, Кевин Греер, Мэтт Тойя, Майкл Хейни, Майк Доэрти, Питер Даль, Рон Хейби.

Мы также благодарны всем, кто поделился полезной информацией, сделал хороший обзор, согласился на интервью,

предоставил экспертную оценку либо материалы или как-либо еще повлиял на эту работу. Это: Эйб Хассан, Адам Рогойски, Алекс Хидалго, Амайа Букер, Эндрю Файкс, Эндрю Херст, Эриел Гоу, Эшли Рентц, Айман Хурье, Барклей Осборн, Бен Эпплтон, Бен Лав, Бен Уинслоу, Бернард Бек, Билл Дюэйн, Билл Петри, Блэр Заяц, Боб Грубер, Брайан Густафсон, Брюс Мёрфи, Бак Клей, Седрик Селье, Чихо Сайто, Крис Карлон, Кристофер Хан, Крис Кеннели, Крис Тейлор, Сиера Камахель-Санфрателло, Колин Филиппс, Колм Бакли, Крэйг Патерсон, Даниель Эйзенбад, Дэниэл В. Клейн, Дэниел Спунхауэр, Дэн Уотсон, Дэйв Филиппс, Дэвид Хиксон, Дина Бетсер, Дорон Мейер, Дмитрий Федорук, Эрик Гроссе, Эрик Шрок, Филип Жыжневски, Фрэнсис Тэнг, Гэри Арнесон, Джорджина Уилкокс, Гретта Бартельс, Густаво Франко, Харальд Вагенер, Хилфден Гоген, Хьюго Сантос, Хирам Райт, Иэн Гулливер, Якуб Турски, Джеймс Чиверс, Джеймс О'Кейн, Джеймс Янгмен, Ян Монш, Джейсон Паркер-Берлингхэм, Джейсон Петсад, Джеффри Макнил, Джефф Дин, Джефф Пек, Дженифер Мейс, Джерри Кен, Джесс Фрейм, Джон Брэйди, Джон Гундерман, Джон Кочмар, Джон Тобин, Жордин Буханан, Йозеф Биронас, Джулио Мерино, Джулиус Пленц, Кейт Уорд, Кэти Полицци, Катрина Состек, Кенн Хамм, Кирк Рассел, Крипа Кришнан, Ларри Гринфилд, Леа Оливейра, Лука Читтадини, Лукас Переира, Магнус Рингман, Махеш Палекар, Марко Паганини, Марио Бонилла, Мэттью Миллс, Мэттью Монро, Мэтт Д. Браун, Мэтт Прауд, Макс Солтонстолл, Михал Ящик, Михай Бивол, Миша Брукман, Оливье Онсальди, Патрик Бернье, Пьер Палатин, Роб Шэнли, Роберт ван Гент, Рори Уорд, Руй Жанг-Шен, Салим Вирджи, Санджей Гемават, Сара Коти, Шон Дорвард, Шон Куинлан, Шон Секрист, Шари Трамбо-Макхенри, Шон Моррисси, Шан-Так Лион, Стэн Йедрус, Стефано Латтарини, Стивен Ширрипа, Таня Райли, Терри Болт, Тим Чаплин, Тоби

Вейнгартнер, Том Блэк, Ури Мейри, Виктор Террон, Влад Грама, Уэс Хартлайн и Золтан Эгед.

Мы благодарны за вдумчивые и подробные отзывы, которые получили от сторонних рецензентов: Эндрю Фонга, Бьорна Рабенштейна, Фарльза Бодрера, Дэвида Блэнк-Эдельмана, Фресси Эконому, Джеймса Мейкла, Джоша Райдера, Марка Берджесса и Расса Элбери.

Кроме того, мы хотели бы отдельно поблагодарить Циан Синнотт, нашу коллегу, которая покинула Google до того, как этот проект был закончен, но значительно повлияла на него, и Маргарет Гамильтон – за то, что она любезно позволила нам рассказать ее историю в предисловии. Помимо этого, мы хотели бы поблагодарить Шилайю Нукалу за то, что смогли воспользоваться услугами ее технических писателей, а также за ее поддержку.

Редакторы также хотели бы выразить свою благодарность.

- Бетси Бейер: бабушке (моему личному герою) — за бесконечные телефонные напутствия и попкорн, а также Рибу — за спортивные брюки, которые согревали меня поздними вечерами. И, конечно, хотелось бы не забыть всех SR-инженеров, внесших вклад в написание этой книги.
- Крис Джоунс: Мишель — за то, что уберегла меня от необдуманных поступков в открытом море, и за ее невероятную способность находить яблоки в неожиданных местах, а также всем, кто обучал меня инженерному делу все эти годы.
- Дженнифер Петофф: моему мужу Скотту — за неимоверную поддержку на протяжении всех двух лет, когда писалась эта книга, и за то, что у редакторов было достаточно сахара на нашем «острове десертов».

- Нейл Мёрфи: Леану, Оисину и Фиакре — за то, что столько лет терпели своего необычно много ноющего отца и мужа. Хочу также поблагодарить Дермота за предложение о переводе.

Часть I. Введение

В этой части приводится общая информация о том, что такое SRE и почему эта дисциплина отличается от более общепринятых практик в IT-индустрии.

Бен Трейнор Слосс, вице-президент, курирующий службу эксплуатации (operations, ops) в Google (и автор понятия *Site Reliability Engineering*), рассказывает о том, как он понимает термин SRE, о принципах работы этой дисциплины, а также сравнивает ее с другими способами решения задач (глава 1).

В главе 2 мы расскажем о производственной среде и о «промышленном» (production) окружении Google, чтобы вы могли познакомиться с множеством новых понятий и систем, которые вам предстоит встретить по всей книге.

1. Вступление

Автор — Бенджамин Трейнор Слосс

Под редакцией Бетси Байер

Надеяться — это плохая стратегия.

Традиционная поговорка SRE

Считается, что информационные системы не запускают себя сами — и это действительно так. Как же в таком случае нужно запускать системы — особенно большие и сложные?

Подход системного администратора к управлению сервисами

Исторически сложилось так, что для запуска сложных информационных систем компании нанимали системных администраторов («сисадминов»).

Подобный подход предполагает построение сервиса путем сборки существующих программных компонентов и их настройки для согласованной работы. Затем администраторам поручается запустить сервис и реагировать на происходящие события и появляющиеся обновления по мере их поступления. С ростом сложности системы и объема трафика количество событий и обновлений тоже растет, и команда администраторов также увеличивается, чтобы успевать выполнять всю эту дополнительную работу. Поскольку роль администратора требует набора навыков, который заметно отличается от навыков разработчика, эти два направления разделяют на две команды: команду разработчиков

(development) и службу эксплуатации (operations, или просто ops).

Такая модель управления сервисами имеет ряд преимуществ. Для компаний, которые планируют запустить и обслуживать сервис, данный подход реализовать относительно легко. Поскольку эта парадигма существует довольно давно, есть много примеров, на которых можно чему-то научиться и которым можно подражать. Найти квалифицированных специалистов также не составляет труда. Множество существующих инструментов, программных компонентов (как готовых решений, так и заказных) и компаний-интеграторов помогут вам запустить такую систему, поэтому начинающим администраторам не придется изобретать велосипед и разрабатывать ее с нуля.

Однако подход с привлечением системных администраторов и разделением команд (dev и ops) имеет также несколько недостатков и подводных камней. Их можно разделить на две категории: явные и неявные издержки.

Неизбежные явные издержки достаточно велики. Функционирование сервиса, который требует ручных операций как при изменениях кода, так и при обслуживании событий, обходится все дороже по мере роста сложности сервиса и объема генерируемого им трафика, поскольку численность обслуживающей его команды также будет увеличиваться пропорционально трудоемкости.

Неявные издержки, сопровождающие разделение команд, не столь очевидны, но зачастую они гораздо дороже обходятся организации. Это проблемы, которые возникают из-за большой разницы в уровне знаний, наборе навыков и мотивации команд. Команды по-разному видят и описывают ситуацию; они делают разные предположения о рисках и возможностях реализации технических решений, о целевом

уровне стабильности продукта. Разделенность команд может приводить к утрате не только общей мотивации, но и единства целей, взаимодействия между ними и в конечном итоге взаимного доверия и уважения, что будет иметь весьма болезненные последствия.

Традиционные службы эксплуатации и их коллеги-разработчики часто конфликтуют, особенно из-за сроков выпуска продуктов. Разработчики хотят быстрее запустить новый функционал и увидеть, что пользователи его приняли. Команда эксплуатации же хочет убедиться, что сервис не откажет в самый неподходящий момент. Поскольку бо́льшая часть сбоев бывает спровоцирована какими-либо изменениями: новой конфигурацией, внедрением нового функционала или новым типом пользовательских запросов, — цели двух команд противоречат друг другу.

Обе команды понимают, что навязывать свои интересы друг другу неприемлемо («Мы хотим запускать все, что хотим и когда хотим, без промедления» против «Мы не хотим ничего менять в системе, если она работает»). И поскольку понятия, которыми они оперируют, и оценки рисков у обеих команд различны, зачастую начинается хорошо известная «окопная война», в которой каждый отстаивает свои интересы. Служба эксплуатации пытается оградить работающую систему от риска изменений, связанных с введением новых функций. Например, ревизия новой версии перед запуском может подразумевать детальные проверки по всем проблемам, которые вызывали сбои в прошлом, — список может оказаться огромным, и не все его элементы будут одинаково значимы. Команда разработчиков быстро находит ответ. Они выпускают меньше «новых версий» и больше разнообразных «обновлений». Они начинают дробить продукт, чтобы на детальные проверки попадал меньший объем нового функционала.

Подход Google к управлению сервисами: техника обеспечения надежности сайтов

Но такие конфликты при выпуске программного обеспечения не следует принимать как нечто неизбежное. В компании Google был выбран другой подход: в наши команды SRE предпочитают набирать разработчиков, которые будут выпускать продукты и создавать вспомогательные системы для тех задач и функций, которые бы иначе выполняли, причем часто вручную, системные администраторы.

Что такое обеспечение надежности информационных систем (SRE) и как такой подход появился в Google? Я могу дать простое объяснение: SRE — это то, что происходит, когда вы просите программиста спроектировать команду службы эксплуатации. Когда я пришел в Google в 2003 году и передо мной поставили задачу возглавить группу эксплуатации «промышленных» систем в составе семи инженеров, я умел только разрабатывать ПО. Поэтому я создал группу такой, какой мне хотелось бы ее видеть, если бы я сам был SР-инженером. И управлял я ею соответственно. Эта группа в итоге выросла в современную команду SRE компании Google, и по сей день следующей своим принципам, заложенным человеком, который всю сознательную жизнь был инженером-программистом.

Главное в подходе Google к управлению сервисами — принцип формирования SRE-команд. Всех сотрудников SRE можно разделить на две основные категории: 50–60 % SР-инженеров — это разработчики Google или, если быть более точным, люди, которые были наняты по стандартной процедуре найма разработчиков Google; остальные 40–50 % — это те, кто имеет практически полную квалификацию разработчика (например, 85–99 % требуемых навыков) и дополнительно владеет навыками, полезными для SRE, которые

редко встречаются у разработчиков. В данный момент мы чаще всего обращаем внимание на знание систем UNIX и работу с сетями (с первого по третий уровень модели OSI).

Всех сотрудников SRE объединяют их убеждения и стремление разрабатывать приложения для решения сложных задач. Внутри службы SRE мы тщательно отслеживаем развитие обеих категорий и на данный момент не видим разницы в производительности между их представителями. На самом деле различия в квалификации членов SRE-команд часто позволяет создавать продуманные, высококачественные системы, которые, очевидно, являются результатом синтеза разнообразных навыков.

Такой подход к найму SR-инженеров дает команду, которая:

- а) быстро начинает скучать при выполнении задач вручную и
- б) имеет набор навыков, необходимый для создания программ, заменяющих ручную работу, даже если решение окажется сложным. В итоге SR-инженеры в основном выполняют ту же работу, которой ранее занималась служба эксплуатации, но их знания и навыки позволяют разрабатывать и реализовывать автоматизированные решения, заменяющие человеческий труд, и именно такую задачу ставит перед ними компания.

Изначально определено, что приоритетная задача для SRE-команд — именно разработка. Без постоянных доработок и улучшений в системе трудоемкость ее эксплуатации будет возрастать, и командам понадобится все больше людей только для того, чтобы успевать за ее ростом. В итоге для традиционной службы эксплуатации количество сотрудников растет линейно вместе с развитием сопровождаемого сервиса: если соответствующие продукты развиваются успешно, то с увеличением объема трафика увеличивается и нагрузка на группу сопровождения. Это означает, что придется нанимать

все больше людей для выполнения одной и той же работы снова и снова.

Чтобы этого избежать, команда должна управлять потребностями системы, иначе она утонет в запросах. Поэтому в Google установлен лимит 50 % для операционной работы всех SR-инженеров — запросы на оперативное вмешательство в работу системы («тикеты»), дежурство, выполняемые вручную действия и т.д. Этот лимит гарантирует, что в расписании команды SRE достаточно времени на работы по улучшению сервиса, чтобы он оставался стабильным и работоспособным. Это верхняя граница. Со временем предоставленная сама себе команда SRE должна прийти к минимизации работ по эксплуатации и практически полностью посвящать себя разработке, поскольку сервис, по сути, работает автономно и восстанавливает себя сам: мы хотим, чтобы системы были *автоматическими*, а не только *автоматизированными*. На практике масштабирование и ввод нового функционала держит SR-инженеров в тонусе.

Итак, в Google есть простое ключевое правило: SR-инженеры должны тратить оставшиеся 50 % своего времени на разработку. Но как обеспечить такое распределение рабочего времени? Для начала мы должны узнать, как тратят свое время SR-инженеры. Имея эти данные, мы контролируем, чтобы команды, которые отдают разработке меньше 50 % времени, скорректировали текущий процесс. Зачастую это означает передачу некоторой части операционной нагрузки команде разработки или введение в SRE-команду людей без добавления дополнительных обязанностей по эксплуатации. Рациональное управление балансом между задачами по эксплуатации и разработке позволяет нам гарантировать, что SR-инженеры имеют возможность создавать креативные и автономные

инженерные решения, при этом сохраняя и знания, почерпнутые из опыта операционной работы.

Выяснилось, что основанный на SRE подход Google к построению и эксплуатации крупномасштабных систем имеет множество преимуществ. Поскольку SR-инженеры перерабатывают код, стремясь к тому, чтобы системы Google работали самостоятельно, для них свойственны стремление к быстрым инновациям и способность легко принимать нововведения. Такие команды относительно недороги — поддержка такого же сервиса только силами службы эксплуатации потребовала бы привлечения большего количества людей (операторов). Вместо этого количество SR-инженеров, достаточное для сопровождения и доработок системы, растет медленнее, чем сама система. Наконец, при таком подходе удается не только избежать проблем, связанных с размежеванием разработчиков и «операторов», но и повысить уровень самих разработчиков: без возможности легкого перемещения между командами разработки и SRE им было бы нелегко изучить особенности построения распределенной системы из миллионов узлов.

Несмотря на все эти преимущества, использование модели SRE связано с некоторыми трудностями. Одна из проблем, с которой постоянно сталкивается Google, — наем SR-инженеров: SRE и отдел разработки продуктов конкурируют за одних и тех же кандидатов. Кроме того, база кандидатов сравнительно невелика, так как в компании установлена высокая планка требований для навыков программирования и проектирования систем. Поскольку наша дисциплина относительно нова и уникальна, на текущий момент имеется не так уж много информации о том, как создать команду SRE и управлять ею (надеемся, что эта книга поможет исправить ситуацию!). Как только команда SRE будет укомплектована, ее

потенциально непривычные подходы к управлению сервисами потребуют серьезной поддержки со стороны менеджмента. Например, решение приостановить выпуск версий до конца квартала лишь из-за того, что исчерпан лимит времени недоступности сервиса, может быть негативно встречено командой разработчиков, если только оно не санкционировано руководством.

DevOps или SRE?

Термин DevOps появился в отрасли в конце 2008 года и на момент написания этой книги (начало 2016 года) все еще не вполне сформировался. Его основные принципы – привнесение IT-составляющей в каждую фазу проектирования и создания системы, максимальное использование автоматизации вместо человеческого труда, применение инженерных подходов и программных инструментов для решения задач эксплуатации – совпадают со многими принципами и рекомендациями SRE. DevOps можно рассматривать как обобщение некоторых основных принципов SRE для более широкого круга организаций, управленческих структур и персонала. В свою очередь, SRE можно рассматривать как конкретную реализацию DevOps с некоторыми специфическими расширениями.

Принципы SRE

Несмотря на то что особенности организации труда, приоритетов и ежедневных задач у разных команд различны,

все эти команды имеют базовый набор обязанностей по отношению к сервису, который они обслуживают, и придерживаются одинаковых принципов. В общем случае команда SR-инженеров отвечает за *доступность, время отклика, производительность, эффективность, управление изменениями, мониторинг, реагирование в аварийных и критических ситуациях и планирование производительности* для своих сервисов. Мы систематизировали правила и принципы взаимодействия команд SR-инженеров с их окружением — не только с сопровождаемыми системами, но и с командами разработчиков и тестировщиков, пользователями и т.д. Эти правила и рекомендации помогают нам сосредоточиться на работе инженера, а не на операционных задачах.

В следующем разделе рассматриваются все основные принципы Google SRE.

Уделяем особое внимание инженерным задачам

Как мы уже говорили, по правилам Google на операционные задачи выделяется не более 50 % от общего времени SR-инженеров. Остальное время должно быть использовано для работы над проектами с применением навыков программирования. На практике это достигается путем наблюдения за количеством операционной работы, выполняемой SR-инженерами, и перенаправлением избытка таких задач командам разработчиков: переадресацией ошибок и поступающих запросов менеджерам по разработке, привлечением разработчиков к дежурствам и т.д. Перенаправление заканчивается, когда операционная нагрузка снижается до 50 % и менее. Это также обеспечивает эффективную обратную связь, ориентируя разработчиков на создание систем, не требующих вмешательства человека. Чтобы такой подход хорошо работал, все в компании — и SRE-

отдел, и разработчики — должны понимать, почему существует это ограничение, и стремиться сократить объем генерируемой сопровождаемым продуктом операционной работы, дабы не провоцировать превышение лимита.

Занимаясь операционными задачами, каждый SR-инженер, как правило, получает не более двух событий за 8–12-часовую смену. Такой объем работы дает ему возможность быстро и точно обработать событие, привести все в порядок и восстановить сервис (в случае ошибки), а затем проанализировать причины произошедшего. Если появляется более двух событий за дежурство, проблемы обычно не удается изучить досконально и у инженеров не хватает времени для того, чтобы предотвратить будущие подобные ошибки, разбравшись в текущих. По мере масштабирования эта ситуация не исправляется. С другой стороны, если SR-инженер стablyно получает менее одного события за дежурство, его работа на месте дежурного — пустая трата времени.

Отчеты с анализом причин произошедшего (так называемые постмортемы) необходимо писать для всех значимых инцидентов, независимо от того, сопровождались ли они уведомлениями. Постмортемы для событий, уведомлений о которых не было, даже более ценные, поскольку они, вероятно, указывают на пробелы мониторинга. Подобное расследование должно установить все детали случившегося, найти все первоначальные причины проблемы и выработать план действий по ее устранению или улучшению способа обработки такого события в случае его повторного возникновения. В Google никого не обвиняют в ошибках — цель состоит в выявлении ошибок и исправлении их, а не в избегании или замалчивании.

Добиваемся максимальной скорости внедрения изменений без потери качества обслуживания

Команды разработчиков и SR-инженеров смогут наслаждаться эффективными рабочими отношениями, избавившись от противоречия между их целями. Это противоречие заключается в соотношении темпов внедрения изменений и стабильности продукта и, как говорилось ранее, часто проявляется неявно. В нашей модели SRE мы выводим этот конфликт на первый план, а затем избавляемся от него, вводя понятие *суммарного уровня*, или *бюджета ошибок* (*error budget*).

Это понятие основано на наблюдении, что *достигнение 100%-ной надежности будет необоснованным требованием в большинстве ситуаций* (за исключением, например, кардиостимуляторов и антиблокировочных систем в тормозах). В общем случае для любого программного сервиса или системы 100 % — неверный ориентир для показателя надежности, поскольку ни один пользователь не сможет заметить разницу между 100%-ной и 99,999%-ной доступностью. Между пользователем и сервисом находится множество других систем (его ноутбук, домашний Wi-Fi, провайдер, энергосеть...), и все эти системы в совокупности доступны не в 99,999 % случаев, а гораздо реже. Поэтому разница между 99,999 % и 100 % теряется на фоне случайных факторов, обусловленных недоступностью других систем, и пользователь не получает никакой пользы от того, что мы потратили кучу сил, добавляя эту последнюю долю процента в доступность системы.

Если нам не нужно стремиться к 100%-ному уровню надежности системы, то к какому тогда? На самом деле это не технический вопрос — это вопрос к продукту, и вам нужно учесть следующие моменты.

- Какой показатель доступности удовлетворит пользователей, если мы знаем о том, как они используют продукт?
- Какие альтернативы имеют пользователи, не удовлетворенные доступностью продукта?
- Как отразится на активности обращений пользователей к продукту изменение уровня его доступности?

Продукт должен иметь установленный целевой показатель доступности. Как только этот показатель определен, допустимый суммарный уровень ошибок будет равен единице минус запланированный показатель доступности. Сервис, который доступен в 99,99 % случаев, недоступен в 0,01 % случаев. Этот допустимый уровень недоступности и есть не что иное, как допустимый суммарный уровень ошибок (*бюджет ошибок*). Мы можем тратить этот «бюджет» на все, что сочтем нужным, пока не выходим за его рамки.

Как же мы собираемся потратить этот бюджет? Команда разработки намерена внедрять новый функционал и привлекать новых пользователей. В идеале мы могли бы потратить весь лимит количества возможных неполадок сервиса на риски, связанные с внедрением, чтобы быстрее запустить продукт. Это простое предположение характеризует в целом всю модель бюджета ошибок. Поскольку деятельность SR-инженеров строится в рамках этой концептуальной модели, высвобождение бюджета ошибок благодаря методам вроде поэтапного развертывания и одного «экспериментального» процента⁹ — это оптимизация, которая позволяет ускорить процесс выпуска продуктов.

Применение принципа допустимого суммарного уровня ошибок разрешает противоречие интересов между командами

разработчиков и SR-инженеров. Цель SR-инженеров уже не сводится к обеспечению отсутствия сбоев; вместо этого обе команды стремятся расходовать предоставленный бюджет ошибок так, чтобы максимально быстро внедрить новый функционал и выпустить продукт. Это и есть главное отличие. Сбои и дефекты (баги) больше не считаются чем-то «плохим» — это ожидаемая часть процесса внедрения новшеств¹⁰. Команды разработчиков и SR-инженеров теперь не боятся их, а управляют ими.

Мониторинг

Мониторинг (наблюдение) — это одно из основных средств отслеживания состояния системы и ее доступности. Мониторинг требует продуманной стратегии. Классический широко распространенный подход к мониторингу предусматривает наблюдение за определенным параметром или условием и, если заданное значение параметра превышено или условие выполнено, отправку оповещения по электронной почте. Однако такое оповещение по электронной почте — неэффективное решение: система, которая требует от человека прочесть электронное письмо и решить, нужно ли предпринимать какие-то действия, в принципе неполноценна. Система мониторинга никогда не должна требовать от человека истолковывать какую-либо часть оповещения. Вместо этого всю интерпретацию должно выполнять программное обеспечение, а люди будут оповещены только в том случае, когда от них требуется предпринять какие-либо действия.

Существует три категории данных от системы мониторинга.

- *Срочные оповещения (alerts, «алерты»)* — указывают, что нужно немедленно реагировать на что-то, что либо уже произошло, либо вот-вот произойдет.

- *Запросы на действия* (*tickets*, «тикеты») — указывают, что человеку нужно вмешаться, но не обязательно немедленно. Система не может обработать ситуацию автоматически, но предпринимать какие-то действия допустимо не сразу, а в течение нескольких дней без каких-либо негативных последствий.
- *Журналирование* (*logging*) — нет необходимости кому-либо просматривать эту информацию, она записывается для диагностических целей или для последующего анализа. По умолчанию журнал читать не требуется, пока что-либо иное не заставит сделать это.

Реагирование на критические ситуации

Надежность — это функция от среднего времени безотказной работы (mean time to failure, MTTF) и среднего времени восстановления (mean time to repair, MTTR) [Schwartz, 2015]. Наиболее значимый критерий при оценке эффективности реагирования на аварии и другие критические ситуации — это быстрота восстановления работоспособности системы, то есть MTTR.

Выполнение операций вручную приводит к увеличению задержек. Система, способная избегать аварий, которые потребовали бы ручного вмешательства (хотя бы в отношении только наиболее частых сбоев), будет иметь лучшие показатели доступности, чем если она нуждалась в таком вмешательстве всегда. Рассматривая ситуации, когда вмешательство людей все же необходимо, мы обнаружили, что продумывание всех деталей и превентивная запись методических рекомендаций в инструкцию приводит к практически трехкратному улучшению времени восстановления (MTTR) по сравнению с неподготовленными

импровизациями. Конечно, героический дежурный инженер, мастер на все руки, выполнит всю необходимую работу, но обычный опытный дежурный инженер, вооруженный инструкцией, справится с этим гораздо лучше. Несмотря на то что ни одна инструкция, какой бы полной она ни была, не заменит толковых инженеров, способных импровизировать на ходу, четкое и исчерпывающее описание шагов и советы по поиску неисправностей очень ценные в тех ситуациях, когда нужно отреагировать на критическое или не терпящее промедления происшествие. Поэтому в Google SRE при подготовке дежурных инженеров полагаются на инструкции, в дополнение к упражнениям вроде «Колеса неудачи»¹¹.

Управление изменениями

Отдел SRE выяснил, что около 70 % сбоев спровоцированы изменениями в уже работающей системе. Рекомендуется использовать автоматизацию для того, чтобы:

- обеспечить поэтапное развертывание обновлений ПО;
- быстро и точно выявлять проблемы;
- безопасно откатывать изменения при возникновении проблем.

Эти три приема позволяют эффективно ограничивать общее количество пользователей и процессов, подвергающихся сбоям. Устранив человеческий фактор из цикла управления изменениями, можно избежать распространенных проблем: усталости, расслабленности, пренебрежения и невнимательности по отношению к часто повторяющимся

задачам. В результате повышаются как скорость, так и надежность внедрения и обновления ПО.

Прогнозирование нагрузки и планирование производительности

Процесс прогнозирования нагрузки и планирования производительности (пропускной способности, capacity) системы можно рассматривать как обеспечение гарантии того, что она будет иметь достаточную производительность и даже некоторую избыточность для удовлетворения прогнозируемой нагрузки с требуемым показателем доступности. В этих концепциях нет ничего особенного, помимо того, что на удивление много команд не предпринимают никаких действий, чтобы это обеспечить. При планировании должен учитываться как естественный количественный рост (вызванный ростом популярности продукта у клиентов), так и скачкообразный (который проявляется при запуске нового функционала, маркетинговых кампаний или других изменениях, проводимых в интересах бизнеса).

При планировании производительности обязательны следующие шаги:

- точное прогнозирование естественного роста, причем простирающееся и за пределы срока, требуемого для ввода новых мощностей;
- точное прогнозирование скачкообразного (по разным причинам) роста нагрузки;
- регулярное нагрузочное тестирование системы для установления соответствия между «чистой» производительностью компонентов системы (серверов,

дисков и т.д.) и результирующей пропускной способностью сервиса.

Поскольку пропускная способность системы критична для обеспечения требуемых показателей ее доступности, из этих принципов естественным образом следует, что команда SRE должна отвечать за планирование мощностей и, следовательно, за материально-техническое обеспечение.

Материально-техническое обеспечение

Материально-техническое обеспечение объединяет в себе управление модернизациями системы и планирование ее пропускной способности. Наш опыт говорит, что снабжение должно осуществляться быстро, но только когда это действительно необходимо, поскольку оборудование обходится недешево. Кроме того, вводить новые мощности тоже нужно правильно, чтобы они заработали как полагается. Наращивание производительности часто предусматривает введение новых экземпляров систем (instance) или площадок размещения инфраструктуры, внесение значительных изменений в существующие системы (конфигурационные файлы, балансировщики нагрузки, настройки сети) и проверку того, что новые мощности работают корректно и эффективно. Поэтому такая операция более рискованна, нежели перераспределение нагрузки (которое может выполняться по нескольку раз за час), и проводить ее нужно крайне осторожно.

Эффективность и производительность

Эффективное использование ресурсов всегда важно для сервиса, создатели которого заботятся о деньгах. Поскольку отдел SRE управляет в конечном счете материально-

техническим обеспечением, он также должен быть вовлечен в любую работу, направленную на повышение «коэффициента использования», своего рода КПД системы — функции, которая показывает, насколько хорошо работает заданный сервис и насколько полно он обеспечивается ресурсами. Из этого вытекает, что стратегия материально-технического обеспечения сервиса и, как следствие, оптимизация его «коэффициента использования» являются мощным фактором, влияющим на общую стоимость сервиса.

Использование ресурсов — это функция, учитывающая требуемый объем ресурсов (нагрузку), производительность и эффективность работы ПО. SR-инженеры прогнозируют требования, обеспечивают производительность и могут модифицировать ПО. Эти три фактора составляют весомую часть общей эффективности сервиса (хотя это и не все ее составляющие).

Программные системы становятся все медленнее по мере нагрузки на них. Замедление сервиса приводит к потере производительности. В какой-то момент замедлившаяся система перестанет обслуживать пользователей, что сделает ее бесконечно медленной. SR-инженеры поддерживают производительность *на заданном уровне*, поэтому они крайне заинтересованы в производительности ПО сервиса. SR-инженеры и разработчики продукта будут (и должны) следить за работой сервиса и модифицировать его для повышения производительности, тем самым увеличивая пропускную способность и повышая эффективность¹².

Конец начала

Техника обеспечения надежности информационных систем — SRE — компании Google представляет собой явление,

значительно отличающееся от передовых практик управления крупными и сложными системами. Поначалу мотивированное знанием предмета — «если нужно выполнять набор повторяющихся задач, то я должен тратить на это свое время именно как разработчик», — это явление в итоге стало чем-то большим: набором принципов, практик, рекомендаций, побудительных мотивов, а также полем для экспериментов внутри более крупной дисциплины — разработки ПО. Остальная часть книги описывает «путь SRE» более подробно.

[9](#) Здесь не вполне ясно, что конкретно имеется в виду. Скорее всего, ограничение внедрений опытных версий и проведения экспериментов над действующими системами одним процентом от общего их количества. — *Примеч. пер.*

[10](#) Разумеется, и авторы упоминают это, такой подход возможен лишь при условии, что ошибки и сбои действительно допустимы. — *Примеч. пер.*

[11](#) См. подраздел «Катастрофа: ролевая игра» раздела «Пять приемов для вдохновления дежурных работников» главы 28.

[12](#) Чтобы узнать, как такое взаимодействие может работать на практике, прочтите раздел «Общение: производственные совещания» главы 31.

2. Среда промышленной эксплуатации Google с точки зрения SRE

Автор — Джей Си ван Винкель

Под редакцией Бетси Байер

Дата-центры (центры обработки данных) Google значительно отличаются от традиционных дата-центров и небольших серверных «ферм». Эти различия привносят как дополнительные проблемы, так и дополнительные возможности. В этой главе рассматриваются проблемы и возможности, характерные для дата-центров Google, и вводится терминология, которая будет использована на протяжении всей книги.

Оборудование

Большая часть вычислительных ресурсов Google располагается в спроектированных компанией дата-центрах, имеющих собственную систему энергоснабжения, систему охлаждения, внутреннюю сеть и вычислительное оборудование [Barroso et al., 2013]. В отличие от типичных дата-центров, предоставляемых провайдерами своим клиентам, все дата-центры Google оснащены одинаково¹³. Чтобы избежать путаницы между серверным оборудованием и серверным ПО, в этой книге мы используем следующую терминологию:

- *машина (компьютер)* — единица оборудования (или, возможно, виртуальная машина);
- *сервер* — единица программного обеспечения, которая реализует сервис.

На машинах может быть запущен любой сервер, поэтому мы не выделяем конкретные компьютеры для конкретных серверных программ. Например, у нас нет конкретной машины, на которой работает почтовый сервер. Вместо этого ресурсы распределяются нашей системой управления кластерами *Borg*.

Мы понимаем, что такое использование термина «сервер» нестандартно. Более привычно обозначать им сразу два понятия: программу, которая обслуживает сетевые соединения, и одновременно машину, на которой исполняются такие программы, но, когда мы говорим о вычислительных мощностях Google, разница между двумя этими понятиями существенна. Как только вы привыкнете к нашей трактовке слова «сервер», вам станет понятнее, почему важно использовать именно такую специализированную терминологию не только непосредственно в Google, но и на протяжении всей этой книги.

На рис. 2.1 продемонстрирована конфигурация дата-центра Google.

- Десятки машин размещаются на *стойках*.
- Стойки стоят *рядами*.
- Один или несколько рядов образуют *кластер*.
- Обычно в здании *центра обработки данных (ЦОД)*, или *дата-центра*, размещается несколько кластеров.
- Несколько зданий дата-центров, которые располагаются близко друг к другу, составляют *кампус*.

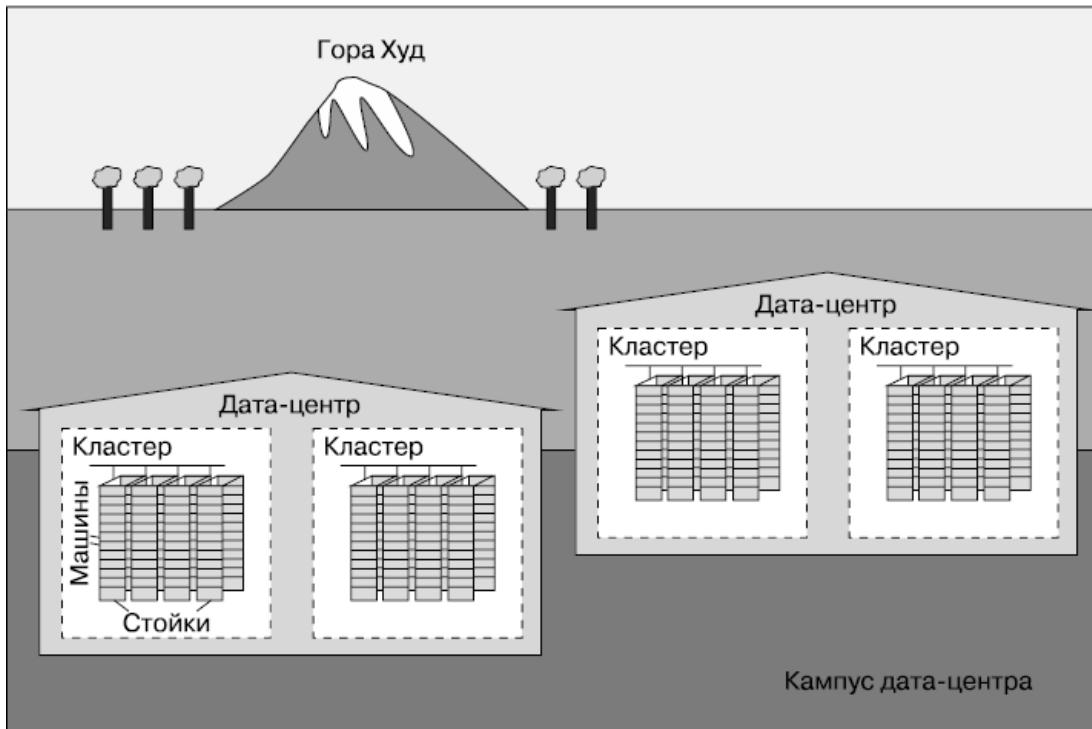


Рис. 2.1. Пример топологии кампуса данных Google

Внутри каждого дата-центра все машины должны иметь возможность эффективно общаться друг с другом, поэтому мы создали очень быстрый виртуальный коммутатор (switch) с десятками тысяч портов. Это удалось сделать, соединив сотни разработанных в Google коммутаторов в «фабрику» на основе топологии сети Клоза [Clos, 1953], названную *Jupiter* [Singh et al., 2015]. В своей максимальной конфигурации *Jupiter* поддерживает пропускную способность¹⁴ 1,3 Пб/с между серверами.

Дата-центры соединены друг с другом с помощью нашей глобальной магистральной сети *B4* [Jain et al., 2013]. *B4* имеет программно-конфигурируемую сетевую архитектуру и использует открытый коммуникационный протокол OpenFlow. *B4* предоставляет широкую полосу пропускания ограниченному количеству систем и использует гибкое

управление шириной канала для максимизации среднего ее значения [Kumar et al., 2015].

Системное ПО, которое «организует» оборудование

Программное обеспечение, которое обеспечивает управление и администрирование нашего оборудования, должно быть способно справляться с системами огромного масштаба. Сбои оборудования — это одна из основных проблем, решаемая с помощью ПО. Учитывая большое количество аппаратных компонентов в кластере, случаются они довольно часто. В каждом кластере за год обычно отказывают тысячи машин и выходят из строя тысячи жестких дисков. Если умножить это количество на число кластеров, функционирующих по всему миру, результат ошеломляет. Поэтому мы хотим изолировать пользователей от подобных проблем, и команды, занимающиеся нашими сервисами, также не хотят отвлекаться на аппаратные проблемы. В каждом кампусе данных есть команды, отвечающие за поддержку оборудования и инфраструктуру данных.

Управление машинами

Borg (рис. 2.2) — это распределенная система управления кластерами [Verma et al., 2015], похожая на Apache Mesos¹⁵. *Borg* управляет заданиями на уровне кластеров.

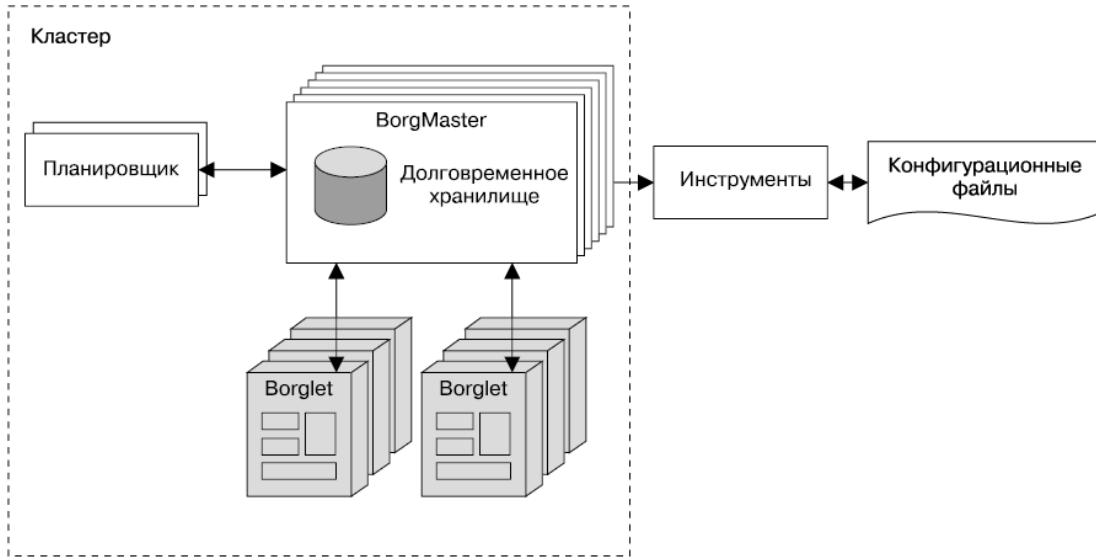


Рис. 2.2. Общая кластерная архитектура Borg

Borg отвечает за запуск *заданий* (*jobs*) пользователей. Эти задания могут представлять собой как постоянно работающие сервисы, так и процессы пакетной обработки вроде MapReduce [Dean and Ghemawat, 2004]. Они могут состоять из нескольких (иногда и тысяч) идентичных *задач* (*tasks*) — как по соображениям надежности, так и потому, что один процесс, как правило, не способен обработать весь трафик кластера. Когда Borg запускает задание, он находит машины для выполнения его задач и командует им запустить программу-сервер. Далее Borg отслеживает состояние этих задач. Если задача работает некорректно, она уничтожается и перезапускается, возможно, на другой машине.

Поскольку задачи свободно распределяются между машинами, мы не можем использовать для обращения к ним IP-адреса и номера портов. Эта проблема решается дополнительным уровнем абстракции: при запуске задания Borg выделяет имя для задания и номер (индекс) для каждой его задачи с помощью *сервиса именования Borg* (*Borg Naming Service*, BNS). Вместо того чтобы использовать IP-адрес и номер порта, другие процессы связываются с задачами Borg по их

BNS-имени, которое затем BNS преобразует в IP-адрес и номер порта. Например, путь BNS может быть строкой вроде `/bns/<кластер>/<пользователь>/<имя_задания>/<номер_задачи>`, которая затем транслируется (в сетях принято говорить «разрешается») в формат `<IP-адрес>:<порт>`.

Borg также отвечает за выделение ресурсов для заданий. Каждое задание должно указать, какие ресурсы требуются для его выполнения (например, три ядра процессора, 2 Гбайт оперативной памяти). Используя список требований всех заданий, Borg может оптимально распределять задания между машинами, учитывая также и соображения отказоустойчивости (например, Borg не будет запускать все задачи одного задания на одной и той же стойке, так как коммутатор данной стойки в случае сбоя окажется критической точкой для этого задания).

Если задача пытается захватить больше ресурсов, чем было затребовано, Borg уничтожает ее и затем перезапускает (поскольку обычно предпочтительнее иметь задачу, которая иногда аварийно завершается и перезапускается, чем которая не перезапускается вовсе).

Хранилище

Для более быстрого доступа к данным задачи могут использовать локальный диск машин, но у нас есть несколько вариантов организации постоянного хранилища в кластере (и даже локально хранимые данные в итоге будут перемещаться в кластерное хранилище). Их можно сравнить с Lustre и Hadoop Distributed File System (HDFS) — кластерными файловыми системами, имеющими реализацию с открытым исходным кодом.

Хранилище обеспечивает пользователям возможность простого и надежного доступа к данным, доступным для

кластера. Как показано на рис. 2.3, хранилище имеет несколько слоев.

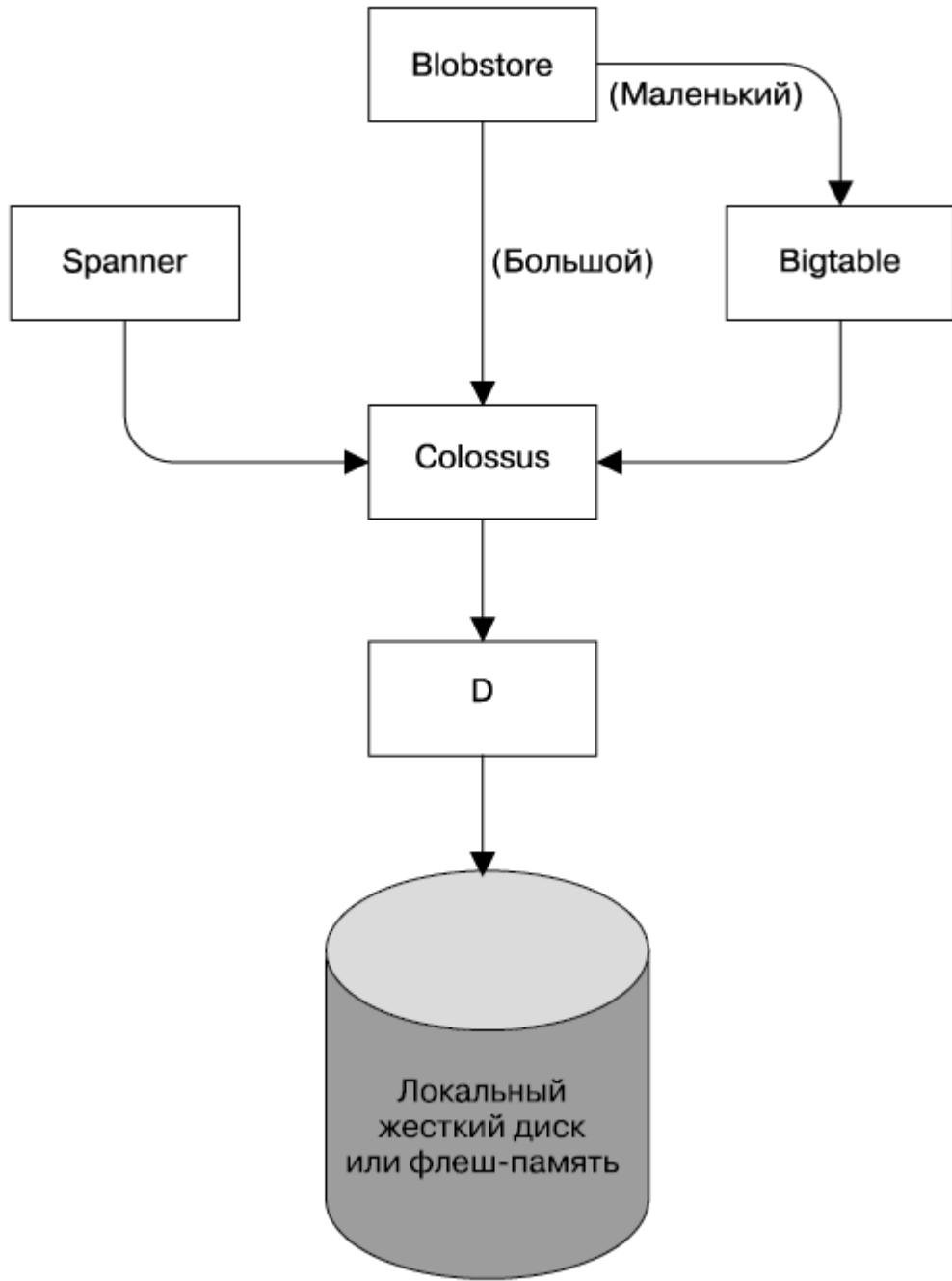


Рис. 2.3. Составляющие стека хранилищ Google

1. Самый нижний слой называется *D* (от *disk*, хотя уровень *D* использует как традиционные жесткие диски, так и

накопители с флеш-памятью). D — это файловый сервер, работающий практически на всех машинах кластера. Однако пользователи, желающие получить доступ к своим данным, не хотели бы запоминать, на какой машине те хранятся, поэтому здесь подключается следующий слой.

2. Над слоем D располагается слой *Colossus*, который создает в кластере файловую систему, предлагающую обычную семантику файловой системы, а также репликацию и шифрование. *Colossus* является наследником GFS, Google File System (файловая система Google) [Ghemawat et al., 2003].
3. Далее, существует несколько похожих на базы данных сервисов, построенных над уровнем *Colossus*.
 - *Bigtable* [Chang et al., 2006] — это нереляционная (NoSQL) система баз данных, способная работать с базами объемом в петабайты. *Bigtable* — это разреженная распределенная отказоустойчивая многомерная упорядоченная база данных, которая индексируется по ключам строк, столбцов и временным меткам; каждое значение базы данных — это произвольный неинтерпретированный массив байтов. *Bigtable* также поддерживает репликацию между data-центрами.
 - *Spanner* [Corbett et al., 2012] предлагает SQL-подобный интерфейс для пользователей, которым требуется целостность и согласованность данных при доступе из любой точки мира.
 - Доступны и некоторые другие системы баз данных, например *Blobstore*. Все они имеют свои достоинства и недостатки (см. главу 26).

Сеть

Сетевое оборудование Google управляется несколькими способами. Как говорилось ранее, мы используем программно-конфигурируемую сеть, основанную на OpenFlow. Вместо «умных» маршрутизаторов мы используем не столь дорогие «глупые» коммутаторы в сочетании с центральным (продублированным) контроллером, который заранее вычисляет лучший маршрут в сети. Это и позволяет использовать более простое коммутирующее оборудование, освободив его от трудоемкого поиска маршрута.

Пропускная способность сети должна грамотно распределяться. Как Borg ограничивает вычислительные ресурсы, которые может использовать задача, так и Bandwidth Enforcer (BwE) управляет доступной полосой пропускания так, чтобы максимизировать среднюю пропускную способность. Оптимизация пропускной способности связана не только со стоимостью: централизованное управление трафиком позволяет решить ряд проблем, которые крайне плохо поддаются решениюю сочетанием распределенной маршрутизации и обычного управления трафиком (Kumar, 2015).

Некоторые сервисы имеют задания, запущенные на нескольких кластерах, размещенных в разных точках мира. Для того чтобы снизить время задержки глобально распределенных систем, мы хотели бы направить пользователей в ближайший дата-центр, имеющий подходящие для этого мощности. Наш *глобальный программный балансировщик нагрузки* (Global Software Load Balancer, GSLB) выполняет балансировку нагрузки на трех уровнях:

- географическую балансировку нагрузки для DNS-запросов (например, к www.google.com), она описана в главе 19;

- балансировку нагрузки на уровне пользовательских сервисов (например, YouTube или Google Maps);
- балансировку нагрузки на уровне удаленных вызовов процедур (Remote Procedure Call, RPC), описанную в главе 20.

Владельцы сервисов задают для них символические имена, список BNS-адресов серверов и производительность, доступную на каждой площадке (обычно она измеряется в *запросах в секунду* – queries per second, QPS). В дальнейшем GSLB направляет трафик по указанным BNS-адресам.

Другое системное ПО

В программном обеспечении дата-центров есть и другие важные компоненты.

Сервис блокировок

Сервис блокировок *Chubby* [Burrows, 2006] предоставляет API, схожий с файловой системой и предназначенный для обслуживания блокировок. Chubby обрабатывает блокировки всех дата-центров. Он использует протокол Paxos для асинхронного обращения к Consensus (см. главу 23).

Chubby также играет важную роль при выборе мастера. Если для какого-то сервиса с целью повышения надежности предусмотрено пять реплик задания, но в конкретный момент реальную работу выполняет только одна из них, то для выбора этой реплики используется Chubby.

Chubby отлично подходит для данных, которые требуют от хранилища надежности. По этой причине BNS использует

Chubby для хранения соотношения BNS-путей и пар IP-адрес:порт.

Мониторинг и оповещение

Мы хотим быть уверены, что все сервисы работают как следует. Поэтому мы запускаем множество экземпляров программы мониторинга *Borgmon* (см. главу 10). *Borgmon* регулярно получает значения контрольных показателей от наблюдаемых сервисов. Эти данные могут быть использованы немедленно для оповещения или сохранены для последующей обработки и анализа, например для построения графиков. Такой мониторинг может применяться для таких целей, как:

- настройка оповещений о неотложных проблемах;
- сравнение поведения: ускорило ли обновление ПО работу сервера;
- оценка характера изменения потребления ресурсов со временем, что необходимо для планирования мощностей.

Наша инфраструктура ПО

Архитектура нашего ПО спроектирована так, чтобы можно было наиболее эффективно использовать аппаратные ресурсы системы. Весь наш код многопоточный, поэтому одна задача с легкостью может задействовать несколько ядер. В целях поддержки информационных панелей (dashboards), мониторинга и отладки каждый сервер включает в себя реализацию сервера HTTP в качестве интерфейса, через который предоставляется диагностическая информация и статистика по конкретной задаче.

Все сервисы Google «общаются» с помощью инфраструктуры удаленных вызовов процедур (RPC), которая называется *Stubby*. Существует ее версия с открытым исходным кодом, она называется gRPC (см. <http://grpc.io>). Зачастую вызов RPC выполняется даже для подпрограмм в локальной программе. Это позволяет переориентировать программу на вызовы другого сервера для достижения большей модульности или по мере разрастания исходного объема кода сервера. GSLB может выполнять балансировку нагрузки RPC точно так же, как и для внешних интерфейсов сервисов.

Сервер получает запросы RPC с *фронтенда* и отправляет RPC в *бэкенд*. Пользуясь традиционными терминами, фронтенд называется *клиентом*, а бэкенд — *сервером*.

Данные передаются в RPC и из них посредством *протокола сериализации* — так называемых *протокольных буферов*¹⁶ (*protocol buffers*), или, кратко, *protobufs*¹⁷. Этот протокол похож на Thrift от Apache и имеет ряд преимуществ перед XML, когда речь идет о сериализации структурированных данных: он проще, от трех до десяти раз компактнее, от 20 до 100 раз быстрее и более однозначный.

Наша среда разработки

Скорость разработки продуктов очень важна для Google, поэтому мы создали специальную среду, максимально использующую возможности своей инфраструктуры [Morgenthaler et al., 2012].

За исключением нескольких групп, продукты которых имеют открытый код, и поэтому для них используются свои отдельные репозитории (например, Android и Chrome), инженеры-программисты Google работают в одном общем репозитории [Potvin, Levenberg, 2016]. Такой подход имеет

несколько практических применений, важных для нашего производственного процесса.

- Если инженер сталкивается с проблемой в компоненте, за пределами своего проекта, он может исправить проблему, выслать предлагаемые изменения («список изменений» — changelist, CL) владельцу на рассмотрение и затем внедрить сделанные изменения в основную ветвь программы.
- Изменения исходного кода в собственном проекте инженера требуют рассмотрения — проведения ревизии (ревью). Весь софт перед принятием проходит этот этап.

Когда выполняется сборка ПО, запрос на сборку отправляется на специализированные серверы дата-центра. Даже сборка крупных проектов выполняется быстро, поскольку можно использовать несколько серверов для параллельной компиляции. Такая инфраструктура также применяется для непрерывного тестирования. Каждый раз, когда появляется новый список изменений (CL), выполняются тесты всего ПО, на которое могут повлиять эти изменения прямо или косвенно. Если фреймворк обнаруживает, что изменения нарушили работу других частей системы, он оповещает владельца этих изменений. Отдельные проекты используют систему *push-on-green* («отправка при успехе»), согласно которой новая версия автоматически отправляется в промышленную эксплуатацию после прохождения тестов.

Shakespeare: пример сервиса

Для того чтобы продемонстрировать, как в компании Google сервис разворачивается в среде промышленной эксплуатации,

рассмотрим пример гипотетического сервиса, который взаимодействует с технологиями Google. Предположим, что мы хотим предложить сервис, который позволяет определить, в каких произведениях Шекспира встречается указанное вами слово.

Мы можем разделить систему на две части.

- Компонент пакетной обработки, который читает все тексты Шекспира, создает алфавитный указатель и записывает его в Bigtable. Эта задача (точнее, задание) выполняется однократно или, возможно, изредка (ведь может обнаружиться какой-нибудь новый текст Шекспира!).
- Приложение-фронтенд, обрабатывающее запросы конечных пользователей. Это задание всегда запущено, поскольку в любой момент времени пользователь из любого часового пояса может захотеть выполнить поиск по книгам Шекспира.

Компонентом пакетной обработки будет сервис MapReduce, чья работа делится на три фазы.

1. В фазе Mapping тексты Шекспира считаются и разбиваются на отдельные слова. Эта часть работы будет выполнена быстрее, если запустить параллельно несколько рабочих процессов (задач).
2. В фазе Shuffle записи сортируются по словам.
3. В фазе Reduce создаются кортежи вида ([слово](#), [список_произведений](#)).

Каждый кортеж записывается в виде строки в Bigtable, ключом выступает слово.

Жизненный цикл запроса

На рис. 2.4 показано, как обрабатывается запрос пользователя. Сначала пользователь переходит в браузере по ссылке **shakespeare.google.com**. Для получения соответствующего IP-адреса устройство пользователя транслирует («разрешает») адрес с помощью DNS-сервера (1). DNS-запрос в итоге оказывается на DNS-сервере Google, который взаимодействует с GSLB. Отслеживая загруженность трафиком всех фронтенд-серверов по регионам, GSLB выбирает, IP-адрес какого из серверов нужно возвратить пользователю.

Браузер соединяется с HTTP-сервером по указанному адресу. Этот сервер (он называется Google Frontend или GFE) представляет собой «обратный» прокси-сервер (reverse proxy), находящийся на другом конце TCP-соединения клиента (2). GFE выполняет поиск требуемого сервиса (например, это может быть поисковый сервис, карты или — в нашем случае — сервис Shakespeare). Повторно обращаясь к GSLB, сервер находит доступный фронтенд-сервер Shakespeare и обращается к нему посредством удаленного вызова процедуры (RPC), передавая полученный от пользователя HTTP-запрос (3).

Сервер Shakespeare анализирует HTTP-запрос и создает «протокольный буфер» (protobuf), содержащий слова, которые требуется найти. Теперь фронтенд-сервер Shakespeare должен связаться с бэкенд-сервером Shakespeare: первый связывается с GSLB, чтобы получить BNS-адрес подходящего и незагруженного экземпляра второго (4). Далее бэкенд-сервер Shakespeare связывается с сервером Bigtable для получения запрашиваемых данных (5).

Результат записывается в ответный protobuf и возвращается на бэкенд-сервер Shakespeare. Бэкенд передает protobuf с результатом работы сервиса фронтенд-серверу Shakespeare, который создает HTML-документ и возвращает его в качестве ответа пользователю.

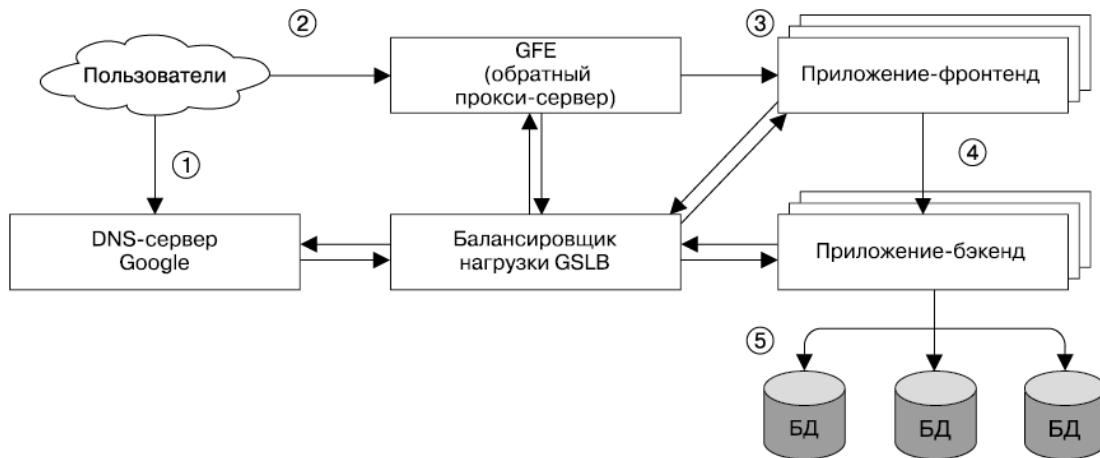


Рис. 2.4. Жизненный цикл запроса

Вся эта цепочка событий выполняется в мгновение ока — всего за несколько сотен миллисекунд! Поскольку задействовано множество компонентов, существует множество мест, где потенциально может возникнуть ошибка; в частности, сбой в GSLB может дезорганизовать всю работу и привести к коллапсу. Однако политика Google, предусматривающая строгий контроль, всеобъемлющее тестирование и безопасное развертывание новых программ в дополнение к нашим упреждающим методам восстановления при ошибках (вроде постепенного отключения функций), позволяет нам создавать надежные сервисы, отвечающие ожиданиям наших пользователей. В конце концов, люди регулярно обращаются к сайту www.google.com чтобы проверить, есть ли подключение к Интернету.

Организация задач и данных

Тестирование нагрузки показало, что наш бэкенд-сервер может обработать около 100 запросов в секунду (QPS). Опытная эксплуатация с ограниченным количеством пользователей показала, что пиковая нагрузка может достигать примерно 3470 QPS, поэтому нам нужно создавать как минимум 35 задач. Однако следующие соображения говорят, что нам понадобится как минимум 37 задач, или $N + 2$.

- Во время обновления одна задача будет временно недоступна, поэтому активными будут оставаться 36 задач.
- Во время обновления может произойти сбой в аппаратной части, из-за чего останется всего 35 задач — ровно столько, сколько нужно для обслуживания пиковой нагрузки¹⁸.

Более подробное исследование пользовательского трафика обнаруживает географическое распределение пиковой нагрузки: 1430 QPS генерируются из Северной Америки, 290 — из Южной Америки, 1400 — из Европы и 350 — из Азии и Австралии. Вместо того чтобы размещать все бэкенд-серверы в одном месте, мы распределяем их по регионам: в США, Южной Америке, Европе и Азии. Учитывая принцип $N + 2$ в каждом регионе, получаем 17 задач в США, 16 — в Европе и шесть — в Азии. В Южной Америке, однако, мы решаем использовать четыре задачи (вместо пяти), чтобы снизить затраты — с $N + 2$ до $N + 1$. В этом случае мы готовы взять на себя небольшой риск появления большего времени задержки и снизить стоимость оборудования: разрешив GSLB при перегрузке южноамериканского data-центра перенаправлять трафик с одного континента на другой, мы можем сэкономить 20 % ресурсов, которые были бы потрачены на оборудование. В

более крупных регионах для дополнительной устойчивости мы распределяем задачи между 2–3 кластерами.

Поскольку бэкенд-сервера должны связываться с хранилищем данных Bigtable, нам также нужно стратегически продумать это хранилище. Если бэкенд-сервер Азии будет связываться с Bigtable, расположенным в США, это приведет к значительному увеличению задержек, поэтому мы дублируем Bigtable в каждом регионе. Это дает нам дополнительную устойчивость на тот случай, если сервер Bigtable даст сбой, а также снижает время задержки доступа к данным. И хотя Bigtable не обеспечивает строгое соответствие данных между экземплярами в любой момент времени, дублирование не становится серьезной проблемой, ведь нам не требуется слишком часто обновлять содержимое хранилища.

Итак, в этой главе вы познакомились с множеством понятий и терминов. Хотя вам не нужно запоминать их все, они могут оказаться полезными при изучении многих других систем, которые мы рассмотрим далее.

[13](#) Ну, практически одинаково. По большей части. За исключением того, что местами есть отличия. В некоторых data-центрах можно встретить несколько поколений оборудования, и иногда мы расширяем data-центры уже после их создания. Но большинство наших data-центров однородны.

[14](#) Bisection bandwidth — пропускная способность между двумя частями сети. Сечение выбирается таким образом, чтобы пропускная способность между частями была минимальной (https://en.wikipedia.org/wiki/Bisection_bandwidth).

[15](#) Некоторые читатели могут быть знакомы с потомком Borg, Kubernetes. Это фреймворк с открытым исходным кодом для управления кластером контейнеров Linux как единой системой. Был запущен Google в 2014 году. Обратите внимание на эти ссылки: <http://kubernetes.io> и [Burns et al., 2016]. Для того чтобы узнать, чем похожи Borg и Apache Mesos, обратитесь к [Verma et al., 2015].

[16](#) Переводы термина неудачны, общепринятого перевода нет. В специализированной литературе часто ограничиваются его оригинальным написанием. — Примеч. пер.

[17](#) Этот протокол сериализации — двоичный, представляет собой независимый от языка и платформы расширяемый механизм сериализации структурированных данных. Для получения более подробной информации перейдите по ссылке <https://developers.google.com/protocol-buffers/>.

[18](#) По нашему мнению, вероятность одновременного сбоя двух задач в нашей среде достаточно мала, чтобы ею можно было пренебречь. В других средах это предположение может оказаться неверным, поскольку сбой может произойти в критических точках — например, в главном коммутаторе или в энергосистеме.

Часть II. Принципы

В этой части рассматриваются *принципы*, на которых основана работа SRE-команд, — шаблоны, типы поведения и проблемные области, влияющие на состав работ, выполняемых в отделе SRE.

Первая глава этой части — глава 3 «Приручаем риски» — самая важная, она предназначена для тех, кто хочет со всех сторон рассмотреть, чем на самом деле занимается отдел SRE. Из нее вы узнаете, как отдел SRE оценивает риски, управляет ими и использует лимит времени недоступности сервиса для того, чтобы объективно принимать решения.

Целевой уровень качества обслуживания (Service Level Objectives, SLO)¹⁹ — еще одно фундаментальное понятие для SRE. В отрасли наблюдается тенденция смешивать разнородные понятия под общим названием «соглашения об уровне обслуживания» (Service Level Agreement, SLA), что препятствует тщательному их изучению. В главе 4 «Целевой уровень качества обслуживания» предпринята попытка отделять показатели от целей, а также рассматривается использование каждого из этих терминов с точки зрения SRE. Помимо этого, в главе 4 вы найдете рекомендации по определению контрольных показателей (метрик), важных для вашего приложения.

Избавиться от утомительной работы — одна из самых важных задач для SR-инженера, и это является темой главы 5 «Избавляемся от рутины». Мы считаем утомительной и рутинной работу однообразную, повторяющуюся изо дня в день, но не дающую конкретных значимых результатов; объем такой работы растет пропорционально росту сервиса.

Мониторинг — это один из основных компонентов успешной работы ПО как в компании Google, так и в других организациях. Если вы не можете наблюдать за сервисом, вы не знаете, что с ним происходит, а если вы не знаете, что происходит, то не можете гарантировать надежность. Прочтите главу 6 «Мониторинг распределенных систем», чтобы получить представление о том, как и за какими компонентами следует наблюдать. В ней вы также найдете описание некоторых приемов мониторинга, не зависящих от реализации.

В главе 7 «Эволюция автоматизации в Google» мы рассмотрим подход SRE к автоматизации, а также примеры ее реализации — как успешные, так и неудачные.

В большинстве компаний на управление выпуском новых версий программ внимание обращают в последнюю очередь. Однако, как вы узнаете из главы 8 «Технологии выпуска ПО», этот аспект не просто критически важен для общей стабильности системы (ведь большинство сбоев бывают вызваны внесением изменений в ПО) — это лучший способ обеспечить уверенность в стабильности и качестве выпускаемого продукта.

Основной принцип эффективной разработки ПО — простота. Утратив это качество, его крайне трудно восстановить. Тем не менее, как гласит старинная поговорка, сложная работающая система начинается с простой работающей системы. В главе 9 «Простота» эта тема рассматривается более детально.

Информация для дальнейшего ознакомления от Google SRE. Безопасное ускорение выпуска продукта — это основной принцип любой организации. В статье *Making Push On Green a Reality* [Klein, 2014], опубликованной в октябре 2014 года, мы показываем, что исключение человека из процесса выпуска продукта снижает операционную нагрузку на SRE, тем самым

увеличивая надежность системы, как бы это ни было парадоксально.

¹⁹ SLO — достаточно сложное и многогранное понятие, и в разных контекстах и у разных авторов можно встретить различные его переводы и толкования. Формально термин SLO считается уже устаревшим, но тем не менее используется очень широко (см. https://en.wikipedia.org/wiki/Service_level_objective). — Примеч. пер.

3. Приручаем риски

Автор — Марк Алвидрес

Под редакцией Кавиты Джулиани

Вы можете подумать, что компания Google старается выпускать на 100 % надежные сервисы, которые никогда не дают сбоев. Однако в определенный момент увеличение надежности приносит сервису (и его пользователям) больше вреда, чем пользы! Предельная надежность имеет свою цену: увеличение стабильности ограничивает скорость разработки новой функциональности и создания новых продуктов, а также повышает их стоимость. В свою очередь, это уменьшает объем функциональности, который команда может позволить себе предложить пользователям. Далее, пользователи, как правило, не замечают разницы между сервисами с высокой и крайне высокой надежностью, поскольку во время их взаимодействия с сервисом значительное влияние оказывают менее надежные компоненты вроде мобильной сети или того устройства, с которым они работают. Проще говоря, пользователь смартфона, надежного на 99 %, не заметит разницы между сервисами с 99,99 % и 99,999 % надежности! Учитывая это, SR-инженеры стараются не просто увеличивать время работы без сбоев, а добиваться баланса вероятности того, что сервис окажется недоступен, и возможности его быстрого развития и эффективного функционирования. Тогда пользователи останутся в целом удовлетворены как функциональностью сервиса, так и его доступностью и производительностью.

Управление рисками

Ненадежные системы быстро утрачивают доверие пользователя, поэтому мы хотим снизить вероятность отказов. Однако опыт показывает, что стоимость системы при повышении надежности растет нелинейно — следующий шаг повышения надежности может стоить в 100 раз больше предыдущего. Можно выделить две составляющие этой стоимости.

- *Стоимость избыточных (резервных) машинных/вычислительных ресурсов* — стоимость, связанная с дополнительным оборудованием, которое, например, позволяет нам отключить систему для запланированных или непредвиденных работ или предоставляет место для контрольных кодов, дающих хотя бы минимальные гарантии сохранности данных.
- *Стоимость упущеных возможностей* — затраты, которые несет организация, когда она выделяет разработчиков для создания служебных систем или отдельных функций, необходимых для снижения рисков, вместо прикладных функций, востребованных конечными пользователями. Эти инженеры больше не работают над новой функциональностью и продуктами для конечных пользователей.

В SRE мы контролируем надежность сервиса в основном через управление рисками. Мы рассматриваем риск как непрерывную функцию (континуум), а также считаем одинаково важным повышать надежность систем Google и устанавливать адекватный уровень устойчивости наших сервисов. Это позволяет нам анализировать стоимость и прибыль, чтобы определить, например, в каких точках кривой зависимости (нелинейной!) рисков поместить сервисы Search,

Ads, Gmail или Photos. Наша цель — явно соотнести риски, которые несет конкретный сервис, с рисками, которые готовы понести бизнес в целом. Мы стремимся сделать уровень доступности сервиса достаточным, но не более необходимого. То есть, когда мы задаемся целью создать сервис, надежный на 99,99 %, мы хотим превзойти этот целевой показатель, но не намного, иначе это не позволяло бы нам добавлять новую функциональность, исправлять технические недоработки или снижать операционные расходы. В некотором роде, мы рассматриваем целевой уровень доступности как минимум и как максимум одновременно. Ключевое преимущество такого подхода — возможность явно и вдумчиво оценивать риски.

Измерение рисков, связанных с сервисом

Стандартная практика Google — определить объективный показатель, позволяющий представить свойства системы, которые мы хотим оптимизировать. Имея цель, мы можем оценить текущий уровень и отслеживать его изменения с течением времени. Для оценки рисков, связанных с сервисом, не совсем понятно, как можно свести все потенциальные факторы в единый показатель. Отказы сервисов могут потенциально иметь множество различных последствий: недовольство пользователя, причинение ему вреда или потерю его доверия; прямое или непрямое снижение выручки; ущерб для бренда или репутации; нежелательное освещение в средствах массовой информации. Очевидно, некоторые из этих последствий трудно измерить. Чтобы упростить эту задачу и иметь возможность распространить ее решение на все типы запускаемых нами систем, мы сосредоточимся на *незапланированных отключениях*.

Для большинства сервисов проще всего представить рискоустойчивость в терминах допустимого уровня незапланированных отключений сервиса. Незапланированные отключения связаны с желаемым уровнем *доступности сервиса*, значение которого обычно выражается количеством девяток: 99,9, 99,99 или 99,999 %. Каждая дополнительная девятка на один порядок приближает нас к 100%-ной доступности. Для работающих систем этот показатель, как правило, вычисляется на основе времени безотказной работы (формула (3.1)). Доступность вычисляется на основе времени безотказной работы и отключений:

	Доступность	Время безотказной работы	
=		Время безотказной работы + .	(3.1)

Используя эту формулу и взяв в качестве рассматриваемого промежутка времени один год, мы можем рассчитать допустимое время в минутах, когда система может быть неработоспособна, сохраняя при этом заданный уровень доступности. Например, система, для которой задана доступность 99,99 %, допустимо будет оставаться неработоспособной до 52,56 минуты за год. В приложении А вы можете увидеть соответствующую таблицу.

Однако для компании Google такой показатель доступности, основанный на времени, обычно не очень полезен, поскольку мы имеем дело с глобальными распределенными системами. Наш метод локализации и изоляции неисправностей позволяет в большинстве случаев сохранять возможность обработки хотя бы части трафика любого из сервисов за счет глобального перенаправления (то есть мы все время как минимум «частично включены»). Поэтому вместо показателей, связанных с временем безотказной работы, мы определяем

доступность на основании *количество успешных запросов*. В формуле (3.2) показано, как определяется такой показатель с использованием «скользящего окна» (то есть доля успешных запросов в рамках одного дня):

	Доступность =	Успешные запросы	.	
		Все запросы		(3.2)

Например, если система обслуживает в день 2,5 миллиона запросов и имеет заданный уровень доступности 99,99 %, для нее допустимо терять из-за сбоев до 250 запросов в день.

В обычном приложении не все запросы равнозначны: ошибка при регистрации нового пользователя отличается от ошибки при фоновой загрузке новых электронных писем. Однако во многих случаях доступность, вычисленная как отношение количества успешных запросов к общему количеству запросов, с точки зрения конечного пользователя служит допустимым приближением к показателю доступности, связанному с незапланированными отключениями.

Определение времени незапланированного отключения через соотношение количества успешных запросов к общему количеству запросов позволяет также использовать этот показатель в системах, которые обычно не обслуживают непосредственно конечных пользователей. Большинство таких «технологических» систем (например, системы пакетной обработки, конвейеры, хранилища и транзакционные системы) имеют строго определенные критерии успешных и неуспешных «единиц работы» (запросов, операций и т.д.). Более того, хотя в этой главе рассматриваются в первую очередь системы, обслуживающие либо пользователей, либо инфраструктуру, многие из описанных подходов и определений можно применить также к «технологическим» системам, лишь слегка их модифицировав.

Например, пакетная обработка, в процессе которой содержимое одной из пользовательских баз данных извлекается, трансформируется и внедряется в общее хранилище для дальнейшего анализа, может запускаться периодически. Используя уровень успешных запросов, определенный через количество записей, обработанных успешно или с ошибкой, мы можем рассчитать и применять показатель доступности, хотя системы пакетной обработки обычно не работают в непрерывном режиме.

Чаще всего мы задаем уровень доступности для квартала, а производительность наших сервисов отслеживаем еженедельно или даже ежедневно. Это позволяет нам управлять сервисом так, чтобы обеспечить высокий уровень его доступности, имея возможность выявлять и исправлять причины значительных изменений состояния сервиса, появление которых со временем неизбежно. Более подробно об этом вы прочитаете в главе 4.

Рискоустойчивость сервисов

Что означает «определить рискоустойчивость сервиса»? В формализованном окружении или для систем, безопасность которых критически важна, понятие рискоустойчивости сервиса обычно включено в характеристики продукта или сервиса. Рискоустойчивость сервисов Google определяется не столь четко.

Для определения рискоустойчивости сервиса SR-инженер должен поработать с владельцами продукта, чтобы преобразовать набор бизнес-целей в четкие требования, которые можно реализовать. В этом случае нас интересуют бизнес-цели, которые непосредственно влияют на производительность и надежность предлагаемого сервиса. На практике такое преобразование удается с трудом. В то время

как у потребительских сервисов зачастую есть владельцы, инфраструктурные сервисы (например, системы хранения или уровень кэширования для протокола HTTP), как правило, определенного владельца не имеют. Рассмотрим оба случая.

Определение рискоустойчивости пользовательских сервисов

Зачастую над нашими пользовательскими сервисами работают команды, которые выступают бизнес-владельцами приложения. Например, сервисы **Веб-поиск** (Search), **Карты** (Google Maps) и **Документы** (Google Docs) имеют собственных менеджеров продукта. Эти менеджеры отвечают за взаимопонимание с пользователями и бизнесом, а также за то, чтобы продукт был успешным на рынке. Когда существует такая «команда продукта», требования к доступности сервиса можно обсудить с ней. При отсутствии такой команды эту роль зачастую играют создающие систему инженеры, знают они об этом или нет.

Существует множество факторов, которые следует принимать во внимание при оценке рискоустойчивости сервисов.

- Какого уровня доступности нужно достичь?
- Как различные типы сбоев влияют на сервис?
- Как мы можем манипулировать стоимостью сервиса, чтобы позиционировать его на кривой зависимости рисков?
- Какие другие показатели сервиса важно иметь в виду?

Целевой уровень доступности

Целевой уровень доступности заданного сервиса Google обычно зависит от выполняемых им функций и от его позиционирования на рынке. В следующем списке приведены вопросы, которые необходимо принять во внимание.

- Какого уровня доступности будут ожидать пользователи?
- Связан ли сервис непосредственно с прибылью (как нашей, так и наших пользователей)?
- Сервис платный или бесплатный?
- Если на рынке есть конкуренты, какого уровня сервисы они предоставляют?
- Сервис предназначен для конечных пользователей или для предприятий?

Рассмотрим требования, предъявляемые к Google Apps for Work. В основном этот сервис используют предприятия, как крупные, так и небольшие. Эти предприятия зависят от сервисов Google Apps for Work (например, **Почта** (Gmail), **Календарь** (Calendar), **Диск** (Drive), **Документы** (Docs)), которые предоставляют им инструменты для повседневной работы сотрудников. Другими словами, сбой в сервисе Google Apps for Work становится сбоем не только для Google, но и для всех предприятий, которые от нас зависят. Для типичного сервиса Google Apps for Work нам следует установить целевой уровень доступности равным 99,9 % (в течение квартала), подкрепив эту цель более высоким целевым уровнем внутренней доступности и контрактом, который допускает штрафы на случай невыполнения внешних показателей.

Для сервиса YouTube, однако, во внимание принимается совершенно другой набор факторов. Когда компания Google приобрела YouTube, нам нужно было определить подходящий для этого сайта уровень доступности. В 2006 году сервис YouTube был ориентирован на пользователей и находился в совершенно другой фазе своего бизнес-цикла, чем Google. Несмотря на то что YouTube уже считался отличным продуктом, он все еще быстро менялся и рос. Мы установили для YouTube более низкие требования к уровню доступности, чем для наших продуктов, используемых на предприятиях, поскольку быстрое развитие функциональности было важнее.

Типы сбоев

Важно также иметь в виду характер сбоев, ожидаемых для каждого сервиса. Насколько устойчивым к отключениям сервиса будет наш бизнес? Что будет хуже для сервиса: постоянные небольшие сбои или периодическое полное отключение сайта? Оба типа сбоев могут при одинаковом абсолютном количестве ошибок иметь совершенно разные последствия для бизнеса.

Разницу между полным и частичным отключением можно естественным образом наглядно продемонстрировать на примере систем, обрабатывающих личные данные. Возьмем приложение по управлению контактами и рассмотрим разницу между сбоем, который не позволяет отрисовать изображения профиля, и ошибкой, приводящей к тому, что контакты одного пользователя показываются другому. В первом случае налицо небольшая ошибка, которую SR-инженеры довольно быстро исправят. Однако во втором случае вероятность выдачи личных данных может легко подорвать доверие пользователей. Поэтому во втором случае приемлемым будет полное отключение сервиса для отладки и (потенциально) очистки.

На другом конце шкалы будут находиться те сервисы Google, для которых допустимо иметь регулярные отключения в запланированное время. Когда-то одним из таких сервисов был Ads Frontend. Он использовался рекламодателями и публикующими рекламу сайтами для создания, конфигурирования, запуска рекламных кампаний и наблюдения за ними. Поскольку большая часть этой работы выполняется в обычное рабочее время, мы решили, что допустимо будет отключать сервис для его обслуживания в заранее запланированные промежутки времени.

Стоимость

Стоимость зачастую является ключевым фактором при определении подходящего целевого уровня доступности сервиса. Сервис Ads особенно удобен в качестве примера, поскольку доля успешных запросов может непосредственно отражаться на прибыли. При определении целевого уровня доступности для каждого сервиса мы задаем следующие вопросы.

- Если нам нужно построить и эксплуатировать подобную систему с доступностью на одну девятку выше заданной, насколько в таком случае вырастет наша прибыль?
- Покрывает ли дополнительная прибыль затраты на достижение такого уровня надежности?

Для того чтобы сделать это компромиссное равенство более конкретным, рассмотрим следующий анализ рентабельности для типового сервиса, где все запросы одинаково важны.

- Предлагаемое улучшение целевого уровня доступности: 99,9 % → 99,99 %.
- Предполагаемое увеличение уровня доступности: 0,09 %.
- Прибыль, получаемая от сервиса: \$1M.
- Прибыль, которую можно будет дополнительно получить от улучшения доступности: $\$1M \times 0,0009 = \900 .

В этом случае, если стоимость улучшения доступности на одну девятку меньше \$900, такое улучшение будет стоить своих денег. Если же эта стоимость больше \$900, она будет превышать потенциальное увеличение прибыли.

Задача по определению целевой доступности может стать сложнее, если у нас нет простой функции, которая бы отражала зависимость между надежностью и прибылью. В таком случае может быть полезной модель фоновых ошибок, которой пользуются интернет-провайдеры. Если взять за основу количество сбоев на стороне конечного пользователя и снизить уровень ошибок настолько, чтобы он стал ниже уровня ошибок пользовательской среды, то эти ошибки можно будет считать непланируемыми помехами в рамках заданного качества соединения с Интернетом. Несмотря на значительные различия между провайдерами и протоколами (например, TCP и UDP, IPv4 и IPv6), измеренные нами типичные уровни фоновых ошибок для различных интернет-провайдеров заключены в пределах от 0,01 до 1 %.

Другие показатели сервисов

Часто бывает полезным выразить рискоустойчивость и через другие показатели. Понимание того, какие показатели важны,

а какие — нет, дает нам некоторую свободу при принятии решений, связанных с оправданным риском.

Наглядным примером может служить время задержки сервиса Ads. Когда компания Google только запустила поисковый сервис, одной из определяющих его особенностей была скорость. Когда мы создали сервис AdWords, отображающий рекламу рядом с результатами поиска, отсутствие задержки при использовании поиска было ключевым требованием к этой системе. Это требование распространяется на разработку каждого поколения систем AdWords и считается неизменным.

Система AdSense выводит на экран контекстную рекламу в соответствии с запросами, получаемыми из скриптов JavaScript, который издатели внедряют в свои сайты. Для нее установлен совершенно другой требуемый показатель величины задержки. Сервис AdSense не должен замедлять отображение сторонней страницы при внедрении в него контекстной рекламы. Конкретный показатель величины задержки будет зависеть от скорости отображения заданной веб-страницы. Это означает, что сервис AdSense может выполнять свою работу на сотни миллисекунд медленнее, чем AdWords.

Более мягкое требование к величине задержки позволило нам пойти на несколько компромиссов при планировании организационно-технического обеспечения (определении количества ресурсов и их местоположения). Это сэкономило нам немало средств. Другими словами, учитывая относительную нечувствительность сервиса AdSense к умеренным изменениям задержек, мы смогли ограничить количество площадок для размещения сервиса и тем самым снизить наши эксплуатационные издержки.

Определение рискоустойчивости инфраструктурных сервисов

Требования к разработке и функционированию инфраструктурных компонентов значительно отличаются от требований к потребительским продуктам. Фундаментальное отличие состоит в том, что инфраструктурные компоненты имеют несколько клиентов, чьи потребности часто не совпадают.

Целевой уровень доступности

Рассмотрим Bigtable [Chang, 2006] – крупную распределенную систему хранилищ структурированных данных. Некоторые пользовательские сервисы работают с данными непосредственно в Bigtable по путям, указанным в запросах пользователя. Время задержки у таких сервисов должно быть небольшим, а надежность – высокой. Другие команды используют Bigtable как репозиторий для данных, которые затем подвергаются анализу в режиме онлайн (например, MapReduce). Для них пропускная способность важнее надежности. Требования к рискоустойчивости для этих двух случаев будут значительно различаться.

Один из подходов к удовлетворению потребностей обоих типов состоит в том, чтобы сделать инфраструктурные сервисы ультранадежными. Учитывая, что эти инфраструктурные сервисы, как правило, потребляют значительные объемы ресурсов, такой подход на практике оказывается слишком затратным. Для того чтобы понять различие в потребностях разных типов пользователей, вы можете взглянуть на то, какое состояние очереди запросов было бы желательно для пользователей каждого типа.

Типы отказов

Пользователи, которым нужно малое время задержки, хотели бы видеть очереди запросов Bigtable пустыми (практически всегда). Это позволит системе обрабатывать каждый особый запрос сразу же после его поступления. (Более того, неэффективное размещение запросов в очередях зачастую является причиной увеличения времени задержки.) Пользователей, заинтересованных в онлайн-анализе, больше интересует пропускная способность системы — они не хотят, чтобы очереди запросов пустовали. Оптимизация пропускной способности требует, чтобы система Bigtable никогда не простаивала в ожидании следующего запроса.

Как мы можете видеть, понятия успешного функционирования и отказа для этих групп пользователей противоположны друг другу. Ситуация, которая оказывается успехом для пользователей, ожидающих получить малое время задержки, становится дефектом для пользователей, связанных с онлайн-анализом.

Стоимость

С точки зрения стоимости это противоречие можно эффективно обойти, разделив инфраструктуру и создав несколько независимых уровней обслуживания. В случае с Bigtable мы можем создать два типа кластеров: кластеры с низкой задержкой и кластеры с высокой пропускной способностью. Первые разрабатываются так, чтобы их могли использовать сервисы, которым нужны малое время задержки и высокая надежность. Чтобы гарантировать, что очереди запросов будут короткими, и обеспечить выполнение более строгих требований по изоляции клиента, системе Bigtable может быть выделено дополнительное количество ресурсов для ослабления конкуренции между их потребителями за счет большей избыточности. С другой стороны, кластеры с

повышенной пропускной способностью могут содержать меньшее количество устройств, чтобы оптимизировать пропускную способность ценой задержек. На практике мы можем гораздо дешевле обеспечивать выполнение таких ослабленных требований и стоимость подобных кластеров составляет 10–50 % от стоимости кластеров с низкой задержкой отклика. Учитывая масштаб системы Bigtable, такая экономия очень быстро становится весомой.

Ключевая стратегия при работе с инфраструктурными сервисами — создание сервисов с явно разграниченным уровнем параметров обслуживания. Это позволяет клиентам находить оптимальные компромиссы рисков и стоимости при построении своих систем. Имея явно разграниченные уровни обслуживания, провайдеры инфраструктуры могут эффективно подчеркнуть разницу между ними, отразив ее в стоимости. Такое разграничение стоимости побуждает клиентов выбирать наиболее дешевый уровень обслуживания, который соответствует их требованиям. Например, Google+ может разместить данные, критически важные для безопасности пользователей, в хранилище данных с высокой доступностью (например, глобально реплициированную SQL-подобную систему вроде Spanner [Corbett et al., 2012]). При этом опциональные данные (которые не являются критически важными и нужны для повышения потребительских качеств) будут храниться в более дешевом, менее надежном, более старом и, соответственно, менее устойчивом хранилище данных (например, хранилище NoSQL с оптимизированной по затратам репликацией вроде Bigtable).

Обратите внимание на то, что мы можем запускать сервисы различного класса, используя одно и то же программное и аппаратное обеспечение. Мы можем обеспечить совершенно разные характеристики сервисов, изменив их параметры:

количество выделяемых ресурсов, уровень избыточности, географические особенности размещения и, что принципиально, конфигурацию инфраструктурного софта.

Пример: фронтенд-инфраструктура

Эти принципы оценки рискоустойчивости могут быть применимы не только для хранилищ, и, чтобы продемонстрировать это, рассмотрим еще один крупный класс сервисов: фронтенд-инфраструктуру Google. В нее входят обратные прокси-серверы и балансировщики нагрузки. Эти системы, помимо всего прочего, поддерживают соединения конечных пользователей (например, TCP-сессии их браузеров). Учитывая критическую роль этих систем, мы проектируем их максимально надежными. В то время как определенный уровень ненадежности пользовательских сервисов может остаться незамеченным, инфраструктурным системам повезло меньше. Если запрос не попадает на фронтенд-сервер прикладного сервиса, он будет полностью потерян.

Мы рассмотрели способы определения рискоустойчивости для пользовательских и инфраструктурных сервисов. Теперь рассмотрим понятие допустимого уровня дефектов в разрезе управления ненадежностью посредством суммарного уровня ошибок.

Обоснование критерия суммарного уровня ошибок

(бюджета ошибок)²⁰

Автор — Марк Рот

Под редакцией Кармелы Куинито

В других главах этой книги рассматривается ситуация, когда между командами разработки и SRE могут возникнуть трения

из-за того, что их работа оценивается разными показателями. Производительность труда команды разработчиков оценивается в основном скоростью выпуска продуктов, что побуждает как можно быстрее писать новый код. В то же время производительность работы SR-инженеров оценивается (что неудивительно) с точки зрения того, насколько надежен сервис, и это побуждает сопротивляться появлению большого количества изменений. Информационная асимметрия между двумя командами еще больше усиливает эти трения. Разработчики продукта имеют более четкое представление о том, сколько времени и сил они потратили на написание и выпуск своего кода, а SR-инженеры лучше ориентируются в надежности сервиса (и в общем состоянии продукта).

Эти трения часто отражаются в разных мнениях о том, насколько старательно нужно разрабатывать продукты. Далее перечислены наиболее типичные спорные моменты.

- *Устойчивость к сбоям.* Насколько хорошо нужно защитить софт от неожиданных ситуаций? Если защита будет слабой, у нас получится уязвимое и нестабильное приложение. Если защита будет сильной, у нас получится приложение, которое никто не захочет использовать (зато оно будет очень надежным).
- *Тестирование.* Опять же, если мало тестировать приложение, это может привести к нежелательным простоям, утечкам личных данных или к другим явлениям, вредящим имиджу. Если тестировать долго, можно потерять свою долю рынка.
- *Частота выпуска новых версий.* Каждое обновление рискованно. Как найти наиболее удачное соотношение затрат времени на снижение этого риска и на выполнение другой работы?

- *Продолжительность тестирования и размер выборки.* Лучше всего тестировать новую версию на небольшом фрагменте данных. Как долго нам следует ждать и насколько большой должна быть выборка?

Существующие команды, как правило, выработали некий неформальный баланс между риском и затрачиваемыми усилиями. К сожалению, вряд ли кто-то из них может с уверенностью сказать, что выработанный ими баланс является идеалом, а не результатом дипломатических талантов инженеров. Принятие таких решений не должно быть продиктовано политикой, страхом или надеждой. (Более того, неофициальный девиз Google SRE гласит: «Надежда — плохая стратегия».) Вместо этого наша цель — определить объективный показатель, с которым будут согласны обе стороны и с помощью которого можно будет направить переговоры в продуктивное русло. Чем больше решение будет основываться на данных и количественных показателях, тем лучше.

Формируем бюджет ошибок

Чтобы строить свои решения на основе объективных данных, обе команды совместно определяют квартальный допустимый суммарный уровень ошибок (бюджет ошибок), основываясь на целевом уровне качества обслуживания (SLO, см. главу 4). Суммарный уровень ошибок — прозрачный и объективный показатель, который определяет, как часто сервис может проявлять ненадежность в пределах одного квартала. Этот показатель позволяет исключить влияние «политических» и эмоциональных факторов на договоренности между SR-инженерами и разработчиками.

Мы пользуемся следующим алгоритмом.

1. Менеджер продукта определяет SLO, задавая тем самым ожидаемую долю времени бесперебойной работы сервера в течение квартала.
2. Реальное текущее время бесперебойной работы измеряется нейтральной третьей стороной — нашей системой мониторинга.
3. Разница между этими двумя числами является запасом (бюджетом) того, насколько ненадежной может быть система в оставшуюся часть квартала.
4. Пока доля времени бесперебойной работы больше времени, заданного SLO (другими словами, пока имеется запас времени недоступности), можно продолжать выпуск новых версий и обновлений продукта.

Например, пусть согласно SLO сервиса он должен успешно обслуживать 99,999 % всех запросов за квартал. Это означает, что лимит времени недоступности сервиса в данном квартале равен 0,001 %. Если из-за какой-либо проблемы ошибки будут возникать при обработке 0,0002 % ожидаемых запросов, будет считаться, что эта проблема поглощает 20 % квартального бюджета ошибок.

Преимущества

Основное преимущество описанного подхода в том, что он дает хороший стимул командам разработчиков и SR-инженеров сконцентрироваться на поиске баланса между инновациями и надежностью.

Во многих продуктах приведенный алгоритм применяется для управления скоростью выпуска новых версий: до тех пор пока заданные целевые показатели обеспечиваются и задачи выполняются, можно продолжать выпускать обновления. Если нарушения требований SLO происходят довольно часто и превышают заданные лимиты, выпуск новых версий временно приостанавливается. При этом дополнительные ресурсы направляются в тестирование и разработку, что служит повышению устойчивости системы, ее производительности и т.д. Существуют и более гибкие и эффективные подходы, чем просто «включение/выключение»²¹: например, релизы можно задерживать или откатывать, если лимит времени недоступности почти исчерпан.

Например, если команда разработки хочет сэкономить время на тестировании или ускорить выпуск новых версий, а команда SRE этому сопротивляется, решение основывается на текущем суммарном уровне ошибок. Если запас велик, разработчики могут позволить себе больше рисковать. Если же лимит практически исчерпан, разработчики сами будут переориентироваться на тестирование или задерживать выпуск, поскольку они не заинтересованы «выйти за бюджет», полностью затормозив выпуски. В результате команда разработчиков начинает больше следить за своей работой. Они знают величину лимита и могут управлять собственными рисками. (Конечно, это возможно при условии, что команда SRE имеет право остановить выпуск новых продуктов, если требования SLO нарушены.)

Что происходит, если снижение измеренного SLO вызвано отключением сети или сбоем в дата-центре? Такие события также расходуют бюджет ошибок. В результате количество выпущенных новых версий до конца квартала может

уменьшиться. Вся команда поддержит подобное снижение, поскольку за работу системы отвечает каждый сотрудник.

Использование суммарного уровня ошибок также позволяет наглядно отобразить цену слишком высоких показателей надежности: снижение гибкости и замедление инноваций. Если у команды возникают проблемы при выпуске новой функциональности, они могут пойти на ослабление требований SLO (тем самым увеличивая доступный бюджет ошибок), чтобы ускорить обновления.

Основные итоги

Управление надежностью сервиса в основном заключается в управлении рисками, и это может быть связано с издержками.

Практически никогда не стоит планировать 100%-ный уровень надежности: во-первых, его невозможно достигнуть, а во-вторых, пользователь может не нуждаться в таком показателе или не заметить его. Необходимо соотнести назначение и особенности сервиса с теми рисками, которые бизнес готов на себя взять.

Введение допустимого суммарного уровня ошибок (бюджета ошибок) стимулирует команды SRE и разработчиков и подчеркивает их общую ответственность за систему. Он позволяет проще принимать решения о скорости выпуска новых версий и эффективно сглаживать конфликт между участниками проекта в случае сбоев, а также дает возможность нескольким командам без лишних эмоций

приходить к одинаковым выводам о рисках при выпуске продукта.

[20](#) Первая версия этого раздела появилась в журнале ;login: (август 2015 года, выпуск 40, № 4).

[21](#) Этот прием также известен как bang/bang control — более подробную информацию вы найдете по ссылке https://en.wikipedia.org/wiki/Bang–bang_control.

4. Целевой уровень качества обслуживания

Авторы — Крис Джоунс, Джон Уилкс и Нейл Мёрфи при участии Коди Смита

Под редакцией Бетси Бейер

Невозможно правильно управлять сервисом (не говоря уже о том, чтобы делать это хорошо), не понимая, какие аспекты поведения действительно важны для него и как их измерить и оценить. Для этого мы хотели бы установить *уровень качества обслуживания* и обеспечить его для наших пользователей независимо от того, используют они внутренний API или общедоступный продукт.

В процессе работы мы прислушиваемся к интуиции и ориентируемся на свой опыт и понимание пожеланий пользователей. В итоге мы стараемся определить *показатели уровня качества обслуживания* (service level indicators, SLIs), а также *целевые показатели* (objectives, SLOs) и *соглашения* (agreements, SLAs)²². Эти количественные характеристики описывают важные базовые свойства продуктов, их необходимые значения и наши действия в том случае, если мы не можем предоставить ожидаемый уровень обслуживания. В конце концов, подбор подходящих показателей помогает принимать правильные решения, если что-то идет не так, а также дает команде SRE уверенность в исправности сервиса.

В этой главе описывается фреймворк, используемый нами для формирования показателей, их выбора и анализа. Большая часть объяснений осталась бы абстрактной без каких-либо примеров, поэтому для иллюстрации основных моментов мы будем использовать сервис Shakespeare, описанный в подразделе «Shakespeare: пример сервиса» на с. 56.

Терминология для уровня качества обслуживания

Скорее всего, многие читатели знакомы с концепцией SLA, но термины *SLI* и *SLO* также заслуживают четкого определения. В повседневной речи термин *SLA* используется слишком часто и имеет несколько значений в зависимости от контекста. Для ясности мы предпочитаем разделять эти значения.

Показатели

SLI — это *показатель (индикатор) уровня качества обслуживания*, четко определенное числовое значение конкретной характеристики предоставляемого обслуживания.

Для большинства сервисов ключевым *SLI* является *время отклика*, или *латентность запросов*, — время, которое требуется для того, чтобы вернуть ответ на запрос. В качестве *SLI* могут также использоваться *уровень* (или *частота*) ошибок, который часто выражается как процент от общего количества запросов, и *пропускная способность системы*, чаще всего измеряемая в запросах в секунду. Результаты измерений обычно агрегируются: например, первичные данные, собранные в пределах «окна измерения», затем усредняются, приводятся к процентному уровню, или к процентилю.

В идеале *SLI* непосредственно отражает интересующий нас уровень качества обслуживания, но иногда нам доступны только приближенные данные, поскольку желаемые точные измеренные показатели трудно получить или интерпретировать. Например, наиболее релевантным для пользователя показателем часто бывает время отклика на стороне клиента, но измерить задержку можно только на сервере.

Еще один *SLI*, имеющий значение для отдела SRE, — это *доступность*: доля времени, когда сервис можно использовать.

Обычно она определяется как доля успешных запросов, иногда ее называют *выработкой* (*yield*). Аналогично важная для систем хранения данных характеристика *долговечность* (*durability*) — это вероятность того, что данные будут сохранены в течение длительного промежутка времени. Хотя 100%-ной доступности достичь невозможно, можно достигать значения, близкого к 100 %. В нашей отрасли высокие значения доступности выражаются количеством девяток, использованных в записи процента доступности. Например, уровни доступности 99 % и 99,999 % могут называться «две девятки» и «пять девяток» соответственно. В частности, текущее значение доступности Google Compute Engine равно трем с половиной девяткам — 99,95 %.

Целевые показатели

SLO — это *целевой уровень качества обслуживания*: целевое значение или диапазон значений, измеряемые с помощью SLI. Естественная структура SLO выглядит как $SLI \leq \text{целевое значение} \leq \text{нижняя граница} \leq SLI \leq \text{верхняя граница}$. Например, мы можем решить, что сервис Shakespeare будет возвращать результат «быстро», и укажем, что среднее время отклика должно быть равно 100 миллисекундам или меньше.

Выбрать подходящий SLO нелегко. Начнем с того, что вы не всегда можете определить конкретное значение! Например, для запросов HTTP, поступающих к вашему сервису из внешнего мира, показатель «запросы в секунду» (queries per second, QPS) естественным способом определяется потребностями ваших пользователей и вы не можете сами установить его.

С другой стороны, вы можете сказать: «Хочу, чтобы среднее время отклика было меньше 100 миллисекунд»²³. Постановка

подобной цели может, в свою очередь, мотивировать вас написать фронтенд, учитывающий варианты поведения с низкой задержкой, или приобрести соответствующее оборудование.

Опять же это вопрос несколько более тонкий, чем кажется на первый взгляд. Эти два показателя — QPS и время отклика — могут быть связаны друг с другом: чем больше QPS, тем больше задержки. Производительность некоторых сервисов снижается по достижении определенного порога нагрузки.

Определение и объявление SLO задает уровень ожиданий, относящихся к работе сервиса. Это может уменьшить количество необоснованных жалоб к владельцам сервиса по поводу, например, его медленной работы. Не имея явного SLO, пользователи зачастую руководствуются собственными представлениями о желаемой производительности, которые могут не совпадать с соответствующими представлениями разработчиков сервиса. Это может привести как к излишнему доверию к приложению, когда пользователь считает его более доступным, чем на самом деле (как это случилось с Chubby: см. ниже врезку «Плановые отключения глобального сервиса Chubby»), так и к недостаточному доверию, когда потенциальные пользователи считают систему менее надежной, чем она есть.

Плановые отключения глобального сервиса Chubby

Автор — Марк Алвидрес

Chubby [Burrows, 2006] — это сервис блокировок Google для слабо связанных распределенных систем. В глобальном случае мы распределяем его копии (реплики) так, чтобы все они находились в разных географических регионах. С

течением времени мы обнаружили, что сбои в этой глобальной системе периодически приводят к отключениям сервиса, что зачастую влияет и на конечных пользователей. Как оказалось, сами по себе отключения Chubby происходили настолько редко, что владельцы других сервисов начали использовать обращения к нему, вообще не учитывая возможность его отключения. Высокая надежность сервиса сформировала ложное чувство безопасности, и в результате другие сервисы не могли работать корректно при недоступности Chubby, хоть и происходило это редко.

Для этой проблемы было придумано интересное решение: SR-инженеры обеспечивают в целом соответствие глобального сервиса Chubby своим SLO и, возможно, лишь незначительное превышение установленных лимитов. Если в каком-то квартале реальное отключение сервиса не опускало уровень доступности ниже целевого, создавался искусственный сбой, который целенаправленно отключал систему. Такой сбой позволял избавляться от некорректно реализованных зависимостей от Chubby вскоре после их появления. Это заставляет владельцев сервисов считаться с реальными свойствами распределенных систем, причем, скорее всего, еще на ранних стадиях разработки¹.

Соглашения²⁴

Наконец, SLA — это *соглашения* об уровне обслуживания, явный или неявный контракт с вашими пользователями,

включающий в себя последствия, которые влечет за собой соответствие (или несоответствие) требованиям SLO. Последствия лучше всего видны, когда затрагиваются финансы — скидки или штрафы, но они могут принимать и другие формы. Простой способ понять разницу между SLO и SLA заключается в том, чтобы задать себе вопрос: «Что случится, если требования SLO не соблюдаются?» Если явных последствий не существует, то, скорее всего, перед вами только SLO²⁵.

Отдел SRE обычно не участвует в создании SLA, поскольку SLA тесно связаны с бизнесом и относящимися к продукту управленческими решениями. Однако отдел SRE помогает избегать последствий при нарушении требований SLO. Он также может помочь определить SLI: очевидно, должен существовать некий способ измерить показатели SLO, чтобы не возникло разногласий.

Google Search — это пример важного и ответственного сервиса, который не имеет общедоступных SLA: мы стремимся сделать для всех использование сервиса Search максимально эффективным и беспроблемным, но мы не подписывали контракт со всем миром. Тем не менее даже в этом случае могут возникнуть определенные последствия, если сервис окажется недоступен — это грозит ударом по нашей репутации, а также снизит прибыль от рекламы. Многие другие сервисы Google вроде Google for Work имеют явные SLA. Но вне зависимости от того, имеет ли конкретный сервис явные SLA, будет полезно определить для него SLI и SLO и использовать их для управления сервисом.

Хватит теории — приступим к практике!

Показатели на практике

Итак, мы обосновали *необходимость* выбора подходящих показателей для сервиса. Как определить, что показатели имеют значение для вашего сервиса или системы?

Что важно для вас и ваших пользователей

Вы не должны использовать в качестве SLI каждый доступный показатель своей мониторинговой системы. Понимая, что именно ваши пользователи хотят от системы, вы сможете выбрать разумное количество показателей. Выбор слишком большого их количества приведет к тому, что вам будет трудно уделить достаточное внимание тем из них, кто действительно важен, а выбор слишком малого количества — к тому, что некоторые варианты поведения системы останутся не охваченными измерениями. Чтобы оценить исправность системы, нам, как правило, хватает небольшого набора индикаторов.

С точки зрения релевантности тех или иных SLI, сервисы делятся на несколько больших категорий.

- Для *пользовательских сервисов* вроде Shakespeare обычно важны *доступность, время отклика и пропускная способность*. Можем ли мы ответить за запрос? Сколько времени для этого потребуется? Сколько запросов мы можем обработать?
- Для *систем хранения* чаще делается акцент на *времени отклика, доступности и долговечности*. Сколько времени нам потребуется для того, чтобы считать или записать данные? Можем ли мы получить доступ к данным всегда, когда это потребовалось? Сохраняются ли в системе данные на тот момент, когда они нам понадобятся? Более подробно эти вопросы обсуждаются в главе 26.

- Для систем обработки больших объемов данных вроде конвейеров важны пропускная способность и общая задержка обработки данных. Сколько данных обрабатывается прямо сейчас? Как долго данные находятся внутри системы? (Некоторые конвейеры также могут иметь целевые значения задержки на отдельных этапах обработки.)
- Для всех систем должна быть важна корректность. Был ли корректен выданный ответ, были ли корректны полученные данные, был ли корректен выполненный анализ? Корректность — это важный показатель исправности системы, даже несмотря на то, что зачастую он является свойством данных, а не инфраструктуры как таковой, и поэтому обычно не входит в зону ответственности отдела SRE.

Сбор показателей

Значения многих показателей наиболее удобно собирать на стороне сервера с помощью мониторинговой системы типа Borgmon (см. главу 10) или Prometheus. Можно также периодически анализировать журналы — например, при определении соотношения количества ответов HTTP 500 к общему количеству запросов. Однако для некоторых систем требуются инструменты, позволяющие собирать данные *на стороне клиента*. Из-за отсутствия такого рода данных можно пропустить целый ряд проблем, влияющих на пользователей, но не отражающихся в показателях на стороне сервера. Например, если вы сосредоточитесь только на значении времени отклика в серверной части поискового сервиса Shakespeare, то можете не заметить большую задержку, связанную с проблемами в коде JavaScript. В этом случае

наилучшим местом измерения задержки готовности страницы будет браузер пользователя.

Агрегирование

Для простоты и удобства использования первичные данные обычно агрегируются. Это нужно делать осторожно.

Кажется, что для некоторых показателей вроде количества обрабатываемых запросов в секунду достаточно просто взять первичные данные, но даже такие измерения неявно агрегируют данные в пределах окна измерения. Мы получаем данные каждую секунду или определяем среднее количество запросов в минуту? Во втором случае мы можем упустить всплески интенсивности поступления запросов, которые делятся всего несколько секунд.

Рассмотрим систему, которая обрабатывает по 200 запросов каждую четную секунду и 0 запросов в остальное время. Средняя нагрузка такой системы будет такой же, как и нагрузка системы, обрабатывающей каждую секунду по 100 запросов, но ее *мгновенная* нагрузка в два раза выше *средней*. Аналогично заманчивым может казаться вычисление среднего времени отклика, но такой подход не раскрывает одну важную деталь: есть вероятность, что большая часть запросов обрабатывается достаточно быстро, и лишь некоторые запросы, находящиеся в удлиненном «хвосте», обрабатываются гораздо медленнее.

Большинство показателей лучше рассматривать не как средние значения, а как *распределения*. Например, если говорить о показателе времени отклика, то одни запросы будут обрабатываться быстрее, а другие — дольше, причем иногда намного дольше. Простой подсчет среднего значения может скрыть эти «хвостовые» задержки, а также их изменения. На рис. 4.1 показан пример: несмотря на то что обычно запрос

обрабатывается примерно 50 миллисекунд, 5 % запросов обрабатываются в 20 раз медленнее! Соответственно, процессы мониторинга и оповещения, учитывающие только усредненные значения задержки, не покажут колебаний в течение дня, хотя в действительности в «хвостовых» задержках заметны значительные изменения (верхняя линия на рисунке).

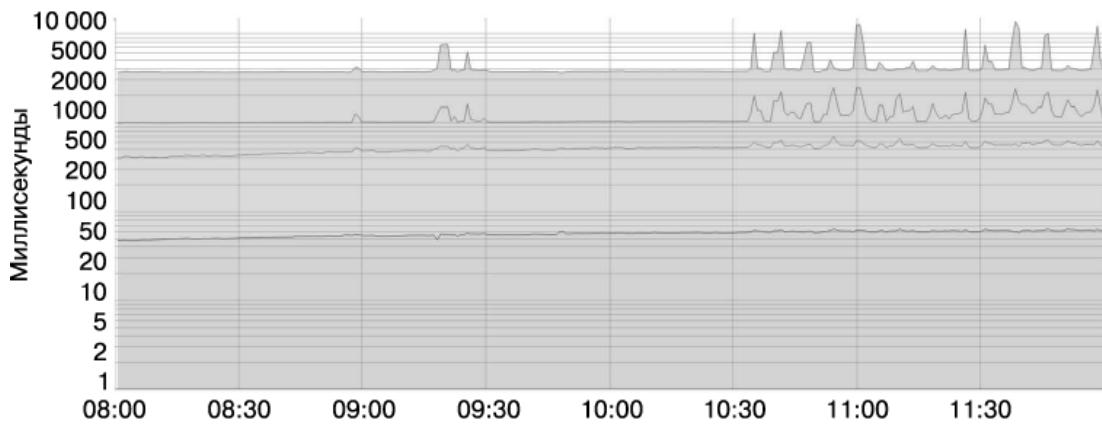


Рис. 4.1. Показаны 50-й, 85-й, 95-й и 99-й процентили задержки системы. Обратите внимание на то, что ось Y имеет логарифмическую шкалу

Использование процентилей в качестве показателей позволяет рассмотреть форму распределения и его различные характеристики: процентиль высокого порядка, например 99-й или 99,9-й, показывает вам предположительно худшие значения, а 50-й процентиль (также известный как медиана) позволяет выделить типичные ситуации. Чем выше разброс значений времени отклика, тем больше будет влиять на пользователя поведение удлиненного «хвоста» распределения, и этот эффект усугубляется при высоких нагрузках из-за особенностей поведения очередей. Исследования, проводимые среди пользователей, показывают, что люди обычно предпочитают чуть более медленную систему, чем имеющую большой разброс времени отклика. По этой причине отдел SRE интересуют только значения высоких процентилей: если

пользователя устроит поведение системы на 99,9-м процентиле, то оно устроит его и в обычном случае.

О статистических заблуждениях

Обычно мы предпочитаем работать с процентилями, а не со средним арифметическим набора значений. Это позволяет нам рассмотреть данные в удлиненном «хвосте» распределения, которые существенно отличаются от средних значений. Поскольку вычислительные системы имеют искусственную природу, значения показателей обычно несимметричны. Например, ни на один запрос нельзя получить ответ менее чем за 0 миллисекунд, а при задержке более 1000 миллисекунд выполнение запроса считается неуспешным. В результате нельзя ожидать, что среднее арифметическое и медиана будут иметь равные или хотя бы близкие значения!

Мы не торопимся предполагать, что данные будут распределены по нормальному закону, пока сами не убедимся в этом. А вдруг стандартные приближения и интуиция нас подведут? Например, если распределение не соответствует ожиданиям, то процесс, который должен выполняться при появлении выбросов в показателях (скажем, перезапуск сервера в ответ на недопустимое увеличение времени обработки запросов), может запускаться либо слишком часто, либо слишком редко.

Стандартизация показателей

Мы рекомендуем стандартизировать определение распространенных показателей, чтобы вам не нужно было каждый раз заново их переопределять. Любую характеристику, соответствующую стандартному шаблону определения, в спецификации конкретного показателя можно опустить. Приведем примеры.

- Интервалы агрегирования: «Среднее значение за минуту».
- Регионы агрегирования: «Все задачи в кластере».
- Частота выполнения измерений: «Каждые 10 секунд».
- Какие запросы нужно включить в выборку: «Запросы HTTP GET, полученные в результате мониторинга методом черного ящика».
- Способ получения данных: «Данные получены путем наблюдения; измерения выполнены на стороне сервера».
- Задержка при доступе к данным: «Время до получения последнего байта».

Для того чтобы сэкономить время, вы можете создать набор повторно используемых шаблонов определений для каждого распространенного показателя. Они также помогут другим понять, что означает каждый конкретный показатель.

Целевые показатели на практике

Подумайте (или узнайте!) о том, что важно для ваших пользователей, а не о том, что вы можете измерить. Зачастую

то, что важно для пользователей, трудно или невозможно измерить, поэтому вам придется хотя бы приблизительно оценить их потребности. Однако если вы просто возьмете в качестве показателей то, что легко измерить, вы получите менее полезные SLO. Таким образом, мы выяснили, что лучше двигаться от желаемых целей к конкретным показателям, чем сначала выбирать показатели и лишь затем ставить цели.

Определение целей

Для максимальной прозрачности SLO должны содержать информацию о том, как их измерять, а также условия, при которых они будут действительны. Например, мы можем указать следующее (вторая строка похожа на первую, но здесь мы использовали стандартные умолчания для определений показателей из предыдущего раздела, чтобы не писать лишнюю информацию):

- 99 % (в среднем за одну минуту) вызовов RPC Get будут завершены менее чем за 100 миллисекунд (измеряется для всех бэкенд-серверов);
- 99 % вызовов RPC Get будут завершены менее чем за 100 миллисекунд.

Если вам важна форма кривых производительности, можете указать в SLO несколько целевых значений:

- 90 % вызовов RPC Get будут завершены менее чем за 1 миллисекунду;
- 99 % вызовов RPC Get будут завершены менее чем за 10 миллисекунд;

- 99,9 % вызовов RPC Get будут завершены менее чем за 100 миллисекунд.

Если у вас есть пользователи с неоднородной загруженностью (вроде конвейера обработки массивов данных), для которых важна пропускная способность, и интерактивный клиент, для которого важно время отклика, то для каждого вида рабочей нагрузки лучше задать отдельные целевые показатели качества:

- 95 % вызовов RPC Set, для которых важна пропускная способность, будут завершены менее чем за 1 секунду;
- 99 % вызовов RPC Set, для которых важна величина задержки и которые имеют нагрузку меньше 1 Кбайт, будут завершены менее чем за 10 миллисекунд.

Настаивать на том, чтобы SLO соблюдались 100 % времени, нереально и нежелательно: это может привести к снижению скорости развертывания и внедрения изменений, а также потребовать применения дорогих или излишне консервативных решений. Вместо этого лучше установить допустимый суммарный уровень ошибок — уровень допускаемого несоответствия SLO — и ежедневно или еженедельно отслеживать его выполнение. А ваши руководители, возможно, захотят также видеть ежемесячный или ежеквартальный отчеты по использованию бюджета ошибок. (Суммарный уровень ошибок — это SLO, отражающий соответствие другим SLO!)

Уровень несоответствия SLO — это полезный индикатор исправности сервиса с точки зрения пользователя. Ежедневное или еженедельное отслеживание SLO (и их нарушений) полезно

тем, что вы сможете увидеть характер изменений и узнать о потенциальных проблемах еще до того, как они произойдут. Ежемесячный или ежеквартальный отчеты могут пригодиться руководству.

Выбор целевых показателей

Выбор целевых показателей (SLO) — это не совсем техническая задача, поскольку в выбранных SLI и SLO (и, возможно, SLA) должны быть также отражены интересы продукта и бизнеса. Аналогично может потребоваться отдать предпочтение тем или иным характеристикам продукта из-за ограничений, накладываемых кадровым составом, сроками выхода продукта на рынок, доступностью аппаратных средств и финансирования. Отдел SRE должен быть частью этого диалога, он участвует в оценке рисков и целесообразности различных вариантов. Мы усвоили несколько уроков, которые могут помочь вам принять более продуктивное решение.

- *Не выбирайте цель, основываясь на текущей производительности.* Конечно, очень важно понимать характеристики и ограничения системы. Но принятие целевых значений без их критического осмысления может завести вас в тупик: обеспечение соответствия системы заданным показателям будет требовать героических усилий, но ее невозможно будет улучшить без значительной перестройки.
- *Не усложняйте.* Сложные варианты агрегирования значений SLI могут маскировать изменения в производительности системы. Кроме того, их труднее обрабатывать.
- *Не гонитесь за абсолютом.* Очень заманчиво потребовать систему, которая может «бесконечно» масштабировать свою

нагрузку и которая «всегда доступна». Но в жизни такие требования оказываются нереальными. Даже систему, которая лишь приближается к идеальным значениям, нужно будет очень долго разрабатывать и строить. Эксплуатировать ее также будет очень дорого — и вполне возможно, что она окажется лучше, чем того хотят пользователи.

- *Имейте минимальное количество SLO.* Выберите те SLO, которых будет достаточно для получения полной информации о характеристиках системы. Обосновывайте необходимость выбранных SLO: если вы не можете победить в дискуссии о приоритетах развития, процитировав нужный SLO, то вам, возможно, лучше избавиться от него²⁶. Однако не все характеристики продукта подчиняются SLO: с их помощью, например, трудно выразить уровень «восхищения пользователя».
- *Идеал может подождать.* Со временем, по мере изучения поведения системы, вы можете совершенствовать определения SLO и целевых значений. Сначала лучше задать в качестве целевого примерное значение, которое вы сможете уточнить позже, а не устанавливать сразу четкое значение и затем снижать его, обнаружив его недостижимость.

SLO могут — и должны — стать главным двигателем при определении приоритетных задач для отдела SRE и разработчиков продукта, поскольку они отражают то, что важно для пользователей. Хорошо построенные SLO — это полезный весомый фактор воздействия на команду разработки. Плохо продуманные SLO могут привести к пустой

трате рабочего времени, если команда прикладывает героические усилия для обеспечения соответствия завышенным SLO, или для выпуска некачественного продукта, если SLO оказались недостаточно жесткими. SLO — это мощный рычаг, и пользоваться им нужно с умом.

SLO как инструмент управления

SLI и SLO являются ключевыми звеньями в цепочке управления системами.

1. Отслеживайте и измеряйте SLI системы.
2. Сравните SLI с SLO и решите, требуется ли вмешательство.
3. Если вмешательство необходимо, определите, что должно измениться или произойти для того, чтобы цель была достигнута.
4. Действуйте.

Например, на шаге 2 обнаруживается рост времени отклика и если ничего не предпринимать, то через несколько часов SLO будут нарушены. Тогда шаг 3 может включать в себя проверку предположения о перегруженности процессоров сервера, а также принятие решения о выделении дополнительных ресурсов с целью перераспределения нагрузки. Без помощи SLO вы бы не узнали, что нужно что-то сделать, и не могли бы определить, когда это делать.

SLO формируют ожидания

Опубликованные SLO устанавливают планку ожиданий относительно характеристик системы и ее поведения. Пользователи сервиса (в том числе потенциальные) зачастую хотят знать, на что они могут рассчитывать, чтобы понять, подходит ли он для них. Например, экономная и эффективная сверхнадежная система с низкой доступностью вряд ли заинтересует команду, планирующую создавать сайт для размещения фотографий, зато она отлично подойдет для системы управления записями архива.

Для того чтобы ожидания пользователей были реалистичными, вы можете воспользоваться следующими тактиками (любой из них или обеими сразу).

- *Имейте запас прочности.* Применение внутренних SLO, более жестких по сравнению с SLO, показанными пользователям, дает вам пространство для маневра, чтобы отреагировать на хронические проблемы до того, как они станут заметны извне. «Буферная зона» значений SLO позволяет также создавать другие реализации продукта, в которых производительность приносится в жертву другим характеристикам — например, стоимости или простоте обслуживания — и пользователи при этом не будут разочарованы.
- *Избегайте перевыполнения.* Пользователи в своих действиях исходят из реально предлагаемых, а не из заявляемых характеристик. В частности, это относится к инфраструктурным сервисам. Если производительность вашего сервиса гораздо выше, чем это указано в SLO, пользователи станут полагаться на текущее значение производительности. Вы можете избежать чрезмерной зависимости от качества системы, периодически намеренно отключая ее (например, мы время от времени отключали

сервис Chubby, поскольку иначе уровень его доступности был избыточным)²⁷, ограничивая количество обрабатываемых запросов или проектируя систему так, чтобы она работала с одинаковой скоростью при разных нагрузках.

Понимание того, насколько система соответствует предъявляемым к ней требованиям, поможет решить, следует ли вкладываться в повышение ее производительности, доступности и устойчивости. В противном случае, если сервис работает достаточно хорошо, время ваших сотрудников, возможно, лучше потратить на другие задачи, например на устранение старых недоработок, добавление новых функций или создание новых продуктов.

Соглашения на практике

Создание SLA требует от команд, отвечающих за ведение бизнеса и юридические вопросы, определить соответствующие меры и штрафы, которые будут применены в случае нарушения соглашений. Роль отдела SRE заключается в том, чтобы помочь им понять реалистичность и сложность выполнения SLO, содержащихся в SLA. Большая часть советов по созданию SLO применима и к созданию SLA. Лучше консервативно подойти к тому, что вы рекламируете потребителям, поскольку чем шире будет круг ваших пользователей, тем сложнее будет в дальнейшем изменить или удалить SLA, которые оказались необоснованными или трудновыполнимыми.

²² Два различных исходных термина (indicator и objectives) приходится переводить одним и тем же словом «показатель», но во втором случае с добавлением «целевой». Нужно помнить: показатель — наблюдаемая количественная характеристика, а целевой показатель — характеристика более высокого порядка,

определяемая через желаемые значения показателей и успешность/неуспешность их выполнения. — *Примеч. пер.*

[23](#) Сто миллисекунд — это произвольное значение, но, как правило, чем меньше время отклика, тем лучше. Эти причины помогают поверить в то, что «быстрее» лучше, чем «медленнее», и что величина задержки свыше некоторого значения может оттолкнуть пользователей. Обратитесь к статье Speed Matters [Brutlag, 2009] для получения более подробной информации.

[24](#) Заметим, что в приведенном примере источником проблем был не Chubby, а зависящие от него прикладные сервисы: их некачественная реализация в большинстве случаев компенсировалась качеством инфраструктурного сервиса, и лишь когда эта компенсация не срабатывала, происходил реальный сбой. Радикальные меры (искусственные «аварии») потребовались для того, чтобы разработчики зависимых сервисов выполняли свою работу с должным качеством (а руководство согласилось оплатить эту дополнительную работу). — *Примеч. пер.*

[25](#) Большинство людей имеют в виду SLO, когда говорят об SLA. Существует один хороший показатель: если кто-то говорит о «нарушении SLA», они говорят о несоответствии требованиям SLO. Реальное нарушение SLA может привести к судебному иску о нарушении контракта.

[26](#) Если вы вообще не можете победить в дискуссии об SLO, вам, возможно, не стоит привлекать службу SRE для разработки данного продукта.

[27](#) Метод «внедрения отказов» [Bennett, Tseitlin, 2012] предназначен для других целей, но он также может помочь в формировании уровня ожиданий.

5. Избавляемся от рутины

Автор — Вивек Рай

Под редакцией Бетси Байер

Если во время выполнения стандартных операций в работу системы должен вмешаться человек-оператор, у вас есть ошибка.

Из определения «нормальных изменений»²⁸ при эволюции систем. Карла Гейссера, Google SRE

Как SRE-инженеры, мы хотели бы заниматься долгосрочными инженерными проектами, а не выполнять операционную работу. Поскольку термин «операционная работа» имеет множество значений, мы используем отдельное слово — «рутина» (в оригинале *toil*. — Примеч. пер.).

Что такое рутинна

Рутинна — это не просто «работа, которую мне не нравится делать». Этот термин также не описывает административную или «грязную» работу. Разные люди предпочитают различные виды работ, и, например, некоторым даже нравится ручная, однообразная работа. Существует также административная работа, которую выполнять необходимо, но ее нельзя назвать рутинной — это дополнительная служебная нагрузка. Служебная нагрузка — это своего рода «накладные расходы», то есть то, что не связано непосредственно с обслуживанием работающего сервиса: митинги и обсуждения, определение целей и их приоритетов²⁹, а также сниппеты³⁰ и бумажная работа для кадровой службы. Результаты «грязной» работы иногда могут

представлять ценность в течение длительного времени, и тогда такую работу также нельзя считать рутиной. «Грязной» работой могут быть очистка конфигурации системы оповещения для вашего сервиса и удаление мусора, но это не рутинна.

Что же такое рутинна? Рутинна — это ручная, однообразная, поддающаяся автоматизации оперативная работа, связанная с поддержкой работающего сервиса. Ее результаты не имеют ценности в перспективе, а трудоемкость растет линейно по мере роста сервиса. Не каждая на первый взгляд рутинная задача имеет все эти признаки, но чем больше она соответствует следующим описаниям, тем более вероятно, что она является рутинной.

- *Ручная.* Такой вид работы включает в себя, например, ручной запуск сценария, автоматизирующего выполнение какой-то задачи. Запуск такого сценария позволит достичь результатов быстрее, чем поочередное выполнение каждого его шага. Но при этом реальное время, которое человек потратит на запуск (а не время выполнения сценария), все еще будет считаться временем, потраченным на рутинную работу.
- *Однообразная.* Если вы решаете какую-то задачу в первый раз (или даже во второй), такая работа еще не является рутинной. Рутинна — это работа, которую вы выполняете раз за разом. Если же вы устраняете какую-то новую проблему или находите новое решение, это не рутинна.
- *Автоматизируемая.* Если машина сможет выполнить работу так же хорошо, как и человек, или же систему можно спроектировать так, чтобы вообще избавиться от необходимости выполнять эту работу, такая работа является рутинной. Если в процессе выполнения задачи человек

должен принять какое-то решение, высока вероятность того, что это не рутинная работа^{[31](#)}.

- *Оперативная.* Рутина — это реагирование на происходящие события, а не стратегическое планирование и действия на опережение. Обработка оповещений системы мониторинга — рутина. Мы, возможно, никогда не сможем полностью избавиться от такой работы, но мы стараемся минимизировать ее.
- *Результаты не имеют ценности в перспективе.* Если после того, как вы выполните задачу, ваш сервис останется в том же состоянии, что и был, задача, скорее всего, рутинная. Если в результате ваш сервис улучшился, то задача, возможно, не является рутиной, даже если для этого пришлось выполнять какую-то «грязную» работу, например разбираться с устаревшим кодом.
- *Трудоемкость растет пропорционально росту сервиса.* Если объем работы, необходимый для решения задачи, растет по мере роста сервиса, увеличения объема трафика или количества пользователей, работа, возможно, является рутиной. Идеально управляемый и спроектированный сервис может вырасти как минимум на один порядок и при этом не потребовать дополнительной работы, помимо разового добавления ресурсов.

Почему лучше иметь меньше рутинной работы

Наш подход к организации SRE ставит целью удерживать такой уровень операционной работы (то есть рутинны), чтобы SR-инженеры тратили на нее менее 50 % своего времени. Минимум 50 % времени каждый SR-инженер должен тратить

на инженерную работу, которая либо снизит количество рутины в будущем, либо добавит сервису новую функциональность. Разработка новой функциональности обычно бывает направлена на повышение надежности, производительности или эффективности использования, что обычно косвенно снижает количество рутины.

Мы определили этот порог 50 % как целевой показатель, потому что объем рутины, если оставить ее без контроля, имеет тенденцию увеличиваться, и в результате может быстро поглотить все рабочее время каждого участника. Работы по снижению объема рутины и масштабированию сервисов и составляют инженерную часть в нашей профессии. Именно эти работы позволяют SRE управлять сервисами более эффективно, чем это делали бы команды, состоящие исключительно из разработчиков или из операционистов, при более медленном (сублинейном) увеличении количества сотрудников относительно роста сервиса.

К тому же, когда мы нанимаем новых SR-инженеров, мы говорим им, что SRE — это не просто отдел операционистов и эксплуатационщиков, и цитируем это «правило 50 %». Мы должны придерживаться этого правила, не позволяя отделу SRE или любой из его команд стать чистыми операционистами.

Определяем количество рутины

Если мы хотим ограничить время, которое отдел SRE тратит на рутину, значением 50 %, то как это должно быть организовано?

Существует нижний порог доли рутинной работы, которую должен выполнить SR-инженер, находясь на дежурстве. Как правило, SR-инженер одну неделю цикла проводит в качестве

дежурного первой «очереди» и одну неделю — второй «очереди» (более подробно организация дежурств рассматривается в главе 11). Это означает, что если в ротации участвуют шесть человек и если как минимум две недели из шести посвящены дежурству и обслуживанию поступающих запросов (прерываний), то нижняя граница потенциальной рутинной работы равна $2 / 6 = 33\%$ всего рабочего времени SR-инженера. При ротации среди восьми человек нижняя граница составит $2 / 8 = 25\%$.

Таким образом, главным источником рутины являются поступающие запросы (например, несрочные сообщения и электронные письма, связанные с работой сервиса). Следующий по значимости источник — это срочные вызовы, относящиеся к выпуску и установке новых версий. И хотя наши процессы сборки и установки продуктов в значительной мере автоматизированы, в них все еще остается место для усовершенствований.

Ежеквартальные опросы SR-инженеров показывают, что в среднем на рутину тратится 33 % рабочего времени, что гораздо ниже целевого уровня 50 %. Однако такое усреднение не показывает крайних значений: некоторые SR-инженеры сообщают об отсутствии (0 %) рутинной работы (проекты, в которых они заняты исключительно разработкой, без работы на дежурствах), а другие — о 80 % рутины. Когда отдельные SR-инженеры сообщают о большом количестве такой работы, это зачастую говорит о том, что менеджерам следует более равномерно распределить рутинные

обязанности между членами команды и способствовать поиску этими SR-инженерами подходящих для них инженерных проектов.

Что такое инженерная работа

Инженерная работа по своей природе требует участия человека. Такая работа подчинена определенной стратегии и позволяет непрерывно улучшать ваш сервис. Она зачастую является творческой и инновационной: инженеры подходят к решению задач через проектирование — чем более универсальным и общим будет решение, тем лучше. С помощью инженерной работы ваша команда или отдел SRE могут справиться с масштабными сервисами (или большим количеством сервисов), не увеличивая количество обслуживающего персонала.

Обычно отдел SRE выполняет задачи, которые можно разделить на следующие категории.

- *Разработка ПО.* Включает в себя написание или модификацию кода, в дополнение к задачам, связанным с проектированием и документацией. В качестве примера можно рассмотреть написание автоматических сценариев, создание инструментов или фреймворков, добавление функциональности для масштабирования и надежности, повышение надежности инфраструктурного кода.
- *Разработка систем.* Включает в себя конфигурирование производственных систем, внесение изменений в конфигурацию или документирование систем таким образом, чтобы их можно было усовершенствовать с

минимальными усилиями. В качестве примера можно рассмотреть настройку систем мониторинга или обновлений, конфигурирование балансировщика нагрузки, конфигурирование сервера, настройку параметров ОС. Разработка систем также подразумевает консультирование команды разработчиков по вопросам архитектуры, проектирования и передачи в промышленную эксплуатацию.

- *Рутинна*. Работа, которая связана непосредственно с обслуживанием сервиса и которая является однообразной, ручной и т.д.
- *Дополнительная служебная нагрузка*. Административная работа, не связанная непосредственно с обслуживанием работающего сервиса. В качестве примера можно привести подбор и перемещения персонала, бумажную работу, совещания и митинги внутри команды или компании, наведение порядка в коде и «зачистку» накопившихся ошибок, написание сниппетов, взаимо- и самопроверку кода, а также прохождение обучающих курсов.

Каждый SR-инженер должен тратить на инженерную работу как минимум 50 % своего времени (в среднем за несколько кварталов или за год). Объем рутинной работы может резко изменяться, поэтому целевое значение 50 % для отдельных команд может оказаться невыполнимым, иногда падая ниже заданного уровня. Но если доля времени на проекты долго остается значительно ниже 50 %, команда должна приостановить работу и подумать, что пошло не так.

Всегда ли рутинна вредна

Однообразная работа не делает всех вокруг несчастными, особенно если ее немного. Выполнение запланированных и повторяющихся действий может, наоборот, оказаться успокаивающим. Они не несут рисков и не ведут к стрессам. Некоторым людям даже нравится выполнять такую работу.

Каждый инженер (в том числе SR-инженер) должен четко понимать, что полностью избежать рутинной работы невозможно. В малых объемах она допустима, и если вас устраивает периодическое выполнение повторяющихся действий, то рутине не становится проблемой. Но она начинает отравлять существование, если ее становится слишком много. Если вы перегружены ею, стоит всерьез озабочиться этим фактом и начать громко жаловаться. Рассмотрим некоторые причины, объясняющие, почему избыток рутины — это плохо.

- *Застой в карьере.* Ваше продвижение по карьерной лестнице замедлится или полностью остановится, если вы будете недостаточно времени уделять реализации проектов. В компании Google поощряется выполнение грязной работы, если этого нельзя избежать, — такая работа принесет большую пользу, но свою карьеру вы на ней построить не сможете.
- *Снижение боевого духа.* У разных людей разное представление о том, сколько рутинной работы они могут выполнить, и у каждого есть свой предел. Слишком большой объем такой работы приводит к «выгоранию», скуке и недовольству.

Кроме того, если приходится слишком много времени тратить на рутину в ущерб выполнению инженерных задач, это вредит и всему отделу SRE.

- *Начинается путаница.* Мы стараемся обеспечивать всем сотрудникам SRE работу в инженерной организации. Тот факт, что некоторые люди или команды внутри отдела выполняют слишком большой объем рутинной работы, подрывает основу этой идеи и мешает другим понять нашу роль.
- *Замедляется прогресс.* Большое количество рутины снижает продуктивность команды. Скорость выпуска новой функциональности снизится, если команда SRE-инженеров будет слишком занята ручной работой и исправлением ошибок, вместо того чтобы вовремя выпустить новый релиз.
- *Создается нежелательный прецедент.* Если вы не против заниматься рутинной работой, ваши разработчики захотят нагрузить вас еще большим ее количеством, иногда даже перепоручая вам те операционные задачи, которые они должны делать сами. Другие команды также могут начать считать, что отдел SRE должен выполнять такую работу, и это по очевидным причинам нехорошо.
- *Потери личного состава.* Даже если лично вас не огорчает необходимость выполнять однообразную работу, вашим нынешним или будущим коллегам такая перспектива может нравиться куда меньше. Если ваша команда будет перегружена рутиной, это подтолкнет лучших инженеров начать искать более интересную работу в другом месте.
- *Подрывается вера.* Новые сотрудники, которые присоединяются к отделу SRE для работы над интересными проектами, будут чувствовать себя обманутыми, что негативно сказывается на боевом духе.

Итоги главы

Если мы все будем стараться понемногу избавляться от рутинной работы, внедряя новые инженерные решения, то постепенно очистим свои сервисы и сможем перенаправить совместные усилия на масштабирование сервисов, создание сервисов нового поколения и сборку специализированных пакетов инструментов для отдела SRE. Давайте же изобретать больше и заниматься рутиной меньше!

[28](#) «Нормальные изменения» — изменения в состоянии системы, для которых предусмотрены соответствующие шаги процессов управления. — *Примеч. пер.*

[29](#) Мы используем систему «Цели и ключевые результаты», ее разработал Энди Гроув из компании Intel [Klau, 2012].

[30](#) Сниппеты — небольшие отчеты в свободной форме, в которых мы рассказываем о том, над чем работали каждую неделю.

[31](#) Не стоит торопиться говорить, что работа «не является рутиной, поскольку в процессе выполнения задачи человек должен принять какое-то решение». Нужно внимательно проанализировать природу задачи, чтобы понять, действительно ли она нуждается в участии человека, или же можно спроектировать ее решение иначе. Например, кто-то может создавать (а кто-то даже уже создал) сервис, который несколько раз в день оповещает обслуживающих его SR-инженеров о возникающих проблемах, и человек затрачивает много времени на разрешение каждой такой проблемы. Очевидно, что подобный сервис плохо спроектирован и чрезмерно усложнен. Систему нужно упростить и перестроить так, чтобы либо вовсе избавиться от состояний, приводящих к ошибке, либо автоматизировать решение возникающих проблем. До завершения перепроектирования и ввода в эксплуатацию новой версии сервиса описанная работа, предполагающая привлечение человека к принятию решений, будет рутиной.

6. Мониторинг распределенных систем

Автор — Роб Иващук

Под редакцией Бетси Байер

SR-инженеры компании Google выработали базовые принципы и приемы построения эффективных систем мониторинга и оповещения. В этой главе вы прочитаете, о каких сбоях системе мониторинга следует оповещать человека и как поступать с несложными проблемами, не требующими оповещений.

Определения

В области мониторинга не существует единого словаря терминов — даже в компании Google используемая терминология может различаться. Самые распространенные определения перечислены ниже.

- *Мониторинг (наблюдение)*. Сбор, обработка, агрегирование и отображение в реальном времени количественных показателей системы, например общее число и тип запросов, количество ошибок и их типы, время обработки запросов и время функционирования серверов.
- *Наблюдение методом белого ящика*. Наблюдение с использованием показателей, доступных внутри системы, включая журналы, интерфейсы вроде Java Virtual Machine Profiling Interface или обработчики HTTP, которые предоставляют внутреннюю статистику.
- *Наблюдение методом черного ящика*. Наблюдение и проверка (тестирование) поведения, видимого извне, с точки зрения

пользователя.

- *Информационная панель (панель управления)*. Приложение (как правило, веб-приложение), которое предоставляет сводную информацию об основных показателях сервиса. Информационная панель может иметь фильтры, селекторы и т.д., но заранее ее конфигурируют так, чтобы показывать наиболее важные для пользователей сведения. Информационная панель также может отображать сведения, важные для команды, например длину очереди тикетов, списка наиболее приоритетных ошибок, имени дежурного инженера или идентификаторы последних установок ПО.
- *Оповещение («алерт»)*. Сообщения, на которые должен обратить внимание человек и которые направляются в конкретную систему, например в очередь запросов («тиcketов»), в электронную почту или на специальное устройство — пейджер. Соответственно, оповещения делятся на «тиcketы» (оповещения по электронной почте)³² и вызовы (экстренные оповещения)³³.
- *Основная причина*. Дефект в ПО или ошибка, допускаемая человеком, после исправления которых можно быть уверенными, что аналогичное событие не произойдет. Некоторые события могут иметь несколько основных причин: например, событие было вызвано недостаточной автоматизацией процесса, программным сбоем из-за некорректных входных данных и недостаточно качественным тестированием сценария, использованного для создания конфигурации. Каждый из этих факторов может быть основной причиной и каждый из них должен быть исправлен.

- Узел или машина. Эти взаимозаменяемые термины используются для обозначения единственного экземпляра запущенного ядра на физическом сервере, виртуальной машине или в контейнере. На одной машине могут работать несколько сервисов, за которыми нужно следить. Эти сервисы могут быть:
 - связаны друг с другом: например, сервер кэширования и веб-сервер;
 - не связаны друг с другом, но могут совместно использовать одно и то же оборудование: например, репозиторий кода и мастер конфигурационной системы вроде Puppet (<https://puppetlabs.com/puppet/puppet-open-source>) или Chef (<https://www.chef.io/chef/>).
- Установка версии (push). Любые изменения, производимые в ПО работающего сервиса или его конфигурации.

Зачем нужен мониторинг

Существует множество задач, решаемых с помощью мониторинга системы, включая следующие.

- Анализ долгосрочных тенденций. Насколько велика база данных и как быстро она растет? Как быстро увеличивается количество пользователей с ежедневной активностью?
- Сравнение с предыдущими версиями или экспериментальными группами. Выполняются ли запросы быстрее с помощью Acme Bucket of Bytes 2.72 по сравнению с Ajax DB 3.14? Насколько улучшится коэффициент попаданий поисковых запросов в кэш, если добавить дополнительный узел?

Работает ли сайт сегодня медленнее, чем на прошлой неделе?

- *Оповещение.* Что-то сломалось, и кто-то должен исправить это прямо сейчас. Или же что-то может скоро сломаться, и кто-то должен будет этим заняться.
- *Создание информационных панелей.* Информационные панели должны содержать ответ на главные вопросы о вашем сервисе, и это обычно так называемые четыре золотых сигнала в том или ином виде (эта тема рассматривается далее, в разделе «Четыре золотых сигнала»).
- *Ретроспективный анализ различного назначения* (например, отладка). Время отклика в нашей системе только что возросло; что еще случилось в это же время?

Наблюдение за системой также позволяет получить первичные данные для бизнес-аналитики и анализа брешей в защите.

Мониторинг и оповещение позволяют системе вовремя сообщить о возникшей или ожидающейся неисправности. Если система не может справиться с ней самостоятельно, то необходимо, чтобы человек проанализировал полученное оповещение, определил серьезность проблемы, нашел временное решение и затем выяснил основную причину проблемы. Однако недопустимо отвлекать инженеров сообщениями о том, что «что-то немного странно выглядит», если только речь не идет о безопасности в некоторых узкоспециализированных компонентах системы.

Вызов инженера — это очень дорогой вариант использования времени сотрудника. Если сотрудник находится в офисе, вызов прервет его текущую работу. Если сотрудник

находится дома, вызов отвлечет его от личных дел или, возможно, даже прервет сон. Если вызовы происходят слишком часто, сотрудник начинает действовать по шаблону, уделять им недостаточно внимания или даже игнорировать их, иногда пропуская «настоящие». Реальные простые системы могут затягиваться из-за того, что подобные «шумы» мешают быстро ее проанализировать и восстановить. Эффективные системы оповещения должны иметь очень хорошее соотношение «сигнал/шум».

Ставим для мониторинга реальные задачи

Наблюдение за крупным приложением — сложная инженерная задача. Даже имея подходящую инфраструктуру для измерений, сбора данных, их отображения и оповещения о проблемах, команда SRE, состоящая из 10–12 человек, обычно выделяет одного сотрудника (иногда двух), чьей основной задачей становится создание и обслуживание системы мониторинга для их сервиса. Количество привлекаемых сотрудников снижалось по мере того, как мы обобщали и централизовали инфраструктуру для мониторинга, но в каждой команде SR-инженеров все равно имеется хотя бы один «наблюдатель». (При этом, впрочем, хотя наблюдать информационные панели с объемами трафика и прочей информацией бывает интересно, команды SR-инженеров старательно избегают ситуаций, когда кому-то приходится «смотреть на экран и искать проблемы».)

В компании Google предпочитают создавать простые и быстрые системы мониторинга с качественными инструментами для анализа состояния «*по факту*». Мы избегаем «магических» систем, которые пытаются самостоятельно исследовать пороговые значения или

причинно-следственные связи. Исключением являются правила, которые обнаруживают неожиданные изменения на уровне запросов конечных пользователей. Оставаясь несложными (насколько это возможно), эти правила позволяют быстро обнаруживать аномалии — наиболее простые, наиболее серьезные или отдельные специфические их виды. При других вариантах использования данных мониторинга, например при планировании производительности и прогнозировании трафика, можно допустить меньшие точность и оперативность и, соответственно, действовать более сложные алгоритмы. Длительное (проводимое на протяжении месяцев или лет) наблюдение с низкой частотой выборки (часы или дни) также обычно устойчиво к погрешностям, поскольку случайный пропуск выборок не может исказить долгосрочные закономерности.

Отдел SRE компании Google лишь ограниченно работает со сложными иерархиями зависимостей. Мы редко используем правила вроде: «Если я знаю, что база данных работает медленно, то сообщить о медленной базе данных; иначе сообщить о том, что сайт работает медленно сам по себе». Правила, построенные на зависимостях, обычно относятся к очень стабильным частям нашей системы. Пример этого — система для отвода пользовательского трафика из data-центра. Так, общим для всех data-центров является правило «Если data-центр пуст, то не сообщать мне о его задержке отклика». Однако некоторые команды SRE все же имеют дело со сложными иерархиями зависимостей, поскольку наша инфраструктура постоянно реорганизуется.

Некоторые идеи, описанные в этой главе, могут вдохновить вас: всегда можно ускорить «постановку диагноза» — перейти от симптома к основной проблеме, особенно в постоянно изменяющихся системах. Здесь мы описываем цели, которые

ставим перед системами мониторинга, а также способы их достижения. При этом всегда важно следить за тем, чтобы системы мониторинга — особенно критически важный этап от появления проблемы и вызова инженера до ее рассмотрения и глубокой отладки — были простыми и прозрачными для всех членов команды.

Аналогично, чтобы улучшить соотношение «сигнал/шум», элементы вашей системы мониторинга, отвечающие за вызов инженера, должны быть очень простыми и надежными. Правила, которые генерируют предназначенные людям оповещения, должны быть простыми и понятными и всегда указывать только на несомненные сбои.

Симптомы и причины

Ваша система мониторинга должна отвечать на два вопроса: что сломалось и почему.

Ответ на вопрос «Что сломалось?» заключает в себе указание на симптом; ответ на вопрос «Почему?» — указание на причину (возможно, промежуточную). В табл. 6.1 перечислены некоторые гипотетические симптомы и соответствующие им причины.

Таблица 6.1. Примеры симптомов и причины, по которым они появляются [34](#)

Симптом	Причина
Сервис возвращает ответы HTTP 500 или 404	Серверы базы данных отказывают в соединении
Ответы генерируются медленно	Процессоры перегружены из-за выполнения сортировки алгоритмом bogosort ¹ или заломлен кабель Ethernet под стойкой, что приводит к частичной потере пакетов
Пользователи в Антарктике	Ваша сеть доставки контента ненавидит ученых и

не получают анимированные GIF-изображения с котиками	представителей семейства кошачьих, поэтому поместила IP-адреса некоторых клиентов в черный список
Личные данные видны всему миру	При установке новой версии ПО были утеряны списки управления доступом (ACL), в результате чего всем пользователям предоставлен полный доступ

Причинно-следственная связь «что» и «почему» — это то, что должна выявлять хорошо сделанная система мониторинга с максимальным соотношением «сигнал/шум».

Методы черного и белого ящика

Для мониторинга мы широко используем метод белого ящика; метод черного ящика применяется редко, но не менее важен. Самый простой способ понять разницу между этими методами — представить метод черного ящика как наблюдение за симптомами, которое выявляет реально возникшие — а не спрогнозированные — проблемы: «В данный момент система работает некорректно». Метод белого ящика зависит от возможности исследовать систему изнутри (например, журналы или интерфейсы HTTP) с помощью каких-либо инструментов. Поэтому метод белого ящика позволяет обнаруживать будущие проблемы, замаскированные повторными попытками сбои и пр.

Обратите внимание: в многоуровневой системе наблюдаемый одними симптом для других становится причиной. Предположим, что производительность базы данных упала. Медленное чтение информации из базы будет симптомом для инженера, который следит за базами данных. Однако для инженера, который следит за фронтендом и замечает, что сайт притормаживает, медленное чтение информации из базы данных становится причиной. Поэтому метод белого ящика иногда является симптомоориентированным, а иногда — причинно-

ориентированным, в зависимости от того, насколько информативен ваш белый ящик.

При сборе данных телеметрии для отладки метод белого ящика крайне необходим. Если веб-серверы с «тяжелыми» запросами к базе данных работают медленно, необходимо выяснить, насколько быстро получает данные из базы сервер и насколько быстро работает база сама по себе. Иначе невозможно будет отделить действительно медленную базу данных от проблемы с соединением между базой и веб-сервером.

Как источник экстренных оповещений метод черного ящика имеет неоспоримое преимущество: такая система обращается к человеку только в том случае, если проблема уже существует и дает реальные симптомы. Однако для еще не произошедших, но неизбежных в будущем проблем метод черного ящика практически бесполезен.

Четыре золотых сигнала

Четырьмя золотыми сигналами (точнее, показателями. — Примеч. пер.) для мониторинга являются время отклика, величина трафика, уровень ошибок и степень загруженности. Если вы можете измерить для своей системы только четыре показателя, сконцентрируйтесь именно на них.

- *Время отклика.* Время, которое требуется для выполнения запроса. Очень важно различать задержки для успешных и неуспешных запросов. Например, код ошибки HTTP 500, вызванной потерей соединения с базой данных или каким-то критическим сбоем на стороне бэкенда, может быть возвращен очень быстро. Однако, поскольку ошибка HTTP 500 указывает на то, что запрос не был выполнен, учитывать

такие результаты при подсчете общей задержки нельзя. С другой стороны, «медленная» (возвращенная с большой задержкой) ошибка даже хуже «быстрой»! Поэтому важно не просто фильтровать ошибки, но и анализировать задержку выполнения соответствующих запросов.

- *Величина трафика.* Величина нагрузки, которая приходится на вашу систему и измеряется в высокоуровневых единицах, зависящих от конкретной системы. Для веб-сервисов трафик измеряется как количество запросов HTTP в секунду, обычно дополняемое информацией о характере запросов (например, статический или динамический контент). Для системы потокового аудио это может быть скорость передачи данных по сети или количество параллельных сессий. Для систем хранения, построенных по схеме «ключ — значение», таким показателем может служить количество выполненных транзакций и возвращенных значений за секунду.
- *Уровень ошибок.* Количество (или частота) неуспешного выполнения запросов: явно (например, если возвращен код HTTP 500), неявно (если возвращен код HTTP 200, но результат получен неправильный) или не соответствующих требованиям (например, если вы решили, что ответ должен приходить не более чем через секунду, то после ожидания ответа в течение этого времени любой запрос будет считаться неуспешным). Поскольку коды ответов не могут отразить весь спектр ошибочных состояний, для определения конкретных неполадок могут понадобиться вторичные (внутренние) протоколы. В подобных случаях подходы к мониторингу могут быть очень разными. Например, фильтрация ответов с кодом HTTP 500 на уровне балансировщика нагрузки позволяет обнаружить

однозначно ошибочные запросы, но лишь «сквозные» тесты всей системы могут выяснить, что вы пытаетесь обрабатывать некорректные данные.

- *Степень загруженности.* Показатель того, насколько полно загружен ваш сервис. Эти измерения показывают те компоненты системы и те ресурсы, в которых вы ограничены больше всего (например, в системах, ограниченных в памяти, это память; в системах, ограниченных в возможностях ввода/вывода, — состояние подсистемы ввода/вывода). Обратите внимание, что многие системы теряют в производительности еще до того, как будут загружены на 100 %, поэтому следить за уровнем загрузки как за целевым показателем также очень важно.

Для сложных систем это может быть дополнено более высокоуровневыми оценками нагрузки: справится ли ваш сервис с удвоением трафика? Обработает ли он всего 10 % дополнительного трафика или даже снизит производительность? Если сервис простой и не имеет параметров, изменяющих сложность запроса (например, «Выдай случайное число» или «Мне нужна уникальная монотонная последовательность целых чисел»), то статичное значение, полученное в результате нагрузочного тестирования, может быть корректным. Однако, как мы говорили в предыдущем абзаце, для большинства сервисов приходится использовать косвенные показатели с известным верхним пределом, например процент загрузки процессора или пропускной способности сети. Часто первым индикатором «насыщения» системы становится увеличение времени отклика. Измерение 99-го перцентиля времени отклика на небольших временных промежутках (например,

за одну минуту) может послужить ранним предупреждением о намечающейся перегрузке.

Наконец, важно обращать внимание на прогнозы о достижении целевого уровня, например: «Похоже, что ваша база данных заполнит жесткий диск через четыре часа».

Если вы можете измерить все четыре золотых сигнала и сообщить пользователю о том, что один из них обнаруживает проблему (или, в случае целевого уровня использования, проблема скоро наступит), качество мониторинга для вашего сервиса будет как минимум удовлетворительным.

Позаботимся о своем «хвосте» (или производительность измеряемая и наблюдаемая)

При проектировании системы мониторинга с нуля очень заманчивой может показаться идея использовать средние значения: среднюю задержку, средний процент загрузки процессора ваших узлов или среднюю заполненность ваших баз данных. Рискованность такой оценки в двух последних случаях очевидна: процессор и база данных могут использоваться очень неравномерно. То же верно и для задержек. Если вы запускаете веб-сервис, имеющий среднюю задержку отклика 100 миллисекунд при 1000 запросах в секунду, то обработка 1 % запросов может занять и 5 секунд³⁵. Если для формирования страницы нужно выполнить несколько подобных запросов, 99-й процентиль задержки отклика одного из серверов легко может стать средней задержкой для клиента.

Самый простой способ различить медленное среднее время обработки и очень медленные «хвостовые» запросы — вместо самих значений задержки использовать количество запросов, величина задержки для которых попадает в заданные интервалы, удобные для построения гистограммы: сколько

запросов потребовали для обработки от 0 до 10 миллисекунд, от 10 до 30, от 30 до 100, от 100 до 300 и т.д. Построение гистограммы с примерно логарифмически расставленными границами интервалов (в нашем случае с основанием, приблизительно равным 3) — зачастую самый простой способ наглядно представить распределение характеристик ваших запросов.

Выбор подходящего уровня детализации для измерений

Для разных компонентов системы измерения должны проводиться с разным уровнем детализации.

- Контроль загрузки центрального процессора с периодичностью в минуту не покажет даже сравнительно продолжительные пики, которые вызывают наибольшие задержки («хвост» распределения времени задержки).
- С другой стороны, если для веб-сервиса заявлено, что в течение года он будет отключен в общей сложности не более 9 часов (99,9 % времени доступности в год), то, по всей видимости, ежеминутная отправка 1–2 проверочных запросов лишь для того, чтобы получить код 200 (успех), будет излишней.
- Аналогично проверять заполненность жестких дисков каждые 1–2 минуты для системы с целевым значением доступности 99,9 %, скорее всего, не требуется.

Вам необходимо тщательно настраивать детализацию своих измерений. Ежесекундные измерения загрузки процессора могут рассказать много интересного, но их сбор, хранение и анализ обойдутся очень дорого. Если для вашего мониторинга

нужна высокая степень детализации, но не требуется особенная оперативность получения данных, вы можете сэкономить, накапливая первичную информацию на сервере, а затем сконфигурировав внешнюю систему так, чтобы она собирала воедино данные за определенное время или с нескольких серверов. Например, можно сделать следующее.

1. Записывать текущий показатель загрузки процессора каждую секунду.
2. Создать заготовку для гистограммы с шагом 5 % и каждую секунду увеличивать значение для соответствующего столбца в зависимости от текущего показателя использования CPU.
3. Раз в минуту собирать итоговые данные.

Такая стратегия позволит вам наблюдать короткие по времени пики нагрузки процессора без излишних затрат на сбор и хранение данных.

Максимально просто, но не проще

Стремление выполнить все эти и многие другие требования сильно усложняет систему мониторинга. Можно выделить следующие уровни сложности:

- оповещение о достижении разных порогов задержки отклика и разных процентиелей для всех видов показателей;
- дополнительный код для обнаружения и отображения возможных причин;

- соответствующие информационные панели для каждой из возможных причин.

Причины потенциального увеличения сложности никогда не иссякают. Как и другие программные системы, система мониторинга может стать настолько сложной, что окажется нестабильной, и ее будет трудно модифицировать и тяжело поддерживать.

Поэтому вам нужно проектировать систему мониторинга так, чтобы она оставалась максимально простой. При выборе контролируемых величин имейте в виду следующее.

- Правила, которые определяют реальные аварийные ситуации, должны быть максимально простыми, предсказуемыми и надежными.
- Следует избавиться от частей системы, отвечающих за сбор данных, их агрегирование и оповещение, которые используются слишком редко (например, реже чем раз в квартал и лишь отдельными командами SR-инженеров).
- Показатели, которые система собирает, но не выводит на готовых информационных панелях и не использует для оповещений, также являются кандидатами на удаление.

По опыту Google, система мониторинга, которая просто собирает и агрегирует показатели, а также занимается оповещениями и созданием информационных панелей, работает как относительно самостоятельная система. (Фактически система мониторинга Google разбита на несколько отдельных программ, но обычно люди изучают их все.) Заманчиво было бы объединить систему мониторинга с

другими контрольными функциями сложных систем вроде детализированного профилирования системы, однопроцессной отладки, отслеживания исключений и сбоев, тестирования нагрузки, сбора и анализа журнала или исследования трафика. Хотя все эти цели имеют много общего с лежащим в их основе мониторингом, объединение слишком большого количества результатов приведет к тому, что система станет слишком сложной и нестабильной. Как и во многих других случаях при разработке ПО, использование отдельных систем, имеющих четкие, простые и слабо связанные точки взаимодействия, — это наиболее удачная стратегия. Например, удобно использовать API веб-запросов для отправки результирующих данных в формате, который может долго оставаться неизменным.

Сводим все принципы воедино

Принципы, рассмотренные в этой главе, можно увязать друг с другом, получив своего рода философию мониторинга и оповещения, которой следуют команды Google SRE. Помимо прочего, эти положения являются хорошей отправной точкой для создания нового оповещения или пересмотра существующего. Это же поможет вам задавать правильные вопросы независимо от размера вашей организации или сложности вашего сервиса или системы.

При создании правил для мониторинга и оповещения, чтобы избежать как реальных ошибок, оставшихся незамеченными, так и огромного количества экстренных оповещений, вам нужно задать себе следующие вопросы³⁶.

- Позволяет ли правило выявить *не обнаруживаемое иными средствами состояние*, которое требует быстрой реакции и

последствия которого уже заметны (или обязательно станут заметными) пользователю?³⁷

- Смогу ли я проигнорировать это оповещение, зная, что оно не критическое? Когда и почему я смогу проигнорировать оповещение и как можно избежать этого варианта развития событий?
- Говорит ли это оповещение однозначно о том, что на пользователей уже оказывается какое-то негативное влияние? Можно ли распознать ситуации, при которых пользователи не страдают (например, при отводе трафика или развертывании тестов) и которые должны быть отфильтрованы?
- Могу ли я что-то предпринять в ответ на это оповещение? Нужно ли отреагировать немедленно, или же это может подождать до утра? Можно ли безопасно автоматизировать это действие? Будет ли это действие постоянным решением или временным?
- Приходит ли это оповещение другим инженерам? Это означало бы, что как минимум одно из них бесполезно.

Следующие вопросы отражают нашу позицию по экстренным оповещениям, передаваемым на пейджеры.

- Каждый раз, когда сигналит пейджер, я должен отреагировать немедленно, но осмысленно. Без переутомления я могу так реагировать всего несколько раз за день.

- Каждому оповещению должны соответствовать определенные ответные действия.
- Каждый ответ на экстренное оповещение должен быть продуманным. Если оповещение заслуживает лишь автоматического ответа, оно не должно быть экстремальным.
- Экстренное оповещение и вмешательство человека необходимы при появлении новой проблемы или ранее не встречавшегося события.

На основе сказанного можно сделать вывод о несущественности некоторых обстоятельств. Например, если оповещение удовлетворяет четырем перечисленным выше признакам, то уже неважно, каким образом оно было сгенерировано: методом черного ящика или методом белого ящика. С другой стороны, иные обстоятельства, наоборот, акцентируются: лучше потратить больше времени и сил на поиск симптомов, чем на поиск причин. Если и стоит искать причины, то только самые явные и неизбежные.

Долгосрочное наблюдение

В наше время для промышленных систем мониторинг обеспечивает слежение за постоянно развивающимися системами, у которых меняются архитектура, характеристики нагрузки и целевой уровень производительности. Если на данный момент какое-то оповещение появляется редко и ответ для него автоматизировать сложно, то в дальнейшем оно может начать появляться чаще и, возможно, даже потребует создания обходного сценария для разрешения проблемы. Тогда же придется заняться поиском и устранением основной

причины проблемы. Если же это окажется невозможным, реакция на такое оповещение должна быть полностью автоматизирована.

Важно, чтобы решения, касающиеся мониторинга, принимались с учетом долгосрочных перспектив. Каждое экстренное оповещение сегодня отвлекает человека от улучшения системы завтра, поэтому зачастую нам приходится отказываться от высоких показателей доступности или производительности сейчас, чтобы улучшить эти же показатели в будущем. Рассмотрим две ситуации, где явно виден такой компромисс.

Bigtable SRE: когда оповещений слишком много

Для оценки характеристик внутренней инфраструктуры компании Google обычно используется целевой уровень качества обслуживания (service level objective, SLO; см. главу 4). Много лет назад SLO в отношении сервиса Bigtable основывались на синтетических средних показателях производительности для «хороших» клиентов. Из-за проблем в самом этом сервисе и в нижних уровнях стека хранения данных среднее значение производительности портил большой «хвост»: худшие 5 % запросов обрабатывались значительно медленнее остальных.

При приближении к заданным в SLO пределам рассыпались оповещения по электронной почте, а при их превышении отправлялись экстренные оповещения на пейджер. Оповещения обоих типов рассыпались в большом количестве, отнимая значительную часть времени инженеров. Команда тратила много времени на проверку всех оповещений, чтобы найти те несколько, которые действительно требовали срочного принятия мер. Мы зачастую пропускали проблемы, непосредственно мешавшие пользователям, поскольку именно

такие оповещения были относительно редки. Многие оповещения были лишними, поскольку появлялись из-за давно известных проблем в архитектуре, и ответ на них был автоматический либо не предполагался вовсе.

Исправление этой ситуации потребовало трехэтапного решения. Помимо того, что мы прилагали значительные усилия для улучшения производительности сервиса Bigtable, мы также временно снизили целевое значение SLO, воспользовавшись значением 75-го процентиля задержки отклика. Мы также отключили оповещения по электронной почте, поскольку их было так много, что все невозможно было рассмотреть.

Такая стратегия дала нам достаточно времени для исправления проблем сервиса Bigtable и нижних уровней стека хранения, вместо того чтобы непрерывно заниматься тактическими проблемами. Дежурные инженеры смогли выполнять свою работу, поскольку больше не отвлекались на постоянные вызовы. В конечном счете отключение избыточных оповещений позволило нам быстрее улучшить наш сервис.

Gmail: предсказуемые ответы человека

Изначально сервис Gmail был построен на базе модернизированной системы управления распределенными процессами Workqueue, которая создавалась для пакетной обработки фрагментов поискового индекса. Workqueue была адаптирована для работы с «долгоиграющими» процессами, и в итоге ее применили к сервису Gmail, но от некоторых ошибок в относительно непрозрачной базе кода планировщика было очень трудно избавиться.

В то время мониторинг сервиса Gmail был построен таким образом, что оповещениями сопровождался «вывод из расписания» отдельных задач сервиса Workqueue. Такой подход был далек от идеала, поскольку даже тогда Gmail уже имел многие тысячи задач, каждая из которых обслуживала лишь долю процента наших пользователей. Мы заботились о том, чтобы пользователям было комфортно работать с Gmail, но поддерживать такое количество оповещений было нереально.

Чтобы справиться с этой проблемой, SR-инженеры, отвечавшие за сервис Gmail, создали инструмент, который бы помогал «подталкивать» планировщик, минимизируя негативное влияние на пользователей. Члены команды долго спорили, стоит ли автоматизировать весь цикл (начиная от обнаружения проблемы и заканчивая «подталкиванием» перепланировщика) на то время, пока не будет создано более качественное долгосрочное решение. Некоторые из них беспокоились, что подобный обходной вариант приведет к откладыванию реального решения проблемы.

В команде часто возникают такие споры: в то время как одни инженеры хотят создать «костыль», который даст время для поиска хорошего решения, другие волнуются, не забудут ли со временем о наличии этого «костыля» или не будет ли навсегда отложен поиск хорошего решения. Такую озабоченность можно понять: очень легко увеличивать «технический долг», выпуская патчи к проблемам, вместо того чтобы действительно искать грамотные решения. При этом ключевую роль при реализации долгосрочных решений играют менеджеры и руководители из инженерного состава. Их приоритет — разработка и поддержка таких решений, даже если на их поиск и реализацию уйдет много времени.

Оповещения, на которые приходят повторяющиеся заученные ответы, должны послужить для вас красным флагом.

Нежелание членов вашей команды автоматизировать обработку таких ситуаций может указывать на то, что они не уверены в своей способности вернуть технический долг. Это серьезная проблема, которая может усугубляться.

Долгосрочная перспектива

У предыдущих примеров, в которых рассматривались сервисы Bigtable и Gmail, есть нечто общее: конфликт между доступностью в краткосрочной и долгосрочной перспективе. Зачастую нестабильную систему все-таки удается довести до требуемых высоких показателей доступности, но это связано с приложением серьезных усилий, приводит к перегрузке и «выгоранию», успех зависит от немногих наиболее героических членов команды, а эффект оказывается не слишком длительным. Контролируемое краткосрочное снижение показателей доступности зачастую очень болезненно, но это стратегическая жертва на благо стабильности системы на длинной дистанции. Очень важно не рассматривать каждое оповещение как отдельное событие. Вместо этого нужно взглянуть на общий уровень вызовов и определить, соответствует ли он работоспособной и достаточно доступной системе, имеющей стабильную команду и долгосрочную перспективу. Вместе с менеджерами мы рассматриваем статистику частоты оповещений (обычно выражаемую в количестве сбоев за смену, где сбой может сопровождаться несколькими оповещениями) в квартальных отчетах. Это гарантирует, что люди, принимающие решения, в курсе количества оповещений о сбоях и общего состояния своих команд.

Итоги главы

Необходимость грамотной системы мониторинга и оповещения легко обосновать. Такая система в первую очередь обеспечивает оповещение инженера при появлении симптомов, оставляя эвристические выводы о причинах для этапа отладки. Наблюдать за симптомами тем проще, чем выше в иерархии находится система. Наблюдение за нагрузкой и производительностью подсистем вроде баз данных зачастую должно производиться непосредственно на самой подсистеме. Оповещения, отправляемые по электронной почте, не всегда эффективны и зачастую скрываются за информационным «шумом». Вместо этого для отслеживания всех существующих докритических проблем лучше пользоваться информационной панелью — там будет видна информация, которая обычно содержится в оповещениях, отправляемых по электронной почте. Информационные панели могут быть объединены с журналом для того, чтобы можно было проанализировать изменение показателей со временем.

В долгосрочной перспективе эффективная организация посменных дежурств и создание успешного продукта потребуют наличия оповещения о симптомах и приближающихся реальных проблемах. Для этого нужно установить такие целевые значения показателей, которые реально достижимы, а также убедиться, что ваша система мониторинга поддерживает быструю диагностику.

[32](#) Иногда их называют спам-оповещениями, поскольку их редко читают и на них редко реагируют.

[33](#) Оригинальный термин — page. Забегая вперед, в качестве пейджеров используются в основном обычные мобильные устройства. — Примеч. пер.

[34](#) Bogosort, или «обезьяня сортировка», — крайне неэффективный, не используемый на практике алгоритм сортировки, состоящий в переборе всех возможных перестановок элементов. — Примеч. пер.

[35](#) Если 1 % от всех запросов обрабатывается в десять раз дольше среднего, это означает, что остальные ваши запросы обрабатываются примерно в два раза быстрее среднего. Но без учета распределения времени предположение о том, что большинство запросов выполняются за время около среднего, — лишь принятие желаемого за действительное.

[36](#) Прочтите статью Applying Cardiac Alarm Management Techniques to Your On-Call (Holmwood, 2014), чтобы увидеть пример перегрузки вызовами в другом контексте.

[37](#) Ситуации с нулевой избыточностью ($N + 0$) считаются неизбежными! Чтобы больше узнать о концепции избыточности, прочтите статью https://en.wikipedia.org/wiki/N%2B1_redundancy.

7. Эволюция автоматизации в Google

Авторы — Нейл Мёрфи, Джон Луни и Майкл Кейсиerek

Под редакцией Бетси Бейер

Кроме черной магии, есть
автоматизация и механизация.

Федерико Гарсия Лорка (1898–1936), испанский поэт и драматург

Для отдела SRE автоматизация является лишь своего рода рычагом, «усилителем», но не панацеей. Конечно, просто увеличив силу, мы никак не повлияем на точность ее приложения: бездумная автоматизация может создать больше проблем, чем решить. И хотя в большинстве случаев считается, что автоматизация (использование программ) предпочтительнее ручного труда, наиболее выгодным вариантом является создание высокоуровневой *автономной* системы, которая не нуждается ни в автоматизации, ни в ручной работе. Другими словами, польза автоматизации зависит от того, что мы делаем и как это используем. В этой главе мы рассмотрим ценность автоматизации и изменение нашего к ней отношения с течением времени.

Польза автоматизации

Чем же хороша автоматизация [38](#)?

Постоянство

Хотя масштабирование само по себе является достаточной мотивацией для автоматизации, существует и множество других причин ее использования. Для примера возьмем вычислительные системы университетов, где начинали свою

карьеру многие системные инженеры. Работавшие там системные администраторы должны были управлять группой компьютеров или поддерживать разнообразный софт, а также вручную выполнять различные действия. Один из наиболее типичных примеров — создание пользовательских учетных записей. К другим работам относятся чисто эксплуатационные задачи вроде проверки создания резервных копий, устранение сбоев при отказе сервера и выполнение манипуляций с данными, например изменение содержимого файла `resolv.conf` вышестоящих DNS-серверов, информации о DNS-зонах и т.д.

Однако в конце концов ситуация, когда множество задач решается преимущественно вручную, перестала устраивать как организации, так и самих работников. Как минимум любое действие, многократно выполняемое человеком или группой людей, каждый раз будет выполняться по-разному. Даже при наличии лучшей в мире силы воли лишь немногие из нас смогли бы быть такими же постоянными, как машины. Это неизбежное непостоянство приводит к ошибкам, упущениям, проблемам с качеством данных, а также к проблемам с надежностью. С этой точки зрения — единообразного выполнения хорошо известных процедур — основной пользой автоматизации становится постоянство действий.

Платформа

Автоматизация дает нам не только постоянство и непротиворечивость. Качественно спроектированные и реализованные автоматические системы также предоставляют нам *платформу*, которую можно расширить, применить к другим системам или, возможно, извлечь из нее какую-либо дополнительную выгоду³⁹. (Противоположный вариант —

отсутствие автоматизации — не является ни эффективным с точки зрения затрат, ни расширяемым: вместо этого он увеличивает нагрузку на систему.)

Платформа также позволяет *централизовать ошибки*. Другими словами, ошибка, исправленная в коде, будет исправлена раз и навсегда, в отличие от рассмотренной ранее ситуации, когда несколько человек выполняют одну и ту же процедуру. Платформу можно расширять, чтобы она решала дополнительные задачи, и сделать это проще, нежели обучить людей решать эти же задачи (или хотя бы объяснить им, что они должны делать). В зависимости от характера задачи система может работать непрерывно и более интенсивно, чем люди, решающие те же задачи. Кроме того, система может функционировать в любое время, даже неудобное для человека. Платформа может экспортировать показатели своей производительности или предоставить вам детальную информацию о процессе, которой вы не владели ранее, поскольку такие подробные измерения гораздо проще выполнить именно в контексте платформы.

Быстрое восстановление

Системы, где автоматизация применяется для устранения наиболее частых ошибок и сбоев (это типично для автоматизации, разработанной командами SRE), получают еще одно дополнительное преимущество. Если автоматизированная часть системы запускается часто и достаточно успешно, результатом будет снижение среднего времени восстановления (mean time to repair, MTTR) после подобных ошибок. Вы сможете потратить свое время на решение других задач, достигая тем самым большей скорости разработки, поскольку вам не придется предотвращать

проблему либо (что более распространено) наводить порядок после нее.

В среде разработчиков хорошо известно, что чем позже обнаруживается проблема, тем дороже ее исправить (см. главу 17). Как правило, исправление ошибок, возникающих в уже работающих системах, обходится дороже всего с точки зрения затрат как времени, так и финансов. Это означает, что автоматизированная система, которая оперативно обнаруживает проблемы по мере их появления, с высокой вероятностью может снизить общую стоимость системы, особенно если система достаточно велика.

Быстродействие

В условиях, располагающих к внедрению автоматизации SRE, люди не могут реагировать так же быстро, как машины. В большинстве случаев, когда, например, в каком-то конкретном приложении способ устранения ошибки или правила переключения трафика могут быть хорошо определены, нет никакого смысла требовать от человека периодически нажимать кнопку «Позволить системе продолжать свою работу». (Да, иногда автоматизированные процедуры могут сделать плачевную ситуацию еще хуже. Именно поэтому они допустимы лишь в хорошо изученных областях.) В системах компании Google применяется большое количество автоматизированных решений. Во многих случаях те сервисы, которые мы поддерживаем, не смогли бы достаточно долго существовать без автоматизации, поскольку количество необходимых для них ручных операций уже давно превысило бы разумные пределы.

Экономия времени

Наконец, экономия времени — наиболее часто упоминаемый аргумент в пользу автоматизации. И хотя его приводят чаще других, прямой подсчет выигрыша во времени зачастую затруднен. Инженеры нередко прикидывают, стоит ли писать какой-то определенный фрагмент кода или решения по автоматизации. При этом они сравнивают усилия, которые можно будет сэкономить, если не заставлять людей выполнять задачу вручную, и усилия, которые нужно приложить для написания этого фрагмента кода⁴⁰. Однако можно легко забыть о том, что, как только вы автоматизируете выполнение какой-то задачи, ее сможет выполнить любой. Поэтому экономия времени распространяется на всех, кто сможет пользоваться автоматизированным решением. Устранение зависимости выполняемых операций от конкретного исполнителя — очень веский аргумент в пользу автоматизации.



Джозеф Биронас, SR-инженер, который в свое время отвечал за дата-центры компании Google, имеет такое мнение: «Если мы разрабатываем процессы и решения, которые нельзя автоматизировать, нам придется продолжить нанимать людей для обслуживания системы. Если нам придется продолжать нанимать людей для выполнения этой работы, выйдет так, что мы будем кормить машины человеческими кровью, потом и слезами. Представьте себе “Матрицу” без спецэффектов, но с множеством измученных и обозленных системных администраторов».

Польза автоматизации для Google SRE

Все эти преимущества и компромиссы актуальны не только для компании Google, где многие предпочитают использовать автоматизацию, но и для всех остальных. Наше стремление к автоматизации частично вытекает из наших бизнес-задач: создаваемые нами продукты и сервисы распространены по всей планете, и у нас нет времени заниматься обучением такого количества персонала⁴¹. Для действительно крупных сервисов факторы постоянства, скорости и надежности имеют решающее значение при обсуждении компромиссов, на которые нам придется пойти при выполнении автоматизации.

Еще один аргумент в пользу автоматизации применительно конкретно к Google — наше сложное, но на удивление унифицированное окружение промышленной эксплуатации систем. Хотя в некоторых организациях значительная часть эксплуатируемого оборудования может не иметь общедоступного API, либо для него не предоставляют нужный исходный код, либо возникают иные препоны, осложняющие контроль над промышленной средой, в Google мы стараемся избегать подобного. Если поставщик не предоставлял API системы, мы писали API сами. Несмотря на то что в краткосрочной перспективе бывает проще купить ПО для решения какой-либо конкретной задачи, мы предпочитаем писать программы самостоятельно, поскольку в долгосрочной перспективе польза от такого API собственной разработки будет намного больше. Мы долго пытались устраниТЬ проблемы, возникавшие при автоматическом управлении системами, а потом взяли и построили такую систему управления сами. Учитывая то, как в Google организовано управление исходным кодом [Potvin, 2016], нам намного проще обеспечить доступность этого кода практически во всех системах, с которыми SR-инженер имеет дело. Кроме того,

упрощается задача «владения промышленно эксплуатируемым продуктом», поскольку мы сами контролируем весь стек.

Несмотря на то что компания Google стремится максимально автоматизировать управление системами, реальность заставляет нас вносить в этот подход некоторые корректизы. Нет необходимости автоматизировать каждый компонент каждой системы, и не каждый способен или склонен разрабатывать решение для автоматизации в текущий момент времени. Некоторые наши системы создавались как прототипы, не предназначенные для длительного использования и поэтому не предусматривавшие автоматизацию. Выше приводилась наша максималистская точка зрения по этому вопросу, и она оказалась достаточно успешной конкретно в среде Google. В целом мы предпочитаем по возможности создавать автоматизированные платформы или позиционировать себя способными создать их с течением времени. Мы считаем, что такой подход, основанный на автоматизированных платформах, необходим для обеспечения управляемости и масштабируемости.

Применение автоматизации

В нашей отрасли *автоматизация* — это термин, который, как правило, применяется к созданию программ для решения широкого круга задач. При этом цели создания таких программ и сами решаемые с их помощью задачи зачастую отличаются друг от друга. В широком понимании автоматизация — это «мета-ПО», то есть программы, предназначенные для работы с программами.

Как мы говорили ранее, автоматизация может применяться для решения различных задач. Вот неполный список примеров:

- создание пользовательских учетных записей;
- включение и отключение кластеров сервисов;
- подготовка к установке или к выводу из эксплуатации ПО или оборудования;
- развертывание и установка новых версий ПО;
- изменение конфигурации ПО во время его работы;
- особый случай изменения конфигурации во время работы: изменения согласно зависимостям.

Этот список можно продолжать *до бесконечности*.

Применение автоматизации в Google SRE

В компании Google можно встретить все перечисленные варианты использования автоматизации и даже больше. Однако служба SRE в первую очередь стремится поддерживать инфраструктуру, а не обеспечивать качество проходящих через нее данных. Это разграничение может быть не вполне понятным, поэтому поясним на примере. Нас побеспокоит тот факт, что в результате очередной установки ПО пропала половина данных, поэтому мы разошлем оповещения обо всех подобных масштабных изменениях данных. Но наши инженеры вряд ли будут писать программу, меняющую содержимое некоторого количества учетных записей системы. Таким образом, для службы SRE автоматизация важна как средство управления жизненными циклами систем (например, развертыванием сервиса на новом кластере), а не их данными.

В этом смысле служба SRE в плане автоматизации делает примерно то же, что и другие люди и организации, но мы пользуемся другими средствами и ставим другие цели (это будет рассмотрено далее).

Такие средства, как Puppet, Chef, cfengine и даже Perl, доступны широкому кругу пользователей. Они предоставляют возможность автоматизировать выполнение ряда задач, различаясь в основном уровнями абстракции компонентов, которые применяются при автоматизации. В языках программирования вроде Perl можно работать на уровне вызовов POSIX, что в теории дает практически неограниченные возможности для автоматизации с использованием всех интерфейсов, доступных системе⁴². Chef и Puppet предлагают готовые абстракции «из коробки», с помощью которых можно манипулировать сервисами или другими высокоуровневыми объектами. Перед нами классический компромисс: абстракции более высокого уровня проще для управления и понимания, но, если вам попадется «дырявая» («протекающая») абстракция, ошибки будут появляться систематически.

Например, мы часто подразумеваем, что отправка нового файла программы в кластер является атомарной операцией: в итоге в кластере либо останется старая, либо появится новая версия. Однако в действительности поведение оказывается несколько сложнее. Сбой может произойти в сети кластера, в компьютере либо на уровне управления кластером (это приведет к тому, что система останется в нестабильном состоянии). В зависимости от ситуации новые исполняемые файлы могут быть проиндексированы, но не отправлены; или отправлены, но кластер не перезапустится; или кластер перезапустится, но его нельзя будет проверить. Лишь некоторые абстракции успешно справляются с подобными

ситуациями — они, как правило, прекращают работу и требуют вмешательства инженера. Совсем плохо построенные системы автоматизации не делают даже этого.

У службы SRE для автоматизации есть свои подходы и средства, одни из которых выглядят скорее как базовые инструменты для развертывания и установки программ без детального описания высокоуровневых сущностей, другие — как языки для описания развертывания сервисов (и т.д.) на очень абстрактном уровне. Последние характеризуются, как правило, возможностью повторного использования решений и независимостью от платформы, но высокая сложность нашей среды иногда позволяет применять только инструменты первого типа.

Уровни автоматизации

Все описанные способы автоматизации очень удобны (как и сама платформа автоматизации), но в идеальном мире внешняя автоматизация была бы не нужна вовсе. Действительно, вместо системы, которая должна иметь внешнюю стыковочную логику, лучше иметь систему, которая *вообще не нуждается в стыковочной логике* — не только потому, что внутренняя реализация более эффективна (хотя такая эффективность тоже полезна), но и потому, что система разработана таким образом, чтобы не нуждаться в ней. Для этого нужно выделить ситуации использования стыковочной логики — как правило, это будут «первоочередные» действия с системой вроде добавления учетных записей или запуска системы — и найти способ обратиться к ним непосредственно из приложения.

Рассмотрим другой пример. Большинство решений по автоматизации процесса запуска систем в Google сталкиваются с проблемами, поскольку они сопровождаются отдельно от

основной системы и зачастую становятся неактуальны, когда, например, основная система изменяется, а они — нет. Несмотря на все усилия, попытки более тесно связать решения по автоматизации и основную систему зачастую проваливаются из-за несбалансированной расстановки приоритетов. Разработчики продукта отказываются выполнять контрольное развертывание после каждого изменения. Кроме того, критически важная часть автоматизации, которая выполняется лишь изредка и которую трудно протестировать, зачастую оказывается особенно уязвимой из-за более продолжительного цикла обратной связи. Восстановление после отказа кластера — один из классических примеров редко выполняемой автоматизации. Кластер может восстанавливаться после отказа раз в несколько месяцев — в общем, недостаточно часто для того, чтобы стало заметно несоответствие экземпляров сервиса. Автоматизация развивается по следующему пути.

1. *Нет автоматизации.* Для главной базы данных восстановление после отказа выполняется вручную на всех площадках.
2. *Управляемая извне автоматизация, характерная для данного типа систем.* SR-инженер создал сценарий для восстановления в своей «домашней» директории.
3. *Управляемая извне общая автоматизация.* SR-инженер добавляет поддержку базы данных в общий сценарий восстановления, которым пользуются все.
4. *Управляемая изнутри автоматизация, специфичная для данного типа систем.* База данных поставляется с собственным сценарием восстановления.

5. Системы, для которых автоматизация не требуется. База данных обнаруживает проблемы и автоматически преодолевает отказы без вмешательства человека.

Специалисты из отдела SRE ненавидят выполнять работу вручную, поэтому мы стараемся создавать системы, для которых она не требуется. Однако иногда выполнения операций вручную не избежать.

Существует еще одна разновидность автоматизации, когда изменения применяются не только в той части конфигурации, которая связана с конкретной системой, но и во всей конфигурации в целом. В высокоцентрализованных проприетарных производственных средах, в том числе в компании Google, многие изменения затрагивают не только один конкретный сервис. Сюда относятся, например, изменения в цепочке серверов Chubby, изменение флагов в клиентской библиотеке Bigtable для повышения надежности доступа и т.д. Тем не менее ими нужно безопасно управлять и откатывать их по мере необходимости. Начиная с определенного объема изменений, вносить их вручную в масштабах всей производственной среды становится невозможно, и даже до этого момента ручное выполнение большого количества изменений, которые либо тривиальны, либо успешно реализуются с помощью стратегии «перезапустить и проверить», оказывается пустой тратой времени.

Далее мы приведем примеры из нашей практики и на них постараемся более подробно проиллюстрировать озвученные идеи. Первый пример показывает, как благодаря продуманным действиям мы смогли достичь нирваны всех SR-инженеров: автоматизировали процесс так, что наше участие в нем не требовалось.

Исключаем себя из процесса: автоматизируем все!

Долгое время данные продуктов Ads хранились в БД MySQL. Поскольку они, очевидно, требовали высокой надежности, за инфраструктуру этой системы отвечала команда SR-инженеров. С 2005 по 2008 год база данных для приложения Ads находилась в стабильном и управляемом состоянии. Например, мы автоматизировали худшую часть рутинной работы (но не всю) по реплицированию. Мы были уверены, что база данных хорошо управляется и что сделано все возможное с точки зрения оптимизации и масштабирования. Однако, как только стало удобно выполнять повседневные операции, члены команды задумались о следующем уровне разработки системы: перейти с MySQL на созданную в Google систему-планировщик для кластеров под названием Borg.

Мы рассчитывали, что такой переход предоставит нам два основных преимущества:

- *полностью исключит необходимость обслуживания машин с развернутыми на них репликами — Borg будет автоматически обрабатывать запуск и перезапуск новых и аварийно завершившихся задач;*
- *даст возможность размещать несколько экземпляров MySQL на одной машине — Borg позволит более эффективно пользоваться ресурсами благодаря контейнерам.*

В конце 2008 года мы развернули экспериментальный экземпляр базы данных MySQL на Borg. К сожалению, это сопровождалось значительными трудностями. Основное свойство Borg заключалось в том, что задачи перемещались автоматически. Как правило, это происходило один-два раза в неделю. Такая частота была приемлемой для наших реплик баз

данных, но оказывается неприемлемой для мастеров (контролеров) перемещений.

В то время процесс восстановления после отказа мастера длился 30–90 минут для одного экземпляра. Поскольку мы запускали процесс в распределенных системах и должны были перезапускать их для обновления ядра, в дополнение к обычному количеству сбоев мы ожидали некоторое количество не связанных друг с другом отказов каждую неделю. Этот фактор вкупе с количеством серверов, на которых размещалась наша система, означал, что:

- выполнение вручную процедуры восстановления после отказа потребует значительного количества человеко-часов и в лучшем случае даст нам доступность 99 %, недостаточную для выполнения бизнес-требований к продукту;
- чтобы уложиться в бюджет времени недоступности сервиса, после каждого отказа система должна восстанавливаться не позже чем через 30 секунд. Оптимизировать зависимую от человека процедуру, заставив ее выполняться менее чем за 30 секунд, было невозможно.

В итоге у нас оставался единственный вариант — автоматизировать процедуру восстановления после сбоев, и не только ее.

В 2009 году SR-инженеры, работающие с продуктами Ads, создали программу-демон⁴³ для восстановления после сбоев. Его назвали Decider. В течение 95 % времени работы он мог выполнять процедуру восстановления для MySQL менее чем за 30 секунд. Благодаря появлению Decider использование MySQL на Borg наконец стало реальностью. Мы больше не пытались

оптимизировать нашу инфраструктуру так, чтобы процедура восстановления требовалась как можно реже. Мы пришли к пониманию, что сбои неизбежны, и поэтому нужно оптимизировать сам процесс восстановления после сбоев, автоматизировав его.

Итак, автоматизация позволила нам получить базу данных MySQL с лучшим в мире показателем доступности, но она должна была перезапускаться два раза в неделю, и за это приходилось платить свою цену. Все наши приложения нужно было доработать, внедрив гораздо больше логики для обработки ошибок. При разработке с использованием MySQL подразумевается, что сам экземпляр MySQL будет наиболее стабильной частью стека. Поэтому подобный переход означал бы, что необходимо настроить прочие программы вроде стандарта JDBC так, чтобы он был более терпимым к нашей склонной к сбоям системе. Однако преимущества перехода на Borg с помощью Decider того стоили. После этого перехода время, которое наша команда тратила на выполнение рутинных операционных задач, сократилось на 95 %. Процедура восстановления после сбоев была автоматизирована, поэтому ошибка при выполнении одной задачи базы данных больше не требовала вызова инженера.

Основное преимущество этого нового подхода к автоматизации заключалось в том, что у нас появилось гораздо больше свободного времени. И его можно было потратить на улучшение других частей нашей инфраструктуры. Таким образом, чем больше времени мы экономили, тем больше было возможностей оптимизировать и автоматизировать приложения. В результате мы смогли автоматизировать работу по изменению схем, что привело к снижению общего количества операций для обслуживания базы данных Ads на 95 %. Наше аппаратное обеспечение также было улучшено.

Переход на MoB в сочетании с Decider высвободил значительное количество ресурсов — мы смогли выполнять задачи для нескольких экземпляров базы MySQL на одних и тех же серверах, что повысило коэффициент использования нашего оборудования. В общей сложности мы смогли высвободить около 60 % машинного парка. После этого наша команда не испытывала недостатка в аппаратных и инженерных ресурсах.

Этот пример демонстрирует целесообразность принятого нами решения — вместо того чтобы заменять отдельные существующие процедуры, выполняемые вручную, мы провели дополнительную работу по созданию целой платформы. Следующий пример основан на нашей инфраструктуре кластеров. Он иллюстрирует несколько наиболее сложных проблем, с которыми вы можете столкнуться при попытке *повсеместной* автоматизации.

Облегчаем жизнь: автоматизируем процесс запуска кластера

Десять лет назад команда SR-инженеров, следившая за инфраструктурой кластеров, каждые несколько месяцев увеличивалась на одного нового работника. Как оказалось, примерно с такой же частотой мы запускали новый кластер. Поскольку включение сервиса в новом кластере позволяло продемонстрировать новым сотрудникам внутренние составляющие сервиса, это стало стандартной практикой при обучении.

Для подготовки кластера к использованию требовалось выполнить следующие действия.

1. Оборудовать здание дата-центра системами питания и охлаждения.

2. Установить основные коммутаторы и настроить соединения с объединяющей магистралью.
3. Установить несколько первых стоек с серверами.
4. Сконфигурировать базовые сервисы вроде DNS и установщиков, затем сконфигурировать сервисы блокировки, хранения и вычисления.
5. Развернуть остальные стойки серверов.
6. Распределить ресурсы сервисов, с которыми работают пользователи, чтобы команды инженеров могли настроить эти сервисы.

Шаги 4 и 6 оказались очень сложными. Несмотря на то что базовые сервисы вроде DNS были относительно простыми, подсистемы хранения и вычислений в то время еще находились в разработке, поэтому новые флаги, компоненты и решения по оптимизации добавлялись еженедельно.

У некоторых сервисов было больше сотни разных подсистем-компонентов, и каждая из них имела сложную сеть зависимостей. Неверное конфигурирование одной подсистемы либо нестандартное конфигурирование системы или компонента могли привести к отключению, затрагивающим пользователей.

Например, многопетабайтный кластер Bigtable по соображениям быстродействия был сконфигурирован так, чтобы не использовать первый (журналирующий) диск 12-дисковых систем. Год спустя одна из программ автоматизации решила, что если первый диск компьютера не используется, то и остальные не используются для хранения данных. Следовательно, компьютер можно полностью очистить и

настроить с нуля. Все данные Bigtable в мгновение ока были уничтожены. К счастью, у нас было несколько актуальных резервных копий этих данных, но такие сюрпризы всегда неприятны. При автоматизации необходимо очень внимательно относиться к неявным признакам «опасности» или «безопасности» выполняемых действий.

Создавая первые решения по автоматизации, мы стремились ускорить включение кластера в работу. При этом использовался интерактивный доступ по протоколу SSH, который позволял распределять пакеты и решать другие проблемы, возникшие при инициализации сервиса. Стратегия поначалу казалась удачной, но подобные «вольные» сценарии приводили к увеличению технического долга.

Выявление несоответствий с помощью Prodtest

По мере роста количества кластеров для некоторых из них потребовались настроенные вручную флаги и ручная настройка в целом. В результате команды тратили все больше и больше времени на отслеживание трудно обнаруживаемых несоответствий в конфигурации. Если флаг, ускоряющий доступ к GFS при работе с журналами, попадал в стандартные шаблоны, это могло привести к нехватке памяти под нагрузкой при большом количестве файлов. Раздражающие несоответствия, отнимающие много времени на исправление, возникали практически при каждом масштабном изменении конфигурации.

Креативные (но нестабильные) сценарии оболочки (скрипты shell — общее название для различных версий командных интерпретаторов или оболочек в Unix-системах. — *Примеч. пер.*), которые мы использовали для конфигурирования кластеров, не масштабировались ни в зависимости от количества людей, желающих внести

изменения, ни в зависимости от общего количества требуемых изменений кластера. Эти сценарии оболочки также не могли дать ответ на следующие серьезные вопросы, из-за которых невозможно было объявить, что сервис готов принимать пользовательский трафик.

- Доступны ли все компоненты, от которых зависит сервис, и корректно ли они сконфигурированы?
- Согласуются ли все конфигурации и пакеты с другими развернутыми конфигурациями и пакетами?
- Может ли команда подтвердить, что каждое исключение, сделанное при конфигурировании, было осознанным?

Система Prodtest (Production test) стала оригинальным решением для предотвращения этих нежелательных сюрпризов. Мы расширили фреймворк модульного тестирования (юнит-тестирования), написанный на Python, чтобы иметь возможность проводить юнит-тесты для реальных сервисов. Для юнит-тестов характерно наличие зависимостей — можно выполнять цепочки тестов, и сбой в одном тесте прекратит выполнение всей цепочки. Рассмотрим в качестве примера тест, показанный на рис. 7.1.

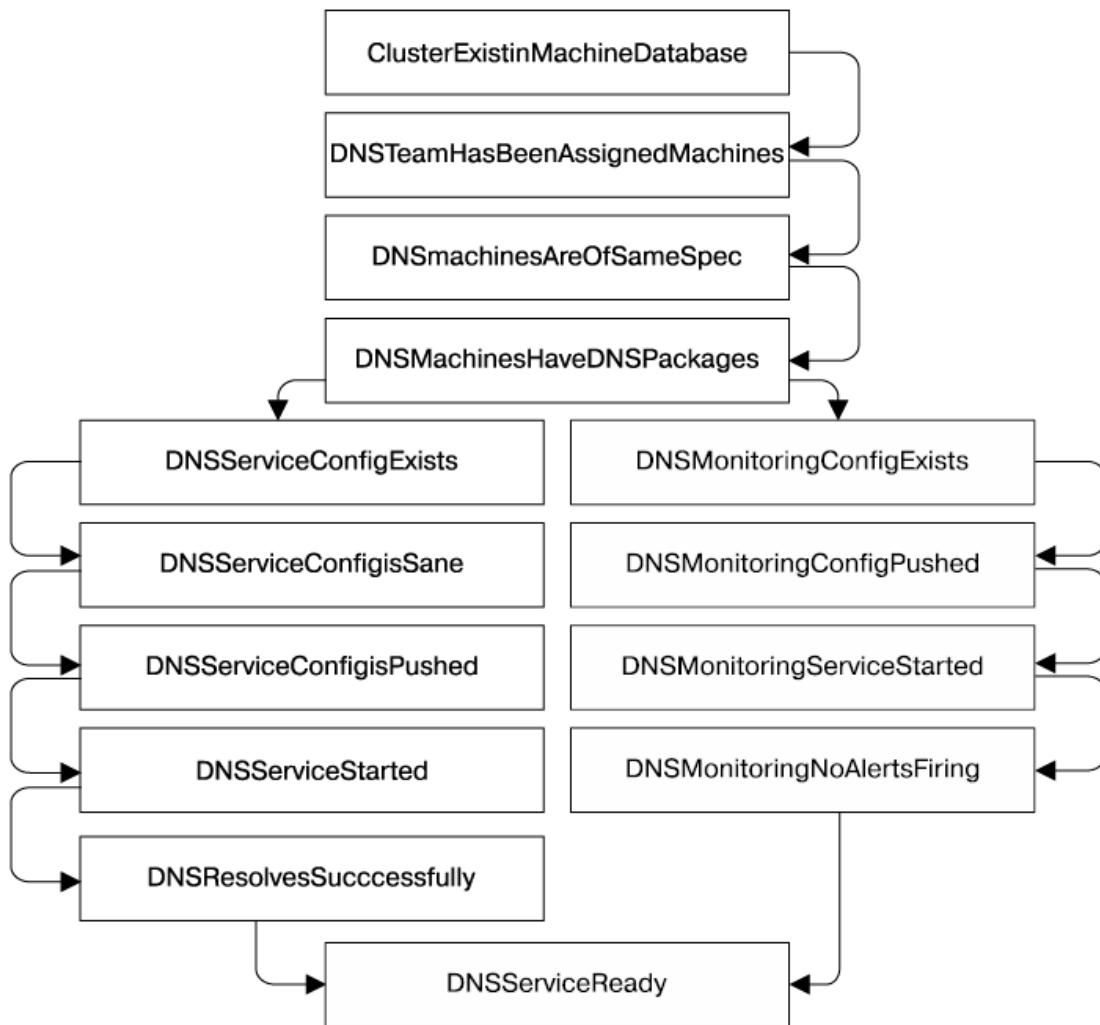


Рис. 7.1. Prodtest для сервиса DNS, который показывает, как один непройденный тест прекращает выполнение всей цепочки

У каждой команды инженеров Prodtest, получив имя кластера, проверял сервисы этой команды, находящиеся в заданном кластере. Последующие дополнения позволили нам сформировать граф юнит-тестов и их состояний. Благодаря этой функциональности инженеры могли быстро проверять, во всех ли кластерах сервис сконфигурирован корректно и если нет, то почему. На графике выделялся шаг, на котором возник сбой, и непройденный юнит-тест на Python выводил на экран развернутое сообщение об ошибке.

Каждый раз, когда команда наблюдала задержку, вызванную неожиданным несоответствием конфигураций, информация об этом происшествии отправлялась в их Prodtest. Это гарантировало, что в будущем подобные проблемы будут обнаружены быстрее. SR-инженеры гордились тем, что смогли обеспечить для своих пользователей отложенную работу сервисов. Как новые, так и уже существующие сервисы в обновленной конфигурации могли надежно обслуживать нагрузку промышленной эксплуатации.

Впервые наши менеджеры проектов могли спрогнозировать, когда именно кластер начнет работать, и хорошо понимали, почему для каждого кластера требуется шесть или более недель для перехода из состояния «готов к работе» к состоянию «обработка трафика в реальном времени». Но внезапно SR-инженеры получили от высшего руководства задание: *через три месяца пять новых кластеров должны перейти в состояние «готов к работе» в один и тот же день. Пожалуйста, запустите их в течение недели.*

Идемпотентное разрешение несоответствий

Задача «запуска за неделю» казалась невыполнимой. У нас были десятки тысяч строк скриптов, написанные десятками команд. Мы легко могли сказать, насколько неготовым был любой конкретный кластер, но подготовка кластера должным образом означала, что десятки команд должны искать сотни ошибок. И нам лишь оставалось надеяться, что эти ошибки будут исправлены.

Как оказалось, эти проблемы можно решить быстрее, перейдя от парадигмы «юнит-тесты Python используются для поиска несоответствия конфигураций» к парадигме «код Python применяется для исправления несоответствия в конфигурации».

Юнит-тесту известно, какой кластер мы исследуем, а также какой именно тест еще не пройден, поэтому мы объединили тесты с кодом, исправлявшим проблему. Если каждое решение было написано так, чтобы быть идемпотентным⁴⁴ и удовлетворять все зависимости, то решить проблему можно было просто и безопасно. Требование к идемпотентности исправлений заключалось в том, что команды могут запускать собственный «скрипт починки» каждые 15 минут, не боясь повредить конфигурацию кластера. Если тест команды DNS заблокирован в конфигурации Machine Database нового кластера, то, как только кластер появится в базе данных, тесты и решения команды DNS начнут работать.

В качестве примера рассмотрим тест, проиллюстрированный на рис. 7.2. Если не проходит тест `TestDnsMonitoringConfigExists`, как это показано на схеме, мы можем вызвать код шага `FixDnsMonitoringCreateConfig`, который получает конфигурацию из базы данных, а затем проверяет основной конфигурационный файл в нашей системе контроля версий. Затем со второй попытки тест `TestDnsMonitoringConfigExists` выполняется, и теперь можно попробовать выполнить тест `TestDnsMonitoringConfigPushed`. Если этот тест не проходит, то выполняется код шага `FixDnsMonitoringPushConfig`. Если исправление не достигает успеха после нескольких попыток, то оно считается некорректным, и автоматизированное выполнение тестов прекращается, оповещая пользователя.

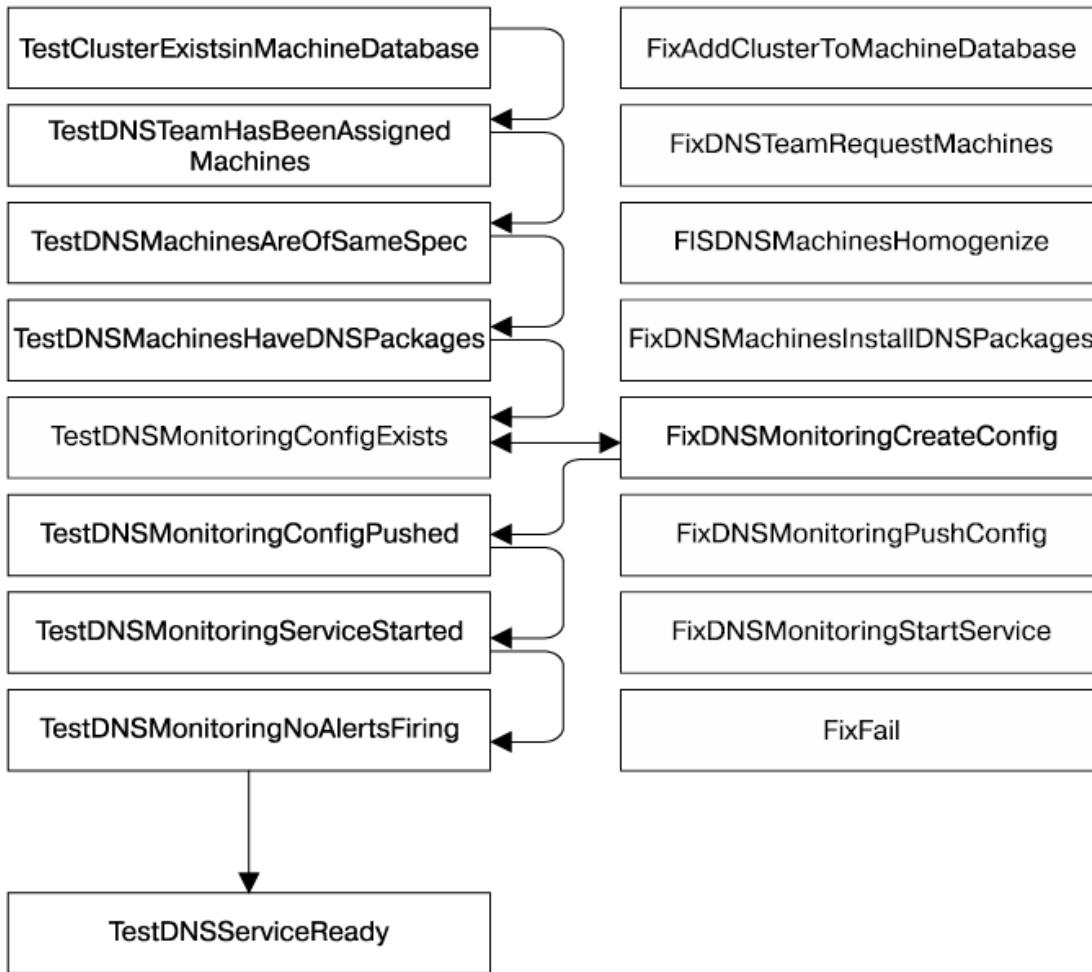


Рис. 7.2. Prodtest для сервиса DNS, показывающий, что один невыполненный тест инициирует применение только одного исправления

Вооружившись этими скриптами, небольшая группа инженеров могла гарантированно обеспечить переход от состояния «Сеть работает, и машины внесены в базу данных» к состоянию «Обслуживаем 1 % трафика сервисов поиска и рекламы». На тот момент наше решение казалось вершиной технологии автоматизации.

Оглядываясь назад, мы можем сказать, что такой подход имел множество недостатков. Из-за большой задержки между этапами выполнения теста, решения и выполнения второго теста появились «капризные» тесты, которые иногда работали, а иногда — нет. Не все решения были естественным образом

идемпотентными, поэтому «капризный» тест, за которым следовало решение, мог сделать систему нестабильной.

Движение к специализации

Процессы автоматизации отличаются друг от друга тремя характеристиками.

- *Компетентностью*, то есть способностью давать корректные, точные результаты.
- *Задержкой (временем ожидания)* — временем до выполнения всех шагов.
- *Релевантностью*, или долей реальных эксплуатируемых процессов, охваченных автоматизацией.

Мы начали с процесса, который имел высокую компетентность (запускался и обслуживался владельцами сервиса), высокую задержку (владельцы сервиса выполняли процессы в свое свободное время или поручали их новым инженерам) и высокую релевантность (владельцы сервисов знали, когда меняется ситуация в реальном мире, и могли исправить решение по автоматизации).

Для того чтобы ускорить ввод кластера в строй, команды владельца сервиса инструктировали «команду запуска», какое именно средство автоматизации следует применять. «Команда запуска» использовала тикеты в начале каждой фазы старта кластера, поэтому мы смогли отследить оставшиеся задачи и узнать, кто должен их выполнять. Ввод кластера в строй мог быть существенно ускорен, если бы люди, участвующие в процессе, могли взаимодействовать непосредственно, в пределах одной комнаты. Наконец мы определили, что

представляет собой надежный, точный и оперативный процесс по автоматизации!

Но такое состояние длилось недолго. Реальный мир хаотичен: ПО, конфигурация, данные и пр. постоянно меняются, что ежедневно приводит к тысячам изменений в соответствующих системах. Люди, которых больше всего затрагивали ошибки в автоматизации, перестали быть экспертами в данной области. В результате автоматизация стала менее релевантной (новые этапы в ней не учитывались) и менее компетентной (могла дать сбой при появлении новых флагов). Однако такое снижение качества работы повлияло на ее скорость лишь через некоторое время.

Код решения по автоматизации, как и код юнит-тестов, отмирает, если обслуживающая его команда не синхронизирует его с базой кода, которую он должен покрывать. Окружающий его мир меняется: команда, отвечающая за DNS, добавляет новые параметры конфигурации, команда, отвечающая за хранение данных, меняет имена пакетов, а команда, отвечающая за работу с сетями, должна поддерживать новые устройства.

Освободив работающие с сервисами команды от обязанности обслуживать код средств автоматизации, мы утратили организационные стимулы.

- Команда, чья основная задача — ускорять текущий процесс ввода кластера в строй, не имеет стимула снижать «технический долг» команды — владельца сервиса, которая в дальнейшем выпустит его в промышленную эксплуатацию.
- Команда, не использующая автоматизацию, не имеет стимула создавать легко автоматизируемые системы.

- Продукт-менеджер, на чье расписание плохая автоматизация не влияет, всегда будет отдавать предпочтение новому функционалу, не стремясь к упрощению и автоматизации.

Наиболее функциональные инструменты обычно создаются теми людьми, которые работают с ними. Этот аргумент можно приводить и для ответа на вопрос, почему командам, разрабатывающим продукт, полезно наблюдать за своими системами в промышленной эксплуатации.

Процесс ввода кластеров в строй вновь перестал быть оперативным, компетентным и безошибочным — это худшее, что с ним могло случиться. Однако требование к безопасности, не связанное с нашей проблемой, позволило нам выбраться из этой ловушки. Большая часть распределенной автоматизации в то время зависела от SSH. Это было неправильно с точки зрения безопасности, поскольку для запуска многих команд требовалось права администратора. Понимание возможных проблем безопасности заставило нас уменьшить права SR-инженеров, сделав их минимально необходимыми для выполнения работы. Нам пришлось заменить sshd аутентифицируемым, управляемым ACL (Access Control List — список управления доступом), основанным на RPC-демоне Local Admin Daemon, также известном как Admin Server. Он позволял настроить права доступа для внесения локальных изменений. В результате никто не мог бесследно поменять настройки сервера. Изменения в Local Admin Daemon и репозитории пакетов проходили с разбором кода, что сделало очень маловероятным превышение своих полномочий. Тот, кто получает доступ к установочным пакетам, не сможет просматривать находящиеся на тех же машинах журналы. Admin Server записывает в журнал необходимые для последующих отладки и аудита безопасности сведения:

инициатора вызова RPC, все параметры и результаты всех вызовов.

Запуск кластера, ориентированный на сервисы

На следующем этапе демон Admin Server стал частью технологического процесса для команд, поддерживающих экземпляры демона, ориентированные как на сервисы (для установки пакетов и перезагрузки), так и на кластеры (для действий вроде отвода трафика или включения сервиса). SR-инженеры перешли от написания скриптов shell в домашних каталогах к построению взаимопросматриваемых RPC-серверов с подробными ACL.

После того как мы поняли, что процесс ввода кластера в строй должен проводиться владельцами сервисов, мы решили попробовать рассматривать его как задачу вида Service-Oriented Architecture (SOA). В этом случае владельцы сервиса должны нести ответственность за создание экземпляров демона Admin Server для обработки удаленных вызовов процедур (RPC) запуска и остановки кластера, к которым обратится система, знающая, когда кластеры переходят в состояние готовности. В свою очередь, каждая команда должна предоставить контракт (API), необходимый для автоматизации процесса ввода кластера в строй, имея при этом возможность менять реализацию по своему усмотрению. Как только кластер приходил в состояние «готов к работе», автоматизация отправляла RPC на каждый Admin Server, который имел отношение к процессу.

Теперь мы получили оперативный, компетентный и точный процесс. Особенno важно, что этот процесс продолжает успешно применяться, хотя количество изменений, команд и сервисов ежегодно увеличивается в два раза.

Как упоминалось ранее, эволюция процесса ввода кластера в строй прошла следующий путь.

1. Ручные действия, выполняемые оператором (нет автоматизации).
2. Написанное оператором решение, специфичное для определенной системы.
3. Общая автоматизация, поддерживаемая извне.
4. Автоматизация, специфичная для определенной системы и поддерживаемая изнутри.
5. Автономные системы, которым не требуется вмешательство человека.

Эта эволюция была успешной, но пример Borg демонстрирует еще один подход к решению проблемы автоматизации.

Borg: появление компьютера размером с дом

Рассмотрим историю развития наших систем управления кластерами⁴⁵, на примере которой тоже можно увидеть, как менялось наше отношение к автоматизации и наше понимание того, когда и где ее лучше разворачивать. Как и при использовании MySQL для Borg (когда мы увидели, что можно успешно преобразовать ручные операции в автоматические) и во время ввода кластера в строй (когда мы узнали о проблемах непродуманного планирования автоматизации), разработка принципов управления кластерами преподнесла еще один урок правильной автоматизации. Как и в предыдущих двух

примерах, мы создали сложную систему, которая стала результатом длительной эволюции более простых объектов.

Кластеры компании Google изначально строились примерно так же, как и небольшие компьютерные сети того времени: они представляли собой стойки с оборудованием, имеющим конкретное назначение и неоднородную конфигурацию. Инженеры должны были входить в систему на одном из нескольких хорошо известных мастер-компьютеров для выполнения административных задачий. На этих мастер-компьютерах располагались «золотые» бинарные файлы и хранилась конфигурация. Поскольку у нас был только один провайдер площадки размещения кластера, в большинстве случаев при именовании файлов мы неявно подразумевали эту площадку. По мере роста «производственных мощностей» мы начали использовать несколько кластеров, поэтому нам пришлось начать применять разные домены (имена кластеров). Потребовалось ввести файл, в котором описывалось, что делает каждый компьютер. Это позволило сгруппировать оборудование в рамках некоторой неформальной стратегии именования. Такой файл-дескриптор в сочетании с аналогом «параллельного» SSH позволил нам перезагружать (к примеру) все поисковые машины за раз. В это время было нормой получить тикет вида «на компьютере x1 выполнен поиск, краулер может воспользоваться компьютером».

Началась разработка решения по автоматизации. Изначально оно состояло из простых сценариев Python, предназначенных для выполнения таких несложных операций, как:

- управление сервисами: поддержание их доступности (например, перезагрузка после ошибок доступа к памяти);

- выяснение, какие сервисы должны выполняться на конкретных компьютерах;
- анализ сообщений журнала: соединение по SSH с каждым компьютером и поиск по регулярным выражениям.

Автоматизация в конечном счете превратилась в соответствующую базу данных, которая отслеживала состояние системы. Кроме того, появились более сложные инструменты мониторинга. Имея под рукой множество доступных решений по автоматизации, мы теперь могли автоматически управлять жизненным циклом компьютеров: мы могли заметить, что компьютер прекратил работу, удалить сервисы, направить их в ремонт и восстановить конфигурацию по их возвращении.

Оглядываясь назад, нужно отметить, что такая автоматизация была удобна, но серьезно ограничена тем, что абстракции системы были жестко привязаны к физическим машинам. Требовалась новая методика, поэтому на свет появилась система Borg [Verma et al., 2015], позволившая перейти от предыдущего подхода, когда мы имели относительно статические назначения «хост/порт/задача», к подходу, когда набор компьютеров рассматривался как управляемое множество ресурсов. Идея рассматривать управление кластером как сущность, для которой использовались вызовы API, как некий центральный координатор была основополагающей для успеха. Это позволило нам высвободить дополнительные ресурсы для повышения эффективности, гибкости и надежности: в отличие от предыдущей модели «владения» машинами Borg позволял запланировать, например, пакетную обработку и выполнение интерактивных процессов пользователей на одном и том же компьютере.

Наличие этой функциональности в итоге привело к тому, что мы смогли выполнять автоматические длительные обновления операционной системы, прикладывая очень небольшие⁴⁶ усилия, которые не возрастили при изменении общего количества развернутых «промышленных» систем. Небольшие отклонения в работе компьютера теперь исправляются автоматически; постоянное обеспечение доступности и управление жизненным циклом больше не лежат на плечах SRE-инженеров. Тысячи компьютеров появляются, прекращают работу и отправляются в ремонт, и это не требует никаких усилий со стороны отдела SRE. Бен Трейнор Слосс сказал: «Считая это задачей программного обеспечения, мы за счет первоначальной автоматизации выиграли достаточно времени, чтобы сделать механизм управления кластерами автономным, а не автоматическим. Мы достигли этой цели, реализуя идеи, связанные с распределением данных, API, архитектурами hub-and-spoke и разработкой классических распределенных систем, что позволило справиться и с уровнем управления инфраструктурой».

Здесь можно провести интересную аналогию: мы можем установить прямую связь от случая единственного компьютера к разработке абстракции для управления кластерами. С этой точки зрения перенос выполнения программы на другой компьютер очень похож на перенос с одного процессора на другой: конечно, эти вычислительные ресурсы могут находиться на другом конце сетевого подключения, но насколько это важно? Здесь перераспределение задач выглядит как неотъемлемая функция системы, а не что-то, что можно «автоматизировать», — люди все равно не могут реагировать достаточно быстро. Аналогично в рамках этой метафоры ввод в строй кластера — всего лишь дополнительная планируемая

производительность, похожая на добавление диска или оперативной памяти для одного компьютера. Однако мы не ожидаем, что одиночный компьютер будет продолжать работать после отказа нескольких его компонентов. В этом его отличие от «глобального компьютера», который, начиная с определенного «размера», может и *должен* иметь способность самовосстанавливаться, поскольку для него будет статистически гарантировано большое количество сбоев ежесекундно. Это подразумевает, что по мере перехода систем вверх по иерархии — от систем, инициируемых вручную, через системы, инициируемые автоматически, до автономных систем — для обеспечения работоспособности системы необходим некоторый объем самодиагностики.

Основное качество — надежность

Конечно, для эффективного обнаружения проблем подробности работы системы, на которые опирается самодиагностика, должны быть понятны людям, управляющим системой. Аналогичные дискуссии о влиянии автоматизации в некомпьютерных областях — например, в авиации⁴⁷ или в промышленности — зачастую указывают на оборотную сторону высокоэффективной автоматизации⁴⁸: люди-операторы все больше устраняются от прямого контакта с системой по мере того, как автоматизация со временем охватывает все большее количество повседневных задач. Далее неизбежно возникает ситуация, когда автоматизация дает сбой и люди уже не могут успешно работать с системой. Они не могут быстро и эффективно отреагировать на ситуацию из-за недостатка практики, и их представление о том, что система *должна* делать, больше не соответствуют действительности⁴⁹. Такая ситуация чаще возникает в неавтономных системах, где,

однако, автоматизированы все ручные действия. Считается, что в любой момент работу можно выполнить вручную, и для этого не будет никаких препятствий. К сожалению, со временем это утверждение становится неверным: ручные действия можно выполнить не всегда, поскольку соответствующая функциональность больше не существует.

Мы также сталкивались с ситуациями, когда автоматизация наносила ущерб (см. врезку «Автоматизация: появление сбоев при масштабировании» ниже). Но, по опыту компании Google, сейчас появилось гораздо больше систем, для которых автоматизация или автономное поведение уже не являются чем-то дополнительным. Это также применимо и для ваших систем по мере их масштабирования, однако существует множество весомых аргументов для того, чтобы создавать автономные системы независимо от их размера. Доступность — ключевое качество, а автономное, устойчивое поведение — один из способов ее достижения.⁵⁰

Автоматизация: появление сбоев при масштабировании

Компания Google запустила более дюжины собственных крупных data-центров, но мы зависим и от оборудования многих сторонних площадок, или «колокаций» (colos). Наши машины на этих площадках используются как оконечные для большинства входящих соединений или в качестве кэша нашей собственной Content Delivery Network, позволяя снизить задержку отклика для конечных пользователей. Какое-то количество этих стоек постоянно вводят в работу или списывают. Оба этих процесса практически автоматизированы. Один из этапов списания стойки состоит в

перезаписывании³ содержимого жестких дисков всех машин в ней, после чего независимая система проверяет успешность удаления данных. Мы называем этот процесс Diskerase (Очистка диска).

Однажды процесс автоматизации, обеспечивающий отключение определенной стойки, дал сбой, но уже после того, как успешно завершился этап Diskerase. Затем процесс

демонтажа кластера снова запустился с начала, чтобы выполнить отладку. При попытке запустить Diskerase для оборудования, находящегося в стойке, процесс автоматизации обнаружил, что список машин, требующих очистки, уже пуст (что было верно). К сожалению, пустой список был воспринят как особое значение и интерпретирован как «все машины». Это означало, что процесс автоматизации запустил выполнение Diskerase почти для всех наших машин.

Всего за несколько минут высокоеффективный процесс Diskerase очистил диски всех машин нашего CDN, и машины больше не могли поддерживать соединения пользователей (или делать что-то еще полезное). Мы все еще могли обслуживать пользователей с помощью наших собственных data-центров, и спустя несколько минут единственным заметным эффектом было лишь небольшое увеличение задержки отклика. Насколько мы могли судить, лишь малое количество пользователей заметили, что есть проблема, и все

благодаря хорошему планированию производительности (хоть это мы сделали правильно!). Тем временем мы потратили почти два дня на переустановку машин в стойках, затронутых сбоем. Затем мы две недели проводили аудит и добавляли в код больше проверок работоспособности нашего процесса автоматизации, включая ограничение скорости, и сделали рабочий процесс по списанию оборудования идемпотентным.

Рекомендации

Ознакомившись с примерами, приведенными в этой главе, вы можете подумать, что перед тем, как внедрять автоматизацию, вашей компании нужно достичь масштабов Google. Это неверно: автоматизация даст вам нечто большее, нежели просто сэкономленное время, поэтому ее стоит реализовывать не только ради этого. Однако экономия времени принесет максимальную пользу на этапе проектирования: передача данных и оперативное взаимодействие позволяют вам быстрее реализовать функциональность, что сложно сделать для уже устоявшейся системы.

Автономные операции трудно качественно внедрить в достаточно крупных системах, но стандартные приемы разработки ПО — наличие несвязанных систем, введение API, минимизация побочных эффектов и т.д. — существенно помогут вам в этом.

[38](#) Читатели, уже понимающие ценность автоматизации, могут перейти к следующему разделу «Польза автоматизации для Google SRE». Однако обратите внимание, что в текущем разделе мы описываем кое-какие детали, которые неплохо бы знать при прочтении остальной части этой главы.

[39](#) Опыт, приобретаемый при построении таких автоматизированных систем, ценен сам по себе. Инженеры хорошо разбираются в процессах, которые они автоматизировали, и в будущем смогут быстрее автоматизировать новые процессы.

[40](#) Обратите внимание на следующий комикс XCKD: <http://xkcd.com/1205/>.

[41](#) Для примера прочтите следующую статью: <http://blog.engineyard.com/2014/pets-vs-cattle>.

[42](#) Конечно, не каждая система из тех, что нужно автоматизировать, предоставляет свой API, что заставляет нас использовать особые инструменты, например вызовы CLI или программно сгенерированные события веб-форм.

[43](#) Демоны — разновидность фоновых программ в Unix-системах. Аналогичное место в Windows занимают системные сервисы (службы). — *Примеч. пер.*

[44](#) Идемпотентная операция — действие, многократное повторение которого эквивалентно однократному.

[45](#) Для упрощения мы приводим ее в сокращенном виде.

[46](#) И при этом неизменные.

[47](#) См., например, https://en.wikipedia.org/wiki/Air_France_Flight_447.

[48](#) См., например, [Bainbridge, 1983] и [Sarter et al., 1997].

[49](#) Еще одна хорошая причина проводить регулярные тренировки; см. подраздел «Катастрофа: ролевая игра» раздела «Пять приемов для вдохновления дежурных работников» главы 28.

[50](#) Имеется в виду запись неинформативных данных поверх прежнего содержимого диска для затруднения его восстановления. — *Примеч. пер.*

8. Технологии выпуска ПО

Автор — Дина Макнамм.

Под редакцией Бетси Байер и Тима Харви

Управление выпуском новых версий, или релизами (release engineering), — это относительно новое и быстро растущее направление в индустрии ПО, которое может быть кратко определено как разработка и доставка программного обеспечения [McNutt, 2014a]. Релиз-инженеры — высококвалифицированные специалисты: они разбираются в принципах управления исходным кодом, в работе компиляторов, инструментов автоматизации сборок, менеджеров пакетов и установщиков, а также знают языки конфигурирования. Они владеют информацией из многих областей, таких как разработка, управление конфигурацией, интеграция тестов, системное администрирование и поддержка пользователей.

Запуск надежных сервисов требует наличия надежных процессов выпуска ПО. SR-инженеры должны знать, что бинарные и конфигурационные файлы, которые они используют, созданы воспроизводимым, автоматизированным способом. Это важно для того, чтобы релизы были повторяемыми и не становились «уникальными снежинками». Изменения на любом этапе этого процесса должны быть намеренными, а не случайными. SR-инженеры заботятся об этом, начиная с момента написания исходного кода и заканчивая развертыванием продукта.

Управление релизами в Google — это особая задача. Релиз-инженеры работают с программными инженерами (software engineers, SWE), когда продукт создается, и с SR-инженерами для того, чтобы определить все шаги, необходимые для

выпуска ПО, начиная от способа хранения исходных текстов в репозитории до написания скриптов (правил) компиляции и сборки, методов выполнения тестирования, формирования пакетов и развертывания.

Роль релиз-инженера

Google — компания, ориентированная на данные, и управление релизами также придерживается этой линии. У нас есть инструменты, которые сообщают о показателях вроде количества времени, требуемого на то, чтобы изменение в коде было развернуто в промышленной среде (другими словами, быстрота выпуска), и статистики, показывающей, какие функции использовались для создания конфигурационных файлов сборки [Adams et al., 2015]. Большая часть этих инструментов спроектирована и разработана релиз-инженерами.

Релиз-инженеры формулируют для нас практические рекомендации по использованию наших инструментов, чтобы убедиться, что проекты выпускаются с учетом соответствующих и проверенных методологий. Наши рекомендации охватывают все этапы процесса выпуска ПО. В них обозначены, например, флаги компилятора, форматы тегов идентификации сборки и необходимые шаги для выполнения сборки. Уверенность в том, что наши инструменты априори корректны и при этом качественно задокументированы, позволяет командам сосредотачиваться на функциональности и пользователях вместо того, чтобы тратить время на повторное изобретение колеса (причем посредственного качества), когда речь заходит о выпуске ПО.

В компании Google работает много SR-инженеров, которые отвечают за безопасное развертывание продукта и

поддержание сервисов Google в работоспособном состоянии. Для того чтобы убедиться, что наши процессы релиза соответствуют бизнес-требованиям, релиз-инженеры и SR-инженеры работают рука об руку, разрабатывая стратегии для тестирования изменений, выпуская новые версии ПО без остановки работы сервисов и откатывая функциональность, которая вызывает проблемы.

Основные положения

В управлении релизами следует придерживаться четырех основных принципов. Рассмотрим их.

Модель самообслуживания

Для того чтобы обеспечить масштабирование, команды специалистов должны быть самодостаточны. В рамках управления релизами были разработаны приемы и инструменты, которые позволяют нашим командам контролировать процессы релизов и запускать собственные процессы. Несмотря на наличие тысяч инженеров и продуктов, мы можем достичь высокой скорости выпуска, поскольку отдельные команды способны сами решать, как часто и когда выпускать новые версии своих продуктов. Процессы релиза могут быть автоматизированы так, чтобы участие инженеров в них было минимальным. Многие проекты автоматически создаются и выпускаются благодаря взаимодействию автоматизированной системы сборки проектов и наших инструментов для развертывания. Выпуск ПО действительно автоматизирован и требует участия инженера только в случае появления какой-то проблемы.

Высокая скорость

Пользовательское ПО (например, многие компоненты системы Google Search) часто модернизируется, поскольку мы ставим перед собой цель развертывать предназначенную для клиентов функциональность настолько быстро, насколько это возможно. В рамках нашего подхода частые релизы приводят к меньшему количеству изменений между версиями. Это упрощает тестирование и поиск ошибок. Некоторые команды выполняют сборку каждый час, а затем выбирают одну из готовых версий для развертывания в промышленной среде. Выбор делается на основе результатов тестирования и функциональности, содержащейся в каждой сборке. Другие команды пользуются моделью релизов Push on Green и развертывают каждую сборку, выполняя для нее все тесты [Klein, 2014].

«Герметичные» сборки

Инструменты для создания сборок позволяют нам гарантировать стабильность и возможность повторения сборки. Если два человека попробуют собрать один и тот же продукт с одним и тем же номером «ревизии» в репозитории исходного кода на разных компьютерах, мы ожидаем, что результат будет одинаковым⁵¹. Наши сборки самодостаточны — «герметичны». Это значит, что они нечувствительны к библиотекам и прочему программному обеспечению, установленному на компьютере, на котором выполняется сборка. Вместо этого сборки зависят от существующих версий инструментов, таких как компиляторы, и зависимостей, например библиотек. Процесс сборки автономен и не должен полагаться на сервисы, которые являются внешними по отношению к среде сборки.

Повторная сборка более старых релизов, когда нам нужно исправить ошибку в ПО, уже работающем как промышленное, может оказаться трудной задачей. Мы решаем ее путем сборки той же самой «ревизии», что была в оригинальной версии, включив туда появившиеся позже необходимые изменения. Мы называем этот процесс избирательной модификации «извлечением изюминок» (в оригиналe — вишенок, *cherry picking*. — Примеч. пер.). Нашим инструментам сборки также присваиваются номера версий, согласованные с «ревизиями» репозитория исходного кода проекта, для которого выполняется сборка. Таким образом, проект, который собирали в предыдущем месяце, и для «изюминок» не будет использовать версию компилятора текущего месяца, ведь эта версия может содержать несовместимую или нежелательную функциональность.

Обязательные политики и процедуры

Несколько уровней безопасности и контроля доступа определяют, кто может выполнять специфические операции при релизе проекта. К контролируемым операциям относятся:

- подтверждение изменений исходного кода — эта операция управляет с помощью конфигурационных файлов, распределенных по всей базе кода;
- указание действий, которые должны быть выполнены во время процесса релиза;
- создание нового релиза;
- подтверждение исходного «запроса на интеграцию» (то есть на выполнение сборки с заданным номером «ревизии» в

репозитории исходного кода) и последующие «извлечения изюминок»;

- развертывание нового релиза;
- внесение изменений в конфигурацию сборки проекта.

Практически все изменения базы кода требуют просмотра кода и подтверждения сделанных изменений (code review) — эта типовая стадия включена в обычный трудовой процесс разработчика. Наша автоматизированная система релизов формирует отчет обо всех изменениях в релизе, который поставляется вместе с прочими файлами, созданными в результате сборки. Этот отчет позволяет SR-инженерам понять, какие изменения включены в новую версию проекта, и может ускорить поиск проблем, если таковые появляются во время релиза.

Непрерывная сборка и развертывание

В компании Google была разработана автоматизированная система релизов *Rapid*. *Rapid* — это система, строящая на основе технологий Google фреймворк для создания масштабируемых, самодостаточных и надежных релизов. В следующих разделах описывается жизненный цикл ПО в Google, а также показывается, как им управляют с помощью *Rapid* и других инструментов.

Сборка

Для выполнения сборки в компании Google был выбран инструмент *Blaze*⁵². Он поддерживает получение исполняемых

файлов программ на множестве языков, включая стандартные языки вроде C++, Java, Python, Go и JavaScript. Инженеры используют Blaze для определения так называемых целей сборки (например, генерируемых JAR-файлов) и указания зависимостей для каждой цели. При выполнении сборки Blaze автоматически выполняет сборку и для зависимостей.

Цели сборки для исполняемых файлов и юнит-тестов определены в файлах конфигурации проекта. Специфические для проектов флаги (например, уникальный идентификатор сборки) Rapid передает Blaze. Все бинарные файлы поддерживают флаг, который отображает дату сборки, номер «ревизии» и идентификатор сборки, что позволяет нам легко связать бинарный файл и информацию о том, как он был собран.

Ветвление

Весь код регистрируется в главной ветви дерева исходного кода (mainline). Однако многие крупные проекты не выходят на релиз непосредственно из главной ветви. Вместо этого мы создаем новую ветвь для конкретной «ревизии» и никогда не объединяем код этой ветви с кодом главной. Изменения для исправления ошибок вносятся сначала в основную ветвь и затем как отдельные «изюминки» выбираются в заданную ветвь, чтобы попасть в релиз. Это позволяет избежать непреднамеренного объединения не связанных с текущим релизом изменений, отправленных в основную ветвь уже после того, как была выполнена первая сборка данной «ревизии». Такой подход позволяет нам точно знать содержимое каждого релиза.

Тестирование

Система непрерывного тестирования запускает модульные тесты для кода основной ветви каждый раз при внесении очередных изменений, что позволяет нам быстро обнаруживать дефекты. С точки зрения управления релизами рекомендуется использовать объекты и цели тестирования, согласованные с релизом проекта. Мы также рекомендуем создавать релизы под номером «ревизии» (версии) последней «непрерывной» тестовой сборки, успешно прошедшей все тесты. Эти меры снижают риск того, что последующие изменения, внесенные в основную ветвь, вызовут сбои во время выполнения сборки релиза.

Во время релиза мы повторно запускаем тесты модулей для «релизной» ветви проекта и создаем контрольный журнал, который показывает, что все тесты были выполнены. Этот этап важен, поскольку, если релиз содержит код после «выборки изюминок», то в «релизной» ветви может быть версия кода, которая больше не существует в основной ветви. Мы хотим гарантировать, что тесты относятся к тому, что на самом деле выпускается.

Чтобы дополнить систему непрерывного тестирования, мы используем для него независимую среду, которая запускает тесты на уровне системы для пакетов, созданных во время сборки. Эти тесты могут быть запущены вручную или с помощью Rapid.

Пакеты

Программное обеспечение распространяется между компьютерами, составляющими нашу «промышленную среду», с помощью менеджера пакетов Midas (Midas Package Manager, MPM) [McNutt, 2014c]. Он собирает пакеты, основываясь на правилах Blaze, где перечислены созданные во время сборки файлы, которые следует включить в пакет, а также их

владельцы и разрешения. Пакеты имеют имена (например, `search/shakespeare/frontend`), для них указывается версия, уникальный хеш-код и подпись для гарантии аутентичности. МРМ поддерживает метки для отдельных версий пакета. Rapid прикрепляет метку с идентификатором сборки — это гарантирует, что для пакета будет создана уникальная ссылка, которая будет представлять собой имя пакета и эту метку.

Метки можно прикрепить к пакету МРМ, чтобы указать его местоположение в процессе релиза (например, `dev`, `canary` или `production`). Если вы примените существующую метку к новому пакету, она будет автоматически перемещена от старого пакета к новому. Например, если пакет имеет метку `canary`, то человек, который впоследствии будет устанавливать `canary`-версию (версию, предназначенную для канареочного тестирования) этого пакета, автоматически получит самую свежую версию пакета с меткой `canary`.

Rapid

На рис. 8.1 показаны основные компоненты системы Rapid. Она сконфигурирована с помощью файлов, которые называются *макетами* (*blueprints*). Макеты написаны на внутреннем конфигурационном языке и используются для определения целей сборки и тестирования, правил развертывания и административной информации (вроде данных о владельце проекта). Списки управления, основанные на ролях, определяют, кто может выполнять конкретные действия для проектов в Rapid.

Каждый проект Rapid имеет рабочие потоки, определяющие действия, которые нужно выполнить во время процесса релиза. Действия рабочих потоков могут быть выполнены

последовательно или параллельно, один рабочий поток может запускать другие рабочие потоки. Rapid отправляет запросы на выполнение задач, которые запущены как задания Borg на наших «промышленных» серверах. Поскольку Rapid использует нашу производственную инфраструктуру, становится возможным обрабатывать тысячи запросов на выполнение релизов одновременно.

Обычный процесс релиза выполняется так.

1. Rapid использует запрошенный интеграционный номер «ревизии» (зачастую получаемый автоматически от системы непрерывного тестирования) для создания «релизной» ветки.

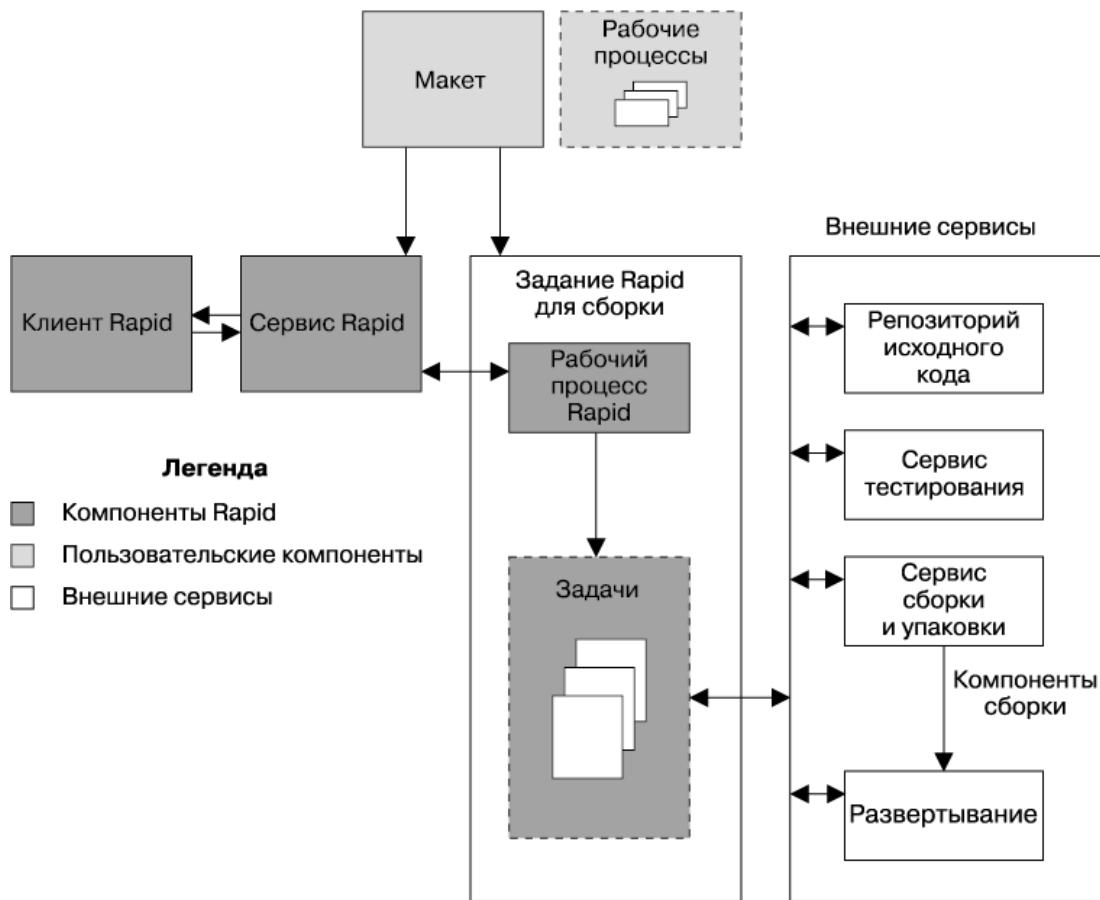


Рис. 8.1. Упрощенный вид архитектуры Rapid, демонстрирующий основные компоненты системы

2. Rapid использует Blaze для компиляции всех исполняемых файлов и выполнения всех модульных тестов, зачастую выполняя оба этих шага параллельно. Компиляция и тестирование происходят в специализированных окружениях, в отличие от задания Borg, где выполняется рабочий поток Rapid. Такое разделение позволяет нам легко распараллелить работу.
3. Файлы, созданные во время сборки, становятся доступными для тестирования и пробного развертывания — canary-релиза. Обычное развертывание canary-версии включает в себя запуск нескольких задач в режиме промышленной эксплуатации после прохождения всех системных тестов.
4. Результаты каждого этапа процесса заносятся в журнал. Создается отчет обо всех изменениях, произведенных с момента создания последнего релиза.

Rapid позволяет нам управлять «релизными» ветвями и избирательной модификацией кода («изюминками»). Отдельные запросы на модификацию кода могут быть одобрены или отклонены при добавлении в релиз.

Развертывание

Для проведения простых развертываний зачастую применяется непосредственно система Rapid. Она обновляет задания Borg для того, чтобы они использовали только что собранные пакеты MPM, основываясь на настройках развертывания, заданных в файлах макетов и специализированных управляющих программах.

В более сложных случаях мы используем *Sisyphus* — универсальный фреймворк для автоматизации, разработанный SR-инженерами. Выпуск финальной версии — это логическая единица работы, которая состоит из одной или нескольких отдельных задач. *Sisyphus* предоставляет набор классов Python, который может быть расширен для поддержки любого процесса развертывания и установки. В нем предусмотрена информационная панель, позволяющая более удобно управлять проведением релиза и наблюдать за его ходом.

Как правило, при выполнении интеграции *Rapid* создает версию для установки в рамках одной крупной, длительной задачи *Sisyphus*. *Rapid* знает метку сборки, связанную с созданным пакетом МРМ, и может передать ее *Sisyphus*. *Sisyphus* использует метку сборки для того, чтобы указать, какая версия пакетов МРМ должна быть развернута.

С помощью *Sisyphus* процесс выпуска версии может быть настолько простым или сложным, насколько это необходимо. Например, фреймворк может обновлять все связанные задачи немедленно или же откладывать выпуск нового исполняемого файла для каждого следующего кластера на несколько часов.

Наша цель заключается в том, чтобы вместить процесс развертывания в профиль риска заданного сервиса. В тестовом или предпромышленном окружении мы можем выполнять сборку новых версий каждый час и отправлять их на установку автоматически, как только они пройдут все тесты. Для крупных пользовательских сервисов мы можем начинать развертывание с одного кластера и распространять его экспоненциально, пока не будут охвачены все кластеры. Для критических элементов инфраструктуры мы можем продлить период развертывания до нескольких дней, обеспечив чередование между экземплярами в разных географических регионах.

Управление конфигурацией

Управление конфигурацией — это область, в которой релиз-инженеры и SR-инженеры взаимодействуют наиболее тесно. Несмотря на то что управление конфигурацией на первый взгляд может показаться обманчиво простой задачей, изменения конфигурации — это потенциальный источник нестабильности. В результате наш подход к созданию релизов и управлению системой, а также ее конфигурацией со временем значительно эволюционировал. Сегодня мы можем применять несколько моделей для распространения файлов конфигурации, как это описано далее. Все схемы включают в себя хранение конфигурации в основном репозитории исходного кода и требуют строгого предпросмотра кода.

- *Использование для конфигураций главной ветви репозитория.* Этот метод был первым из тех, что мы использовали для конфигурирования сервисов в Borg (и систем, которые ей предшествовали). Придерживаясь такой схемы, разработчики и SR-инженеры просто модифицировали конфигурационные файлы в верхней «ревизии» основной ветви. Изменения анализировались и применялись к запущенной системе. В результате релизы исполняемых файлов и изменения в конфигурациях не были связаны друг с другом. Несмотря на простоту идеи и ее реализации, такой прием зачастую приводил к рассогласованию между проверенной (зарегистрированной) и запущенной версиями конфигурационных файлов, поскольку задачи должны обновляться перед тем, как к ним будут применены какие-то изменения.
- *Включение конфигурационных и исполняемых файлов в один MPM-пакет.* Для проектов с небольшим количеством конфигурационных файлов или проектов, где эти файлы

(или подмножество файлов) изменяются с каждым циклом релиза, конфигурационные файлы могут быть включены в пакет MPM вместе с исполняемыми. Хотя такая стратегия несколько ограничивает нам гибкость, слишком тесно связывая бинарные и конфигурационные файлы, она упрощает развертывание, поскольку потребуется установка только одного пакета.

- *Сборка файлов конфигураций в конфигурационные пакеты MPM.* Мы можем применить принцип герметичности и к управлению конфигурацией. Конфигурация, как правило, тесно связана с определенной версией исполняемых файлов, поэтому мы пользуемся системами сборки и упаковки для того, чтобы сделать снимок состояния и выпустить конфигурационные файлы вместе с соответствующими исполняемыми. Так же как и для исполняемых файлов, мы можем использовать идентификатор сборки, чтобы воспроизвести конфигурацию в любой заданный момент времени.

Например, изменение, в котором реализуется новая функциональность, может быть выпущено с конфигурирующим ее флагом. Создавая два пакета MPM, один для бинарного файла, а второй — для конфигурационного, мы сохраняем возможность изменять каждый пакет независимо от другого. Иными словами, если функциональность была выпущена с установленным флагом `first_folio`, но мы обнаружили, что это должен быть флаг `bad_quarto`, мы можем включить это изменение в «релизную» ветвь, повторно собрать конфигурационный пакет и развернуть его. Преимущество такого подхода

заключается в том, что он не требует повторной сборки исполняемого файла.

Мы можем воспользоваться метками для МРМ-пакетов, чтобы указать, какие версии этих пакетов следует устанавливать вместе. Метка `much_ado` может быть применена к МРМ-пакетам, описанным в предыдущем абзаце, что позволит нам выбрать оба пакета. При сборке новой версии проекта метка `much_ado` будет применена к новым пакетам. Поскольку эти теги уникальны в рамках пространства имен пакета МРМ, будет использован только самый последний пакет с этим тегом.

- *Считывание конфигурационных файлов из внешнего хранилища.* Отдельные проекты имеют конфигурационные файлы, которые нужно изменять часто или динамически (например, при работающем исполняемом файле). Эти файлы могут храниться в Chubby, Bigtable или нашей файловой системе [Kemper, 2011].

В итоге владельцы проекта рассматривают разные варианты распространения конфигурационных файлов и управления ими и решают, какой из этих подходов подойдет лучше в каждом конкретном случае.

Итоги главы

Несмотря на то что в этой главе рассматривается подход компании Google к управлению релизами, а также способы работы релиз-инженеров и их взаимодействия с SR-инженерами, эти приемы могут быть использованы и в других областях.

Это работает не только для Google

При наличии правильных инструментов, соответствующей автоматизации и качественно настроенных политиках разработчики и SR-инженеры не должны волноваться о создании релизов. Релиз должен создаваться простым нажатием кнопки.

Большинство компаний пытаются решить те же инженерные задачи, пусть и в различных масштабах и с использованием различных инструментов. Как следует управлять версиями для своих пакетов? Следует ли вам применять модель непрерывной сборки и развертывания или же лучше выполнять периодические сборки? Как часто нужно делать релизы? Какие правила по управлению конфигурацией вам необходимо использовать? Какие показатели вас интересуют?

Релиз-инженеры компании Google создали собственные инструменты, поскольку ни один доступный инструмент не мог работать в необходимых нам масштабах. Такие инструменты позволили нам включить функциональность для поддержки (и даже принудительного внедрения) правил процесса релиза. Однако эти правила сначала нужно определить, чтобы добавить соответствующую функциональность нашим инструментам. Все компании должны определить этапы процесса выполнения релиза независимо от того, могут ли эти процессы быть автоматизированы и/или сделаны обязательными для исполнения.

Управляйте релизами с самого начала

Об управлении релизами зачастую вспоминают лишь в самом конце, и такой подход необходимо менять по мере роста

масштабов и сложности платформы и сервисов.

Команды должны спланировать бюджет ресурсов для управления релизами в самом начале цикла разработки продукта. Использовать правильные методики с самого начала работ гораздо дешевле, нежели внедрять их в систему позднее.

Критически важно, чтобы разработчики, SR-инженеры и релиз-инженеры работали вместе. Релиз-инженеры должны понять, как код должен быть собран и развернут. Разработчики не должны ограничиваться лишь сборкой проекта, оставляя результат на откуп релиз-инженерам.

Команды конкретных проектов решают, когда следует начать управление релизом для своего проекта. Поскольку управление выпуском ПО — это относительно новая дисциплина, менеджеры не всегда планируют и закладывают в бюджет эти затраты на ранних стадиях проекта. Поэтому при определении того, как вы будете внедрять приемы управления выпуском ПО, убедитесь, что оценили его роль в применении ко всему жизненному циклу вашего продукта или сервиса — в том числе и на ранних этапах.

Дополнительная информация

Более подробную информацию о технологиях выпуска ПО вы можете получить в следующих презентациях, для каждой из которых доступны видеозаписи:

How Embracing Continuous Release Reduced Change Complexity
[\(<http://usenix.org/conference/ures14west/summit-program/presentation/dickson>\)](http://usenix.org/conference/ures14west/summit-program/presentation/dickson), USENIX Release Engineering Summit West 2014, [Dickson, 2014];

Maintaining Consistency in a Massively Parallel Environment
(<https://www.usenix.org/conference/ucms13/summit-program/presentation/mcnutt>), USENIX Configuration Management Summit 2013, [McNutt, 2013];

The 10 Commandments of Release Engineering
(https://www.youtube.com/watch?v=RNMrjYV_UsQ8), 2nd International Workshop on Release Engineering 2014, [McNutt, 2014b];

Distributing Software in a Massively Parallel Environment
(<https://www.usenix.org/conference/lisa14/conference-program/presentation/mcnutt>), LISA 2014, [McNutt, 2014c].

[51](#) Компания Google использует единый унифицированный репозиторий для всего исходного кода; см. [Potvin, Levenberg, 2016].

[52](#) Исходный код Blaze был размещен под названием Bazel. Взгляните на Bazel FAQ по адресу <http://bazel.io/faq.html>.

9. Простота

Автор — Макс Люббе

Под редакцией Тима Харви

Цена надежности — гонка за предельной простотой.

Ч-А. Хоар, лекция на премии Тьюринга

Программное обеспечение по своей природе динамично и нестабильно [53](#). Оно может быть полностью стабильно только в том случае, если находится в вакууме. Если мы перестанем изменять исходный код, мы перестанем создавать ошибки. Если аппаратная часть или библиотеки никогда не изменятся, ни один компонент не будет вызывать ошибки. Если мы заморозим текущую базу пользователей, нам никогда не придется масштабировать систему. Фактически описать подход SR-инженеров к управлению системами можно такой фразой: «В конечном счете наша задача заключается в том, чтобы поддерживать баланс между гибкостью и стабильностью»[54](#).

Стабильность или гибкость?

Иногда имеет смысл пожертвовать стабильностью ради гибкости. Я часто подходил к решению незнакомой проблемы, начиная работу с так называемого разведочного программирования. Я устанавливал «срок годности» для любого своего кода, понимая, что нужно будет сделать немало ошибок, прежде чем я пойму, что именно требуется сделать. Код с явно определенным «сроком годности» может быть гораздо более удобным в работе с точки зрения тестов и управления релизами, поскольку он никогда не будет

отправлен в промышленную эксплуатацию и пользователи его не увидят.

Для большинства ПО, находящегося в промышленной эксплуатации, важно гармонично сочетать стабильность и гибкость. SR-инженеры стараются создавать процедуры, приемы и инструменты, которые делают ПО более надежным. В то же время они гарантируют, что их работа будет минимально влиять на гибкость разработчиков. Действительно, на своем опыте SR-инженеры убедились, что надежные процессы, как правило, увеличивают гибкость для разработчиков: быстрые и надежные релизы позволяют легко заметить изменения. В результате, как только появляется ошибка, для ее поиска и исправления потребуется немного времени. Гарантия надежности позволяет разработчикам сосредоточиться на том, что действительно важно для них, — на функциональности и производительности их ПО и систем.

«Скучность» как добродетель

Когда речь идет о программном обеспечении, его «скучность» является достоинством. Нам не столько важно, чтобы наши программы были спонтанными и интересными, сколько важно, чтобы они работали по сценарию и предсказуемо решали свои бизнес-задачи. Как сказал инженер компании Google Роберт Мут: «В отличие от детективной истории, желательно, чтобы исходный код не давал повода для волнения, беспокойства и загадок». Сюрпризы на производстве — злейшие враги SR-инженеров.

Как предполагает в своем эссе *No Silver Bullet* Фред Брукс [Brooks, 1995], очень важно понимать разницу между сложностью органичной (естественной) и неорганичной (случайной). Органичная сложность — это сложность,

свойственная заданной ситуации, ее нельзя избежать по определению, а неорганичная сложность более гибкая, от нее можно избавиться, приложив некоторые усилия. Например, при создании веб-сервера необходимо учитывать органичную сложность задачи быстрого формирования веб-страниц. Однако если мы пишем код веб-сервера на языке Java, то можем создать неорганичную сложность, попытавшись минимизировать влияние сборки мусора на производительность.

Нацелившись на минимизацию неорганичной сложности, команды SR-инженеров должны делать следующее:

- проводить тестирование при появлении неорганичной сложности в системах, за которые они ответственны;
- постоянно стремиться избавляться от сложности в системах, с которыми они работают.

Не отдам свой код!

Поскольку инженеры — обычные люди, они могут эмоционально привязываться к своим творениям, и поэтому нередки конфликты из-за того, что были удалены крупные фрагменты кода. Некоторые могут протестовать: «Что, если этот код понадобится нам в будущем?», «Почему бы нам просто не закомментировать код, чтобы мы могли вновь добавить его позднее?» или «Почему бы нам не пропустить код, пометив его флагом, вместо того чтобы удалять?». Все эти возражения безосновательны. Системы контроля версий позволяют легко откатить изменения, а сотни строк закомментированного кода лишь отвлекают и сбивают с толку разработчиков (особенно по мере увеличения файлов исходного текстов). Код, который никогда не выполняется, помечен флагом и всегда отключен,

похож скорее на бомбу замедленного действия, что на своем печальном опыте прочувствовала, например, компания Knight Capital (см. *Order In the Matter of Knight Capital Americas LLC [Securities..., 2013]*).

Не хочу утрировать, но, когда вы задумываетесь о создании веб-сервиса, который должен быть доступен в режиме 24/7, в какой-то мере вы несете ответственность за каждую новую строку кода. SR-инженеры должны использовать приемы, гарантирующие, что весь код следует исходной цели. К числу таких приемов можно отнести, например, следующее:

- тщательное изучение кода для того, чтобы убедиться, что он на самом деле позволяет достичь бизнес-целей;
- регулярное удаление «мертвого» кода;
- внедрение методов, позволяющих обнаружить чрезмерное увеличение объема кода («разбухание») на всех уровнях тестирования.

Показатель «Отрицательные строки кода»

Термин «разбухание ПО» был введен для описания тенденции, в рамках которой с течением времени программное обеспечение увеличивается в объеме и замедляется из-за постоянного добавления дополнительной функциональности. Хотя «разбухшее» ПО кажется нежелательным даже интуитивно, его недостатки становятся особенно заметны, если посмотреть на них с точки зрения SR-инженера. Каждая новая или измененная строка кода проекта создает угрозу появления новых дефектов и ошибок. Небольшой проект проще понять, протестировать, и в нем зачастую меньше дефектов. Учитывая такую точку зрения, мы должны

дополнительно проверять код, когда нам нужно срочно добавить в проект новую функциональность. Я был свидетелем того, как удалялись тысячи строк кода, утратившего актуальность.

Минимальные API

Французский писатель Антуан де Сент-Экзюпери написал: «... совершенство достигается не тогда, когда уже нечего прибавить, но когда уже ничего нельзя отнять» [Saint Exupery, 1939]. Этот же принцип применим и к разработке ПО. API — особенно яркий пример того, почему нужно следовать этому правилу.

Написание понятных, четких API — это существенный аспект в достижении простоты программного обеспечения. Чем меньше методов и аргументов мы предоставляем пользователям API, тем проще будет понять этот API и тем больше усилий мы сможем приложить для совершенствования данных методов. Опять же сознательный отказ устранять некоторые проблемы позволит нам сосредоточиться на основной задаче и улучшить решения, выполнение которых мы явно откладывали. В области ПО «меньше» значит «больше»! Небольшой простой API также является признаком хорошо проработанного проекта.

Модульность

По мере увеличения масштабов и ухода от API и отдельных самодостаточных исполняемых файлов многие правила, действующие в объектно-ориентированном программировании, можно применить и к проектированию распределенных систем. Способность вносить изменения в

изолированные системы очень важна при создании качественных проектов. Например, отсутствие жестких связей между исполняемыми файлами и файлами конфигурации повышает одновременно и гибкость для разработчика, и стабильность системы. Если ошибка обнаружена в программе, которая является компонентом более крупной системы, ее можно исправить и внедрить решение в эксплуатацию независимо от остальной части системы.

Хотя с первого взгляда модульность, предлагаемая API, может показаться слишком простой, не совсем очевидно, что идея модульности распространяется и на способ внедрения изменений. Всего одно изменение API может заставить разработчиков полностью перестроить их систему и столкнуться с риском внесения новых ошибок. Управление версиями API позволяет разработчикам продолжить использовать ту версию, от которой зависит их система, а затем перейти на новую версию гораздо более безопасным и продуманным способом. Частота релизов может варьироваться в разных частях системы, и не обязательно заново разворачивать всю систему полностью каждый раз, когда добавляется новая функциональность или изменяется имеющаяся.

По мере увеличения сложности системы разделение обязанностей между API и исполняемыми файлами становится все более важным. Это прямая аналогия с проектированием классов в объектно-ориентированном программировании. Вы прекрасно понимаете, что создание класса-«солянки», который содержит несвязанные функции, — плохой прием. Так вот, плохим приемом считается также и создание и передача в промышленную эксплуатацию исполняемого файла util или misc. Хорошо спроектированная распределенная система

содержит взаимодействующие объекты, каждый из которых имеет четкую цель.

Концепция модульности применима и к формату данных. Так, одной из определяющих особенностей и целей проектирования «протокольных буферов» Google⁵⁵ было создание формата для взаимодействия, обладавшего прямой и обратной совместимостью.

Простота релизов

Простые релизы зачастую лучше сложных. Гораздо проще измерить и понять влияние одного изменения, а не группы изменений, выпущенных одновременно. Если мы в один момент выпустим 100 не связанных друг с другом изменений и при этом снизится производительность, то для того, чтобы понять, какое именно изменение и как повлияло на производительность, потребуются значительные усилия и дополнительный инструментарий. Если выполняются небольшие релизы, мы можем двигаться быстрее и более уверенно, поскольку каждое изменение кода в крупной системе можно рассмотреть по отдельности. Такой подход к релизам можно сравнить с градиентным спуском в машинном обучении, когда оптимальнее продвигаться небольшими шагами, рассматривая каждое изменение.

Итоги главы

В этой главе раз за разом повторяется одна идея: простота ПО — необходимое условие надежности. Наши попытки упростить каждый шаг конкретной задачи — это не лень. Вместо этого мы проясняем, чего конкретно хотим достичь и какой способ окажется наиболее простым. Каждый раз, когда мы убираем

какую-то функциональность, мы не препятствуем нововведениям. Мы поддерживаем порядок в среде, чтобы сконцентрироваться непосредственно на инновациях и иметь возможность выполнять реальную инженерную работу.

[53](#) Зачастую это верно для сложных систем вообще; см. [Perrow, 1999] и [Cook, 2000].

[54](#) Фраза придумана моим бывшим менеджером Йоханом Андерсоном примерно в то время, когда я стал SR-инженером.

[55](#) Протокол сериализации, называемый также протокольными буферами (protobuffs), — это независимый от языка и платформы расширяемый механизм для сериализации структурированных данных. Для получения более подробной информации см. <https://developers.google.com/protocol-buffers/docs/overview#a-bit-of-history>.

Часть III. Практики

Говоря простым языком, SR-инженеры запускают сервисы, которые представляют собой набор несвязанных систем, и отвечают за работоспособность этих систем. Успешное управление сервисом включает в себя различные действия: разработку систем мониторинга, планирование пропускной способности, реагирование на критические ситуации, обеспечение принятия мер по устранению основных причин сбоев, и т.д. В этой части мы рассмотрим теорию и практику повседневных задач SR-инженеров: основы построения и эксплуатации крупных распределенных вычислительных систем.

Мы можем описать работоспособность системы — примерно так же, как Абрахам Маслоу описал человеческие потребности [Maslow, 1943] — от наиболее простых действий, необходимых для работы системы, до высших уровней функционирования, включая возможность самосовершенствования системы и управление развитием сервиса вместо точечного реагирования на проблемы. Мы лишь недавно осознали всю важность этого описания для формирования подходов к оценке сервиса в Google. Это произошло, когда несколько наших SR-инженеров, включая бывшего коллегу Майки Диккерсона⁵⁶, вынуждены были временно присоединиться к ИТ-команде правительства США для того, чтобы помочь им запустить сайт **healthcare.gov** в конце 2013 — начале 2014 года. В тот момент им нужен был способ объяснить, как повысить надежность системы.

Мы используем иерархию, показанную на рис. III.1, чтобы взглянуть на элементы, которые позволяют сделать сервис надежным: от наиболее простых до наиболее сложных.

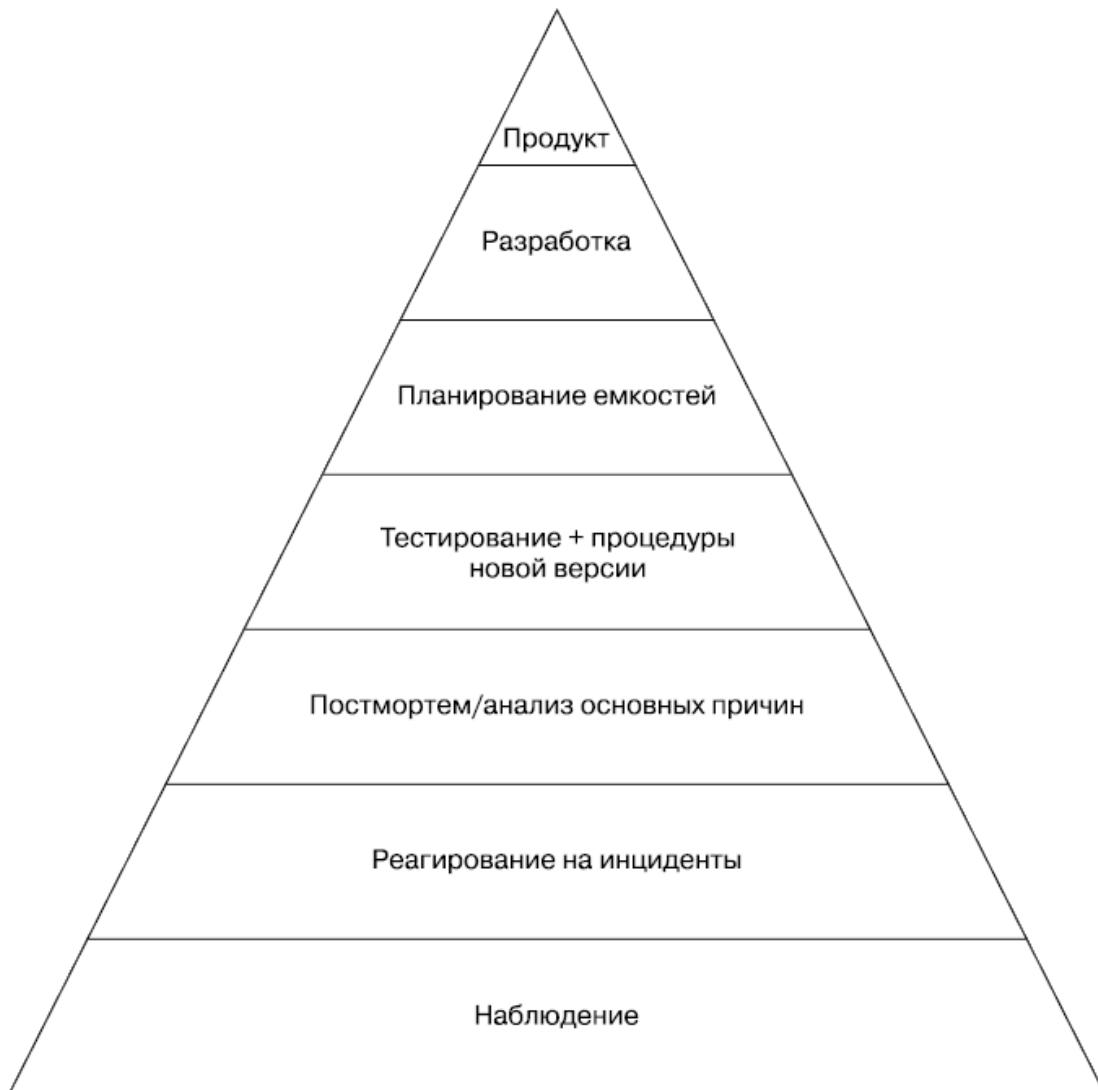


Рис. III.1. Иерархия надежности сервиса

Мониторинг

Без мониторинга вы не сможете сказать, работает ли сервис. Отсутствие продуманной инфраструктуры мониторинга приведет к тому, что вы будете работать вслепую. Может, посетители сайта увидят ошибку, а может, и нет, но вы должны знать о проблемах до того, как их заметят ваши пользователи.

Инструменты и принципы мониторинга мы рассмотрим в главе 10 «Оповещения на основании данных временных рядов».

Реагирование в критических ситуациях

SR-инженеры не просто так ходят на дежурства. Поддержка дежурными «на связи» — это средство для осуществления наших главных задач и для того, чтобы знать, как именно работают (и как не работают!) распределенные вычислительные системы. Если мы найдем способ освободить себя от необходимости носить пейджер, то обязательно сделаем это. В главе 11 «Быть на связи» мы объясняем, как балансируем между дежурными обязанностями и другими служебными делами.

Узнав, что появилась проблема, как вы избавитесь от нее? Не обязательно сразу исправлять ее раз и навсегда — возможно, вы сумеете лишь «остановить кровотечение», временно отключив некоторую функциональность, или направить трафик на другой экземпляр сервиса, работающий корректно. Особенности решения, которое вы захотите реализовать, будут характерны для вашего сервиса и организации. Но эффективное реагирование на критические ситуации актуально для всех команд.

В первую очередь важно определить, что пошло не так. Соответствующий структурный подход мы описываем в главе 12 «Эффективная диагностика и решение проблем».

Во время аварии зачастую хочется поддаться панике и начать реагировать по ситуации. В главе 13 «Реагирование в критических ситуациях» мы советуем не поддаваться этому искушению, а в главе 14 «Управление в критических ситуациях» рассказываем, как эффективное управление в

критических ситуациях смягчает их последствия и позволяет избавиться от тревожности в ожидании сбоев.

Постмортем и анализ основных причин

Мы стремимся настроить систему так, чтобы получать оповещения по поводу только новых и действительно существенных проблем наших сервисов. Решать раз за разом одну и ту же проблему может быть ужасно скучно. Фактически такой подход — один из ключевых факторов, отличающих философию SRE. Эта тема рассматривается в двух главах.

Формирование культуры отчетов о проблемах (постмортемов), не предусматривающей поиска виновных, — это первый шаг к пониманию того, что пошло не так (и что пошло по плану!). Об этом — глава 15 «Культура постмортема: учимся на ошибках».

С этой темой связана также глава 16 «Контроль неисправностей», где мы кратко описываем инструменты и средства, которые позволяют командам SRE отслеживать недавние аварии в «промышленных» системах, анализировать их причины и предпринятые действия.

Тестирование

Как только мы поняли, что именно идет не так, следующим шагом должна быть попытка предотвратить инцидент, поскольку предотвратить проблему гораздо важнее, чем исправить ее. Наборы тестов дают некоторые гарантии того, что наше ПО не генерирует определенные ошибки перед отправкой в эксплуатацию. О том, как их лучше использовать, мы поговорим в главе 17 «Тестирование надежности систем».

Планирование пропускной способности

В главе 18 «Разработка ПО службой SRE» мы предлагаем рассмотреть в качестве примера разработку Auxon — инструмента для автоматизации планирования пропускной способности.

За планированием пропускной способности логично следует балансировка нагрузки, которая позволяет гарантировать, что мы сможем соответствующим образом воспользоваться установленными мощностями. В главе 19 «Балансировка нагрузки на уровне фронтенда» мы рассмотрим, как запросы к нашим сервисам отправляются в data-центры. Далее мы продолжим это обсуждение в главе 20 «Балансировка нагрузки в data-центре» и главе 21 «Справляемся с перегрузками» — каждая из этих тем очень важна для гарантирования надежности сервиса.

Наконец, в главе 22 «Справляемся с каскадными сбоями» мы дадим советы о том, как справиться с каскадными сбоями при проектировании системы, а также в тех случаях, когда каскадный сбой уже произошел.

Разработка

Одними из основных особенностей подхода компании Google к обеспечению надежности сайтов являются проектирование крупномасштабных систем и инженерная работа внутри организации.

В главе 23 «Разрешение конфликтов: консенсус в распределенных системах и обеспечение надежности» мы объясняем понятие консенсуса (согласованности), которое (в виде алгоритма Paxos) лежит в основе многих распределенных систем компании Google, включая нашу глобальную распределенную систему cron. В главе 24 «Cron:

планирование и расписание в распределенных системах» мы в общих чертах описываем систему, которая масштабируется на все дата-центры и даже за их пределы, что совсем не просто.

В главе 25 «Конвейеры обработки данных» рассматривается, какие формы могут принимать конвейеры по обработке данных: от периодически запускаемых одиночных задач MapReduce до систем, которые могут работать практически в режиме реального времени. Использование разных архитектур приводит к неожиданным проблемам, решать которые приходится постоянно.

Гарантия того, что сохраненные вами данные будут на месте в тот момент, когда вы захотите их прочитать, — основной смысл целостности данных. В главе 26 «Сохранность данных: как пишется, так и читается» мы объясняем, как безопасно хранить данные.

Продукт

Наконец, добравшись до самой верхушки пирамиды надежности, мы обнаруживаем, что имеем рабочий продукт. В главе 27 «Надежный масштабируемый выпуск продукта» мы рассказываем о том, как компания Google выпускает свои продукты, стремясь, чтобы пользователи были довольны с первого дня работы сервиса.

Другие источники от Google SRE

Как мы говорили ранее, процесс тестирования хрупок и ошибки в нем могут сильно повлиять на общую стабильность. В статье ACM [Krishan, 2012] мы объясняем, как в Google выполняется тестирование устойчивости в масштабах

компании. Оно позволяет гарантировать, что мы сумеем справиться с непредвиденными ситуациями.

Несмотря на то что планирование производительности кажется темным искусством, полным таинственных таблиц, оно критически важно. Как показывает [Hixson, 2015a], для этого вам *не потребуется* хрустальный шар.

Наконец, новый интересный подход к безопасности корпоративной сети подробно рассматривается в [Ward, 2014]. Вы узнаете об инициативе перехода от сетей интранет с управлением доступом на основе привилегий к использованию идентификационных данных для устройств и пользователей. Такой подход, управляемый SR-инженерами на уровне инфраструктуры, определенно стоит учитывать при создании вашей следующей сети.

[56](#) Майки покинул компанию Google летом 2014 года, чтобы стать первым администратором US Digital Service (<https://www.whitehouse.gov/digital/united-states-digital-service>).

10. Оповещения на основании данных временных рядов

Автор — Джейми Уилкинсон

Под редакцией Кавиты Джулиани

Пусть запросы идут, а ваш пейджер молчит.

Традиционное благословение SR-инженеров

Мониторинг, находящийся на нижнем уровне иерархии потребностей производства, критически важен для стабильной работы вашего сервиса. Благодаря мониторингу владельцы могут принимать рациональные решения о влиянии изменений на сервис, грамотно реагировать на критические ситуации и, конечно же, обосновывать необходимость самого сервиса: измерять и оценивать, насколько он соответствует бизнес-целям (см. главу 6).

Независимо от того, обслуживается сервис SR-инженерами или нет, он должен поддерживать возможность мониторинга.

Мониторинг очень крупной системы может быть трудно обеспечить по нескольким причинам:

- огромное количество компонентов, которые необходимо анализировать;
- необходимость ограничивать нагрузку на инженеров, ответственных за систему.

Системы мониторинга компании Google не просто измеряют простые показатели вроде средней задержки отклика незагруженного европейского веб-сервера. Мы также

должны понимать распределение величины этой задержки среди всех серверов в данном регионе. Это знание позволит нам определить, какие факторы влияют на задержку.

Учитывая масштабы наших систем, совершенно неприемлемо проектировать их так, чтобы оповещения приходили обо всех сбоях на всех машинах. Такие данные больше похожи на «шум», и на них трудно отреагировать. Мы стараемся создавать приложения, устойчивые к сбоям систем, от которых они зависят. Вместо того чтобы требовать управления множеством индивидуальных компонентов, крупная система должна собирать сигналы и отсекать ненужные значения. Нам требуются системы мониторинга, которые позволяют отправлять оповещения о самых важных показателях сервиса, но при этом сохранять уровень детализации для того, чтобы при необходимости исследовать отдельные компоненты.

Системы мониторинга компании Google развивались на протяжении десяти лет, пройдя путь от традиционных моделей пользовательских сценариев, которые снимают показания и рассылают нам оповещения, до нового подхода. В этом современном подходе сбор временных рядов стал основным процессом новой системы мониторинга, а сценарии были заменены мощным языком для преобразования временных рядов в графики и оповещения.

Укрепление позиций Borgmon

Практически сразу после того, как в 2003 году была создана инфраструктура по планированию задач Borg [Verma, 2015], в качестве ее дополнения была спроектирована новая система мониторинга — Borgmon.⁵⁷

Мониторинг с помощью временных рядов за пределами Google

В этой главе описывается архитектура и программный интерфейс инструмента мониторинга, который является основополагающим для развития и надежности компании Google на протяжении почти десяти лет... Но как это поможет вам, наш дорогой читатель?

Не так давно в сфере мониторинга произошел Кембрийский взрыв: появились системы Riemann, Heka, Bosun и Prometheus¹. Это инструменты с открытым исходным кодом, которые очень похожи на систему оповещения Borgmon, основанную на временных рядах. В частности, Prometheus очень похожа на Borgmon, особенно если сравнивать два языка их правил. Принципы сбора переменных и вычисления правил остаются одинаковыми для всех этих программ. Сами программы предоставляют рабочую среду, где вы можете экспериментировать, и, я надеюсь, она поможет вам реализовать идеи, которые появятся после прочтения этой главы.

Вместо того чтобы выполнять предоставленные пользователями сценарии для обнаружения сбоев системы, Borgmon полагается на общепринятый формат представления данных. Это позволяет организовать масштабный сбор данных, не затрачивая слишком много ресурсов, и обойтись без выполнения подпроцессов и настройки сетевого соединения. Мы называем это мониторингом *методом белого ящика* (в главе

6 мы сравнивали виды мониторинга методами белого и черного ящика).

Данные применяются как для построения графиков и диаграмм, так и для генерирования оповещений; это делается на основе простой арифметической обработки. Поскольку сбором данных занимается уже не «короткоживущий» процесс, все накопленные данные могут быть использованы и для расчета в целях генерирования оповещений.

Эти особенности помогают «поддерживать простоту», как описано в главе 6. Они дают возможность ограничивать связанную с обслуживанием системы нагрузку на сотрудников, работающих с сервисами, чтобы им легче было реагировать на постоянные изменения, происходящие в системе по мере ее роста.

Чтобы способствовать такому масштабному сбору данных, необходимо стандартизировать формат представления показателей. Более старый метод экспорта внутренних состояний (известный как varz) был формализован, чтобы у нас появилась возможность собирать все показатели для каждого конкретного объекта с помощью единственного запроса HTTP. Например, для того, чтобы вручную запросить показатели страницы, вы можете использовать следующую команду:

```
% curl http://webserver:80/varz
http\_requests 37
errors\_total 12
```

Программа Borgmon способна получать данные от других систем Borgmon, поэтому мы можем строить иерархии, соответствующие топологии сервиса, собирая, обобщая и «прореживая» информацию на каждом уровне согласно общей стратегии. Обычно команда запускает один экземпляр системы Borgmon для каждого кластера, а также несколько экземпляров

на глобальном уровне. Очень крупные сервисы разбиваются ниже уровня кластера на множество экземпляров-скраперов, которые, в свою очередь, наполняют данными экземпляр, работающий на уровне кластера.

Инструментарий для приложений

Обработчик HTTP `/varz` просто возвращает экспортируемые переменные в виде списка в текстовом формате, в виде разделенных пробелами ключей и значений, по одной паре в каждой строке. В дальнейшем была введена связанная (*mapped*) переменная, что позволило экспортирующей стороне определить несколько меток для имени переменной, а затем экспортить таблицу значений или диаграмму. Пример переменной вида «ключ — значение» приведен ниже. Здесь мы видим 25 ответов HTTP 200 и 12 ответов HTTP 500:

```
http_responses map:code 200:25 404:0 500:12
```

Добавление показателя в программу требует всего одного объявления в том коде, где этот показатель нужен.

При взгляде назад становится очевидно, что такой текстовый интерфейс без заданной схемы значительно облегчает добавление нового инструментария, и это хорошо как для разработчиков, так и для команд SRE. Однако такой компромисс оказывается на ежедневном обслуживании. Отделение определения переменной от места, где она используется в правилах Borgmon, требует тщательного контроля за изменениями. На практике такой компромисс допустим, поскольку, помимо прочего, были созданы инструменты для проверки корректности правил и их генерирования [58](#) [59](#) [60](#)

Экспорт переменных

Компания Google глубоко пустила корни в Интернете: каждый из крупных языков, использованных в ней, имеет реализацию интерфейса экспортируемых переменных, который автоматически регистрирует HTTP-сервер, по умолчанию встроенный в каждый исполняемый файл Google². Экземпляры экспортируемых переменных позволяют автору сервера выполнять очевидные операции вроде сложения значений, установки ключа для определенных значений и т.д. Библиотека `exrvar`, написанная на языке Go³, и ее выходные данные в формате JSON представляют собой вариант такого API.

Сбор экспортованных данных

Для поиска объектов экземпляр сервиса `Borgmon` настраивается согласно списку этих целей с использованием одного из методов разрешения имен⁶¹. Список объектов зачастую является динамическим, поэтому такой подход с поиском сервисов снижает затраты на его обслуживание и позволяет выполнять мониторинг.

В заранее заданных интервалах `Borgmon` получает URI `/varz` для каждого объекта, декодирует результаты и сохраняет значения в памяти. Программа `Borgmon` также расширяет коллекцию для каждого экземпляра в списке объектов на весь интервал, поэтому коллекция для каждого из объектов не соответствует строго своим «коллегам».

Программа Borgmon также записывает «синтетические» переменные для каждой цели, чтобы определить:

- было ли разрешено имя для хоста и порта;
- ответил ли объект-цель на запрос о сборе данных;
- ответил ли объект-цель на проверку работоспособности;
- в какое время завершился сбор данных.

Эти «синтетические» переменные позволяют легко создавать правила для проверки доступности наблюдаемых задач.

Инструмент varz существенно отличается от SNMP (Simple Networking Monitoring Protocol — простой сетевой протокол мониторинга), который «разработан... для того, чтобы соответствовать минимальным требованиям к передаче данных и продолжать работать, когда другие сетевые приложения дают сбой» [Microsoft, 2003]. Использование HTTP для скраппинга противоречит этому принципу, однако опыт показывает, что это редко становится проблемой⁶². Система сама по себе разработана таким образом, чтобы быть устойчивой к сбоям сети и машины, и Borgmon позволяет инженерам писать более умные правила оповещения, используя в качестве сигналов даже сами сбои при сборе данных.

Память временных рядов как хранилище данных

Сервисы обычно состоят из множества исполняемых файлов, запущенных как множество задач на множестве машин

множества кластеров. Система Borgmon должна организованно хранить все эти данные, позволяя при этом гибко их запрашивать и делать их срезы.

Borgmon хранит все данные в базе, которая расположена в памяти и регулярно сохраняется на жесткий диск. Точки на графике записываются в формате (*временная_метка*, *значение*) и хранятся в отсортированных в хронологическом порядке списках, которые называются *временными рядами*. Каждый из них обозначается уникальным набором *меток*, имеющих вид *имя=значение*.

Как показано на рис. 10.1, временной ряд представляет собой одномерную матрицу чисел, растущую с течением времени. По мере того как вы делаете во временном ряду перестановки, матрица становится многомерной.

"http_requests"	:	:	:	:	:	:	:	:	:	:	:
	0	0	0	0	0	0	0	0	0	0	0
⋮	0	0	0	0	0	0	0	0	0	0	0
Текущий момент – $2\Delta t$	0	0	0	0	0	0	0	0	0	0	0
Текущий момент – Δt	0	0	0	0	0	0	0	0	0	0	0
Текущий момент	0	0	0	0	0	0	0	0	0	0	0
	host1	host2	host3	host4	host5	...					

Рис. 10.1. Временной ряд для ошибок, имеющий метки исходных хостов, с которых они были получены

На практике структура представляет собой блок памяти фиксированного размера, именуемый также *памятью временных рядов*. У этого блока есть сборщик мусора, удаляющий самые старые записи, когда память заполняется. Интервал между самой свежей и самой старой записями

называется *временным горизонтом* и показывает, сколько данных, готовых к запросам, хранится в оперативной памяти. Как правило, дата-центр и глобальный экземпляр Borgmon имеют размер, позволяющий хранить данные примерно за 12 часов⁶³ для визуализирующих консолей и данные за гораздо меньший промежуток времени, если речь идет о сборщиках данных низшего уровня. Для хранения данных об одной точке на графике требуется примерно 24 байта, поэтому мы можем вместить миллион уникальных временных рядов для каждой минуты из 12 часов в объем памяти, чуть меньший чем 17 Гбайт RAM.

Периодически состояние внутренней памяти сохраняется во внешнюю систему, известную как Time-Series Database (TSDB, база данных временных рядов). Программа Borgmon может запрашивать у TSDB старые данные и, хотя эта база работает медленнее, она дешевле и более емкая, чем RAM Borgmon.

Метки и векторы

Как показано в примере на рис. 10.2, временные ряды сохраняются как последовательности чисел и временных меток, которые называются *векторами*. Как и векторы в линейной алгебре, эти векторы являются срезами и поперечными сечениями многомерной матрицы точек вне памяти временных рядов. В принципе, вы можете проигнорировать временные метки, поскольку значения добавляются в вектор с заданной периодичностью — например, каждую секунду, десять секунд или минуту.

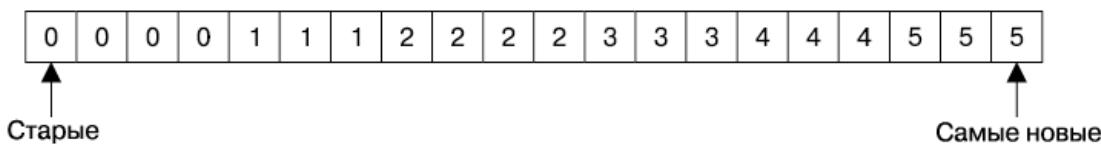


Рис. 10.2. Пример временного ряда

Этот временной называется *набором меток* (*labelset*), поскольку он реализован как набор меток, представляющих собой пары **ключ=значение**. Одна из этих меток содержит само имя переменной, а соответствующий ключ появляется на странице varz.

Некоторые имена меток объявлены как важные. Для того чтобы можно было идентифицировать конкретный ряд в базе данных, он должен иметь как минимум следующие метки:

- `var` (переменная) — имя переменной;
- `job` (задача) — имя, заданное для типа наблюдаемого сервера;
- `service` (сервис) — в свободной форме определенная коллекция задач, которые обеспечивают сервис для пользователей, как внешних, так и внутренних;
- `zone` (зона) — принятая в Google абстракция, которая ссылается на местоположение (обычно это дата-центр) экземпляра Borgmon, выполняющего сбор данных для указанной переменной.

Объединение этих переменных дает что-то подобное приведенному ниже. Это называют *переменным выражением*:

```
{var=http_requests,job=webserver,instance=host0:80,service=web,zone=us-west}
```

Запрос кциальному ряду не включает указание всех этих меток, а поиск для набора меток возвращает все соответствующие ряды вектора. Поэтому мы можем вернуть

вектор результатов, удалив метку `instance` из предыдущего запроса, если в кластере запущено больше одного экземпляра системы. Например, запрос:

```
{var=http_requests,job=webserver,service=web,zone=us-west}
```

может вернуть в качестве результата пять рядов вектора с наиболее свежими значениями временных рядов:

```
{var=http_requests,job=webserver,instance=host0:80,service=web,zone=us-west} 10
{var=http_requests,job=webserver,instance=host1:80,service=web,zone=us-west} 9
{var=http_requests,job=webserver,instance=host2:80,service=web,zone=us-west} 11
{var=http_requests,job=webserver,instance=host3:80,service=web,zone=us-west} 0
{var=http_requests,job=webserver,instance=host4:80,service=web,zone=us-west} 10
```

Источниками добавления меток во временной ряд могут служить:

- имя цели, например задание и экземпляр;
- сама цель, например переменные вида «ключ — значение»;
- конфигурация Borgmon, например примечание о местоположении или смене меток;
- модификация (перерасчет) правил Borgmon.

Мы также можем запросить временной ряд по времени, указав в выражении переменных требуемую продолжительность:

```
{var=http_requests,job=webserver,service=web,zone=us-west}(10m)
```

Такой запрос вернет хронологию соответствующих выражению временных рядов за последние 10 минут. Если мы собираем точки на графике раз в минуту, то ожидаем получить десять точек за десятиминутное окно. Результат будет выглядеть примерно так⁶⁴:

```
{var=http_requests,job=webserver,instance=host0:80, ...} 0 1 2 3 4 5 6 7 8 9 10
{var=http_requests,job=webserver,instance=host1:80, ...} 0 1 2 3 4 4 5 6 7 8 9
{var=http_requests,job=webserver,instance=host2:80, ...} 0 1 2 3 5 6 7 8 9 9 11
{var=http_requests,job=webserver,instance=host3:80, ...} 0 0 0 0 0 0 0 0 0 0 0
{var=http_requests,job=webserver,instance=host4:80, ...} 0 1 2 3 4 5 6 7 8 9 10
```

Вычисление правил

Система Borgmon, по сути, представляет собой программируемый калькулятор с небольшим количеством «синтаксического сахара»⁶⁵, который позволяет ему генерировать оповещения. Проанализировав уже описанные компоненты для сбора данных и их хранения, можно сделать вывод, что этот программируемый калькулятор подходит нам в качестве системы мониторинга :).



Централизованное вычисление правил в системе мониторинга вместо делегирования этой функции отделившимся подпроцессам означает, что вычисления могут быть запущены параллельно для многих похожих целей. Такая практика позволяет поддерживать относительно небольшой размер конфигурации (например, благодаря удалению повторяющегося кода), которая станет более функциональной благодаря своей выразительности.

Программы *Borgmon*, называемые также *правилами Borgmon*, состоят из простых алгебраических выражений, которые позволяют вычислить одни временные ряды на основании других. Эти правила могут быть довольно эффективными, поскольку дают возможность запросить хронологию одного временного ряда (например, временную ось), разные подмножества меток для нескольких временных рядов одновременно (например, пространственную ось) и выполнять многие математические операции.

Правила, если это возможно, работают параллельно в пуле потоков, но могут потребовать упорядочения, если вы используете в качестве входных данных правила, вычисляемые ранее. Размер векторов, возвращаемых с помощью запросов, также определяет общее время применения правила. Поэтому типична ситуация, когда в *Borgmon* добавляется контроль вычислительных ресурсов в ответ на то, что задача выполняется слишком медленно. При необходимости проведения более детального анализа внутренние показатели во время действия правил экспортируются для исследования

производительности и для мониторинга самой системы мониторинга.

Агрегирование — это краеугольный камень вычисления правил в распределенной среде. Агрегирование требует объединения нескольких временных рядов, чтобы рассматривать задачи как единое целое. Используя эти объединения, можно вычислить общие показатели. Например, общая интенсивность запросов для задачи в дата-центре является суммой интенсивностей изменений⁶⁶ всех счетчиков запросов⁶⁷.



Счетчик — это любая переменная, значение которой только увеличивается. Индикаторы же могут иметь любое значение.

Счетчики отражают возрастающие значения, например пройденные километры, а индикаторы показывают текущее состояние, например количество оставшегося топлива или текущую скорость. При сборе данных в стиле Borgmon предпочтительнее использовать счетчики, поскольку они не теряют свое значение, если событие происходит между запросами данных. Если какое-то действие или изменение произойдет между запросами данных, то, скорее всего, при использовании индикаторов они останутся незамеченными.

Например, для веб-сервера мы хотим добавить оповещение, которое генерируется при условии, что его кластер начинает выдавать больше ошибок выполнения запросов, чем было определено в качестве лимита. Если говорить научным языком,

оповещение должно запускаться, когда сумма интенсивностей кодов ответов, отличающихся от HTTP 200, для всех задач, отнесенная к сумме интенсивностей всех запросов для всех задач, начнет превышать некоторое значение.

Это можно сделать так.

1. Сначала мы собираем коды ответов для всех задач и размещаем их в векторах интенсивностей на текущий момент времени. Для каждого кода создается свой вектор.
2. Далее мы вычисляем общую интенсивность ошибок как сумму этого вектора, выводя на экран единственное значение для кластера в заданный момент времени. Этот общий уровень ошибок уже не учитывает коды HTTP 200, поскольку они не относятся к ошибкам.
3. После этого мы вычисляем соотношение количества ошибок с количеством запросов для всего кластера, разделив интенсивность ошибок на интенсивность поступления запросов, снова выводя на экран одно значение для кластера в заданный момент времени.

Каждое выводимое значение в любой момент присоединяется к выражению именованной переменной, что создает новый временной ряд. В результате мы можем проверить хронологию изменения уровня ошибок и частоту появления ошибок в разные промежутки времени.

Правила для интенсивности запросов можно написать на языке правил Borgmon таким образом:

```
rules <<<
  # Compute the rate of requests for each task
  from the count of requests
```

```

{var=task:http_requests:rate10m,job=webserver}
} =
    rate({var=http_requests,job=webserver}.
(10m));

# Sum the rates to get the aggregate rate of
queries for the cluster;
# 'without instance' instructs Borgmon to
remove the instance label
# from the right hand side.
{var=dc:http_requests:rate10m,job=webserver}
=
sum without
instance({var=task:http_requests:rate10m,job=we
bserver})
>>>

```

Функция `rate()` принимает заключенное в скобки выражение и возвращает разность значений, разделенную на полное время, прошедшее между получением самого раннего и самого позднего значений.

Для примера данных временного ряда из рассмотренного ранее запроса результат выполнения правила `task:http_requests:rate10m` будет выглядеть так⁶⁸:

```

{var=task:http_requests:rate10m,job=webserver,i
nstance=host0:80, ...} 1
{var=task:http_requests:rate10m,job=webserver,i
nstance=host2:80, ...} 0.9
{var=task:http_requests:rate10m,job=webserver,i
nstance=host3:80, ...} 1.1
{var=task:http_requests:rate10m,job=webserver,i
nstance=host4:80, ...} 0

```

```
{var=task:http_requests:rate10m,job=webserver,i  
nstance=host5:80, ...} 1
```

А результаты выполнения правила
dc:http_requests:rate10m будут выглядеть так:

```
{var=dc:http_requests:rate10m,job=webserver,ser  
vice=web,zone=us-west} 4
```

Второе правило использует первое в качестве входных данных.



Метка instance более не выводится, поскольку не учитывается правилами агрегирования. Если бы она оставалась в правиле, система Borgmon не смогла бы просуммировать пять рядов.

В этих примерах мы используем временное окно, поскольку работаем с дискретными точками временного ряда, в отличие от непрерывных функций. Это упрощает получение интенсивностей по сравнению с выполнением расчетов, но также означает, что для вычисления интенсивности необходимо достаточное количество отсчетов. Нам также следует учитывать вероятность того, что некоторые из последних попыток сбора данных дадут сбой. Обратите внимание, что в исходной нотации выражения с переменными используется диапазон (10m), чтобы избежать ситуации, когда данные отсутствуют из-за ошибок, возникших при их сборе.

В этом примере мы также придерживаемся нашего соглашения, позволяющего улучшить читаемость. Имя каждой вычисляемой переменной состоит из трех частей, разделенных

двоеточием. Они указывают уровень агрегирования, имя переменной и операцию, создавшую это имя. В этом примере по левую сторону находятся переменные «запросы HTTP для задач в десятиминутном промежутке» и «запросы HTTP для дата-центров в десятиминутном промежутке».

Теперь, когда мы знаем, как определить интенсивность запросов, мы можем на этой же основе вычислить интенсивность ошибок, а затем узнать соотношение ответов к запросам, чтобы видеть, сколько полезной работы выполняет сервис. Мы можем сравнить уровень (частоту появления) ошибок с целевым значением этого показателя для сервиса (см. главу 4) и сгенерировать оповещение в случае, если целевое значение не выполняется или его выполнение под угрозой:

```
rules <<<
    # Compute a rate per task and per 'code' label
    {var=task:http_responses:rate10m,job=webserve
     r} =
                           rate      by
code({var=http_responses,job=webserver}(10m));

    # Compute a cluster level response rate per
    'code' label
    {var=dc:http_responses:rate10m,.job=webserver}
    =
                           sum      without
instance({var=task:http_responses:rate10m,.job=w
          ebserver});;

    # Compute a new cluster level rate summing
    all non 200 codes
```

```

{var=dc:http\_errors:rate10m,job=webserver} =
sum without code(
    {var=dc:http\_responses:rate10m,job=webserver
.,code!=200/};

# Compute the ratio of the rate of errors to
the rate of requests
{var=dc:http\_errors:ratio\_rate10m,job=webserver} =
{var=dc:http\_errors:rate10m,job=webserver}
/
{var=dc:http\_requests:rate10m,job=webserver};
>>>

```

Опять же такое вычисление демонстрирует соглашение, в рамках которого к имени переменной временного ряда добавляется имя операции, которая ее создала. Результат можно прочесть как «процент интенсивности ошибок HTTP для дата-центра в рамках десяти минут».

Результат работы этих правил может выглядеть следующим образом⁶⁹:

```

{var=task:http\_responses:rate10m,job=webserver}
    {var=task:http\_responses:rate10m,job=webserver
.,code=200,instance=host0:80, ...} 1
    {var=task:http\_responses:rate10m,job=webserver
.,code=500,instance=host0:80, ...} 0
    {var=task:http\_responses:rate10m,job=webserver
.,code=200,instance=host1:80, ...} 0.5
    {var=task:http\_responses:rate10m,job=webserver
.,code=500,instance=host1:80, ...} 0.4

```

```
{var=task:http_responses:rate10m,job=webserve
r,code=200,instance=host2:80, ...} 1
  {var=task:http_responses:rate10m,job=webserve
r,code=500,instance=host2:80, ...} 0.1
    {var=task:http_responses:rate10m,job=webserve
r,code=200,instance=host3:80, ...} 0
      {var=task:http_responses:rate10m,job=webserve
r,code=500,instance=host3:80, ...} 0
        {var=task:http_responses:rate10m,job=webserve
r,code=200,instance=host4:80, ...} 0.9
          {var=task:http_responses:rate10m,job=webserve
r,code=500,instance=host4:80, ...} 0.1

{var=dc:http_responses:rate10m,job=webserver}
  {var=dc:http_responses:rate10m,job=webserver,
code=200, ...} 3.4
    {var=dc:http_responses:rate10m,job=webserver,
code=500, ...} 0.6

{var=dc:http_responses:rate10m,job=webserver,
code!=200/}
  {var=dc:http_responses:rate10m,job=webserver,
code=500, ...} 0.6

{var=dc:http_errors:rate10m,job=webserver}
  {var=dc:http_errors:rate10m,job=webserver,
...} 0.6

{var=dc:http_errors:ratio_rate10m,job=webserver
}
  {var=dc:http_errors:ratio_rate10m,job=webserver
er} 0.15
```

Здесь показан промежуточный запрос для правила dc:[http_errors:rate10m](#), который отфильтровывает все коды, не равные 200. Хотя значение выражений остается неизменным, вы можете увидеть, что метка для кода есть только в одном примере.

Как упоминалось ранее, правила Borgmon создают новые временные ряды, поэтому результаты вычислений хранятся в памяти временных рядов и могут быть проинспектированы точно так же, как и исходные временные ряды. Эта возможность позволяет создавать при необходимости новые специализированные запросы, выполнять их, вычислять и анализировать таблицы. Такая особенность может оказаться полезной при отладке во время дежурства, и, если эти ситуативно созданные запросы докажут свою полезность, они могут быть помещены в сервисную консоль на постоянной основе.

Оповещение

Когда правило оповещения вычисляется системой Borgmon, результатом может стать либо значение `true`, и тогда оповещение сработает, либо значение `false`. Практика показывает, что оповещения могут очень быстро менять свое состояние. Поэтому в правилах указывается минимальная продолжительность сигнала `true`, по прошествии которой отправляется оповещение. Обычно продолжительность устанавливается равной двум итерациям вычислений правил, чтобы гарантировать, что ошибки при сборе данных не вызвали ложное оповещение.

В следующем примере создается оповещение, которое срабатывает, когда уровень ошибок в течение 10 минут превышает 1 %, а общее количество ошибок превышает 1:

```
rules <<<
  {var=dc:http\_errors:ratio\_rate10m,job=webserver} > 0.01
    and by job, error
  {var=dc:http\_errors:rate10m,job=webserver} >
1
  for 2m
  => ErrorRatioTooHigh
    details "webserver error ratio at
((trigger_value))"
    labels {severity=page};
>>>
```

В нашем примере коэффициент равен 0,15, что гораздо больше установленного в качестве границы значения 0,01. Однако количество ошибок в этот момент не превышает 1, поэтому оповещение будет неактивно. Как только количество ошибок превысит 1, оповещение на две минуты получит статус «ожидает», чтобы гарантировать, что состояние не временное, и только после этого оно будет запущено.

Правило оповещения содержит небольшой шаблон для заполнения сообщения, куда входит информация о контексте: задача, для которой предназначено это оповещение, имя оповещения, числовое значение для срабатывания правила и т.д. Информация о контексте заполняется Borgmon при запуске оповещения, после чего оно отправляется в соответствующую RPC.

Система Borgmon соединена с центральным сервисом, известным как Alertmanager, который получает RPC с оповещениями, когда срабатывает правило, а затем еще раз, когда оповещение считается «отправляющимся». Alertmanager

отвечает за маршрутизацию оповещений. Он может быть сконфигурирован для выполнения таких задач, как:

- задержка некоторых оповещений, если активны другие;
- дедуплицирование оповещений с одинаковыми наборами меток от нескольких экземпляров Borgmon;
- разветвление оповещений на входе и выходе, основанное на их наборах меток, в тот момент, когда запускаются несколько оповещений с одинаковыми наборами меток.

Как это описывалось в главе 6, команды отправляют свои оповещения, требующие немедленного рассмотрения, коллегам на дежурстве, а важные, но еще не критические оповещения — в очередь несрочных оповещений (тикетов). Все остальные оповещения должны быть сохранены в качестве справочных данных для информационных панелей мониторинга.

Более подробное руководство по проектированию оповещений вы можете найти в главе 4.

Разбиваем топологию системы мониторинга на части

Система Borgmon может импортировать данные временных рядов из других экземпляров Borgmon. Если попытаться собрать данные для всех задач сервиса глобально, такой процесс может быстро стать проблемой и единственной точкой отказа. Вместо этого для передачи данных временных рядов между экземплярами Borgmon применяется потоковый протокол, экономя время центрального процессора и байты сетевого трафика по сравнению с основанным на тексте форматом varz. Типичный процесс развертывания использует

два и более глобальных экземпляра Borgmon для высокоуровневого агрегирования и один экземпляр в каждом дата-центре для наблюдения за всеми задачами, запущенными на данной площадке. (Компания Google разделяет производственную сеть на зоны для внесения изменений, поэтому наличие двух или более глобальных экземпляров предоставляет множество вариантов на случай обслуживания и отключений.)

Как показано на рис. 10.3, более сложные развертывания приводят к большему разбиению экземпляра дата-центра Borgmon на уровни скрапинга (зачастую это происходит из-за ограничений в объеме оперативной памяти и мощности процессора) и агрегирования (на этом уровне в основном выполняется вычисление правил). Иногда глобальный уровень разбивается на уровень вычисления правил и уровень информационных панелей. Экземпляр Borgmon более высокого уровня может фильтровать данные, передаваемые потоком от экземпляров более низкого уровня, чтобы глобальный экземпляр Borgmon не переполнял свою память временными рядами экземпляров нижнего уровня. Поэтому иерархия агрегирования собирает локальные кэши релевантных временных рядов, которые при необходимости можно проанализировать.

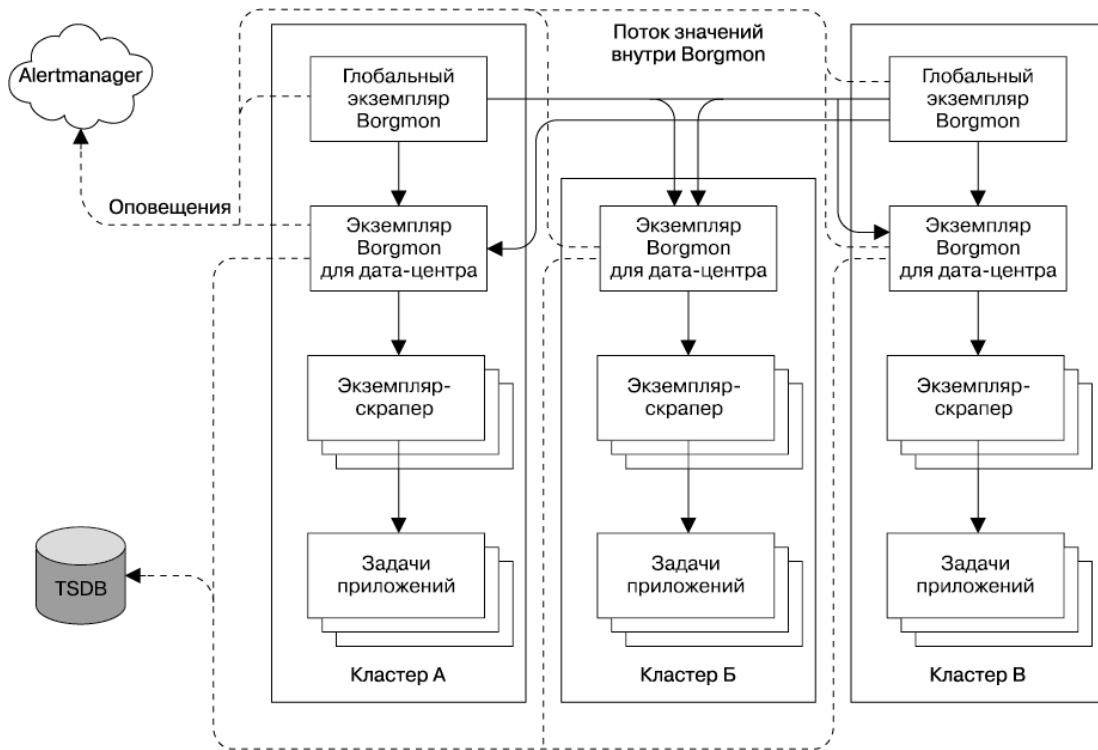


Рис. 10.3. Модель потока данных для иерархии экземпляров Borgmon в трех кластерах

Мониторинг методом черного ящика

Borgmon выполняет мониторинг методом белого ящика — система проверяет внутреннее состояние целевого сервиса, и ее правила создаются с ориентацией на это состояние. Понятный механизм такой модели дает вам широкие возможности, с помощью которых вы можете быстро понять, какие компоненты дают сбой, какие очереди переполнены и где появляются узкие места. Вы можете делать это, как реагируя на инцидент, так и тестируя развертывание новой функциональности.

Однако мониторинг методом белого ящика не дает полной картины для наблюдаемой системы. Применение только этого метода означает, что вы не знаете о том, что видят пользователи. Вы можете видеть только запросы, которые

поступают к целевому сервису. Запросы, с которыми этого не происходит из-за ошибок DNS или из-за сбоя сервера, остаются незамеченными. Вы можете отправлять оповещения только о тех сбоях, которые ожидаете увидеть.

В Google эту проблему покрытия решают с помощью инструмента Prober. Он проверяет работу протокола вплоть до конечной точки и отправляет отчет об успехе или неудаче. Prober может отправлять оповещения непосредственно Alertmanager, или же его собственные данные varz могут быть получены экземпляром Borgmon. Prober может проверить процент полезной нагрузки в ответах протокола (например, HTML-содержимое HTTP-ответа) и убедиться, что содержимое соответствует ожиданиям, и даже извлечь и экспортировать значения как временной ряд. Команды часто применяют Prober для экспорта гистограмм задержки по операциям каждого типа и по каждому диапазону объемов полезной нагрузки, что позволяет проанализировать доступную пользователям производительность. Prober — это сочетание модели проверки и тестирования с расширенными возможностями по работе с переменными, позволяющими создавать временные ряды.

Prober можно либо поместить в клиентскую часть приложения, либо связать с балансировщиком нагрузки. Используя оба варианта, мы можем обнаруживать локализованные сбои и отменять вывод оповещений. Например, нам нужно наблюдать за сайтом www.google.com, выровненным по нагрузке, и за веб-серверами каждого data-центра, находящимися за балансировщиком нагрузки. Это позволяет нам узнать, например, что при сбое data-центра трафик все еще обрабатывается, а также быстро выделить провал на графике в момент возникновения ошибок в работе сервиса.

Обслуживаем конфигурацию

Конфигурация Borgmon отделяет определение правил от наблюдаемых объектов-целей. Это означает, что один и тот же набор правил может быть применен ко многим объектам сразу и не нужно раз за разом писать одну и ту же конфигурацию. Такое разделение может показаться несущественным, но оно значительно снижает стоимость обслуживания системы мониторинга и позволяет избежать повторяемости при описании целевых систем.

Borgmon также поддерживает шаблоны языков. Эта макросоподобная система позволяет инженерам создавать библиотеки правил, которые можно использовать повторно. Такая функциональность также снижает повторяемость и, соответственно, вероятность появления ошибок в конфигурации.

Конечно, любая высокоуровневая программная среда может быть излишне сложной, поэтому Borgmon предоставляет способ создать масштабные модульные и регрессионные тесты, синтезируя их на основе данных временных рядов. Они позволяют гарантировать, что правила ведут себя именно так, как задумал их автор. Команда мониторинга систем в промышленной эксплуатации запускает сервис непрерывной интеграции, который выполняет набор этих тестов, упаковывает конфигурацию и отправляет ее во все экземпляры Borgmon. Перед тем как принять конфигурацию, экземпляры проверяют ее корректность.

В обширной библиотеке уже созданных шаблонов появились два класса конфигурации сервиса мониторинга. Первый класс просто кодирует схему переменных, экспортированных из заданной библиотеки кода, и любой пользователь библиотеки может повторно задействовать шаблон своего varz. Такие шаблоны существуют для библиотек

HTTP-серверов, выделения памяти, хранилища для клиента, общих RPC-сервисов и пр. (Несмотря на то что в интерфейсе varz схема не объявляется, для библиотеки правил, связанной с библиотекой кода, схема будет объявлена.)

Второй класс библиотек появился из-за того, что мы создавали шаблоны для управления агрегированием данных задач разного масштаба — от выполняющихся на одном сервере до глобальных сервисов. Эти библиотеки содержат правила агрегирования для экспортируемых переменных. С помощью этих правил инженеры могут моделировать топологию своих сервисов.

Например, сервис может предоставлять единый глобальный API, но располагаться в нескольких дата-центрах. Внутри каждого дата-центра сервис состоит из нескольких уровней, каждый из которых включает в себя несколько задач с произвольным количеством заданий. Инженер может смоделировать такое разбиение с помощью правил Borgmon, чтобы при отладке подкомпоненты были изолированы от остальной части системы. В таких группировках судьбы компонентов обычно взаимосвязаны: отдельных задач — из-за конфигурационных файлов, заданий — из-за того, что они размещены в одном дата-центре, а физических сайтов — из-за общей сети.

Соглашения по именованию меток сделали возможным такое разделение: Borgmon добавляет метки, указывающие имя экземпляра цели, а также логическую группу и дата-центр, где она размещается. Эти метки могут быть использованы для группирования и агрегирования временных рядов.

Поскольку у нас есть несколько вариантов применения меток для временных рядов, можно безболезненно заменять один другим.

- Метки, которые определяют разбиение самих данных (например, наш код HTTP-ответа для переменной [`http_responses`](#)).
- Метки, которые определяют источник данных (например, экземпляр или имя задачи).
- Метки, которые показывают местоположение или агрегирование данных внутри сервиса в целом (например, метка `zone`, которая описывает физическое местоположение, метка `shard`, описывающая логическое группирование задач).

Шаблонная природа этих библиотек позволяет их гибко использовать. Один и тот же шаблон можно применять для агрегирования данных на каждом уровне.

Десять лет спустя

Borgmon преобразовал модель проверки и оповещения для одной цели в массовый сбор переменных и централизованное вычисление правил для множества временных рядов, обеспечивая оповещение и диагностику.

Такое разделение позволяет масштабировать систему мониторинга в соответствии с размером контролируемой системы, но независимо от количества правил оповещения. Подобные правила проще обслуживать, поскольку они абстрагированы от принятого формата временных рядов. Новые приложения поставляются с уже готовыми для экспорта показателями для всех компонентов и библиотек, связанных с ними, а также с шаблонами для агрегирования и для консоли, что еще больше облегчает реализацию.

Гарантия того, что затраты на обслуживание растут медленнее, чем размер сервиса, — это основа, благодаря которой работа по обслуживанию системы мониторинга (и вся поддерживающая операционная работа) может быть выполнимой. Эта идея встречается во всех областях деятельности SR-инженеров, а усилия самих SR-инженеров направлены на то, чтобы довести масштаб всех сфер своей деятельности до глобального.

Десять лет — это большой промежуток времени, и, конечно же, сегодня системы мониторинга компании Google преобразились в результате экспериментов, нацеленных на их непрерывное улучшение по мере роста компании.

Даже несмотря на то, что Borgmon остается внутренней системой компании Google, идея рассматривать данные временных рядов как источник данных для генерирования оповещений теперь доступна всем благодаря инструментам с открытым исходным кодом вроде Prometheus, Riemann, Heka, Bosun, а также другим, которые могли появиться к тому моменту, когда вы возьмете в руки эту книгу.

[57](#) Prometheus — это система мониторинга и создания временных рядов с открытым исходным кодом. Доступна по адресу <http://prometheus.io>.

[58](#) Многие команды, в которых нет SR-инженеров, применяют генератор для того, чтобы избежать использования стандартных шаблонов и выполнения постоянных обновлений. Они считают, что такой генератор гораздо проще применять (хотя он и менее мощный), чем непосредственно модифицировать правила.

[59](#) Многие другие приложения используют свои протоколы сервисов для того, чтобы экспортить свое внутреннее состояние. OpenLDAP экспортирует его с помощью поддерева cn-Monitor; MySQL может отправлять отчет о состоянии с помощью запроса SHOW VARIABLES; Apache имеет собственный обработчик mod_status.

[60](#) <https://golang.org/pkg/expvar>.

[61](#) Система именования Borg (Borg Naming System, BNS) описана в главе 2.

[62](#) Вспомните, как в главе 6 мы рассматривали разницу между оповещением о симптомах и о причинах.

[63](#) Этот 12-часовой горизонт является магическим значением, которое предназначено для получения достаточного количества информации для отладки инцидента, случившегося в RAM, и касается быстрых запросов. При этом требуется совсем небольшой объем оперативной памяти.

[64](#) Метки service и zone elided здесь не приводятся ради экономии места, но они будут в выведенном программой выражении.

[65](#) Синтаксический прием, облегчающий восприятие текста программы.

[66](#) Вычисление суммы интенсивностей вместо интенсивности изменения сумм защищает итоговый результат от сбросов счетчиков или потери данных. Такие события могут произойти из-за перезапуска задачи или ошибки при сборе данных.

[67](#) Несмотря на отсутствие в varz типизации, большинство из них являются обычными счетчиками. Функция rate системы Borgmon обрабатывает все особые случаи при сбросе счетчиков.

[68](#) Метки service и zone опущены по соображениям экономии места.

[69](#) Метки service и zone опущены из соображений экономии места.

11. Быть на связи⁷⁰

Автор — Андреа Спадачини

Под редакцией Кавиты Джулиани

Быть на связи — критически важное качество, которым обязаны обладать разработчики и группы сопровождения, чтобы их сервисы оставались надежными и доступными. Тем не менее есть несколько подводных камней при организации обслуживания поступающих запросов и распределении обязанностей, которые могут привести к серьезным последствиям для сервисов и команд. Эта глава описывает принципы и подходы, разработанные в Google инженерами по обеспечению надежности информационных систем (SREs) в течение многих лет, и объясняет, как благодаря этим принципам обеспечивается надежная и устойчивая работа сервисов под нагрузкой в течение долгого времени.

Введение

Ряд профессий требуют от работников постоянно быть на связи, причем это может относиться как к рабочему, так и к нерабочему времени. В сфере информационных технологий (IT) исторически сложилась практика выделения службы оперативной поддержки — команд операторов (*operations*, или просто *ops*), главная задача которых — поддерживать сервис в исправном и работоспособном состоянии.

В Google многие сервисы, такие как, например, поиск, контекстная реклама или почта, обслуживаются командами SRE, отвечающими за их производительность и надежность. SRE-команды постоянно доступны и постоянно заняты поддержкой сервисов, за которые они отвечают. Эти команды

существенно отличаются от команд операторов тем, что они подходят к решению проблем в первую очередь со стороны разработки. Речь идет о проблемах такого уровня, что, попади они к команде операторов (как это обычно и бывает), решение вряд ли можно будет найти без участия разработчиков.

Для реализации такого подхода Google набирает в SRE-команды людей с разносторонней подготовкой и с опытом проектирования программ и программных систем. Мы отводим не более чем 50 % рабочего времени SRE-команды на исполнение функций операторов; оставшиеся не менее чем 50 % времени должны уделяться задачам разработки: усовершенствованиям сервисов и средствам автоматизации для повышения эффективности работы команды.

Жизнь дежурного инженера

В этом разделе описывается типичная деятельность дежурных инженеров, а также сообщаются некоторые базовые сведения, необходимые для остальной части главы.

Являясь хранителями систем, которые находятся в реальной промышленной эксплуатации, дежурные инженеры поддерживают назначенных операторов, устраняя выявленные сбои, а также вносят и/или анализируют изменения в работающей системе.

В условиях промышленной эксплуатации дежурный инженер должен быть доступен для выполнения операции за считаные минуты, это время согласовывается между командой и владельцами бизнес-системы. Обычно это пять минут для сервисов, с которыми пользователи взаимодействуют непосредственно, или иных критических по времени доступа сервисов и 30 минут для менее чувствительных к времени систем. Нередко компания предоставляет устройства для

оперативного получения уведомлений; чаще всего это телефон. Google имеет гибкую систему доставки, которая может рассыпать уведомления на множество устройств посредством различных механизмов (электронная почта, СМС, автоответчик, приложение).

Время отклика связано с уровнем доступности сервиса, как показано в следующем простейшем примере: если пользовательская система должна достигать в течение квартала «четырех девяток» (99,99 %), допустимое ежеквартальное времяостояния составляет около 13 минут (см. приложение А). Это означает, что время реакции дежурного инженера должно укладываться в несколько минут (если быть точнее, 13 минут). Для систем с менее жесткими требованиями к доступности (SLO) время отклика может исчисляться десятками минут.

После подтверждения получения уведомления от дежурного инженера требуется ранжировать проблему по ее приоритету и приступить к ее решению, при необходимости привлекая к этому других членов команды.

События промышленной среды без экстренных уведомлений, такие как низкоприоритетные оповещения или появление новых версий программного обеспечения, также могут быть обработаны и/или проверены дежурным инженером в рабочее время. Эти действия не столь срочные, как события с экстренными уведомлениями, которые имеют приоритет практически над всеми остальными задачами, включая и работу с проектом. (Для лучшего понимания прерываний и других событий (без уведомлений), составляющих рабочую нагрузку службы поддержки, см. главу 29.)

Практикуется деление дежурных на первую и вторую «очереди». Распределение обязанностей между ними варьируется от команды к команде. В одних командах второй

«очереди» могут передаваться те запросы, которые не успевает обработать первая. В других первая «очередь» может обрабатывать только экстренные уведомления, а все остальные события достаются второй.

Если распределение обязанностей дежурных не требует обязательного наличия второй «очереди», вместо нее часто прибегают к передаче заданий между взаимосвязанными командами — из одной в другую. Такая форма работы позволяет обойтись без отдельной второй «очереди».

Есть ряд способов организовать работу «очередей»; более подробное их рассмотрение можно найти в соответствующей главе [Limoncelli, 2014].

Сбалансированная организация дежурств

Для SRE-команд существуют определенные количественные и качественные ограничения на состав дежурных смен. Количество дежурных может быть вычислено через процент рабочего времени, затрачиваемого инженерами на обслуживание запросов. Качественный состав дежурных можно определить по количеству событий (инцидентов), происходящих за одну смену.

В обязанности SRE-менеджеров входит обеспечение сбалансированной по обеим этим осям нагрузки на дежурных.

Баланс количества

Мы искренне верим, что Е в SRE является определяющей характеристикой нашей организации, поэтому прилагаем усилия, чтобы инвестировать хотя бы 50 % времени наших SRE-команд в инжиниринг, то есть в разработку. Из другой половины на срочные работы «по вызову» может быть

потрачено не более 25 %, оставляя последние 25 % на другие виды работ, не связанные непосредственно с разработкой проектов.

Используя это правило (25 % на работы «по вызовам»), мы можем вывести минимальное количество SRE-инженеров, необходимых для обслуживания запросов в режиме 24/7. Если предположить, что на дежурстве всегда заняты два человека (первая и вторая «очереди» с разными обязанностями), минимальное необходимое количество инженеров в каждой смене на единственной площадке — восемь: при недельной длительности смены каждый инженер находится на дежурстве (в первой или второй «очереди») в течение одной недели каждого месяца. Для команд, размещенных на двух различных площадках, обоснованная минимальная численность — шесть в каждой части команды, как для соблюдения правила 25 %, так и для обеспечения достаточной «критической массы» инженеров в команде.

Если с сервисом связан достаточно большой объем работ, делающий оправданным расширение команды на данной площадке, мы предпочитаем разделить команду между несколькими различными площадками. Такая распределенная команда выгодна по двум причинам.

- Ночные смены отрицательно сказываются на здоровье работников [Durmer, 2005], а передача смен между площадками в разных часовых поясах позволяет создать «команду, над которой никогда не заходит солнце», и вовсе отказаться от ночных смен.
- Ограничение количества инженеров в дежурной смене гарантирует, что инженеры не потеряют связи с эксплуатируемой системой (см. подраздел «Коварный враг:

операционная недоработка» раздела «Избегание неуместной нагрузки» далее в этой главе).

Однако распределение команды на несколько площадок влечет за собой дополнительные коммуникационные и координационные издержки. Следовательно, решение о создании единой или распределенной команды должно приниматься с учетом особенностей каждого варианта, значимости сопровождаемых систем и трудоемкости их сопровождения.

Баланс качества

На протяжении каждой дежурной смены у инженера должно быть достаточно времени на обработку всех инцидентов и на все последующие за этим действия, например на написание отчетов (постмортемов) [Loomis, 2010]. Опишем инцидент как последовательность событий и оповещений, которые имеют одну общую причину и будут отражены в одном общем отчете. Нами обнаружено, что в среднем на выяснение причин, предотвращение последующих инцидентов и написание отчетов уходит 6 часов. Из этого следует, что за 12-часовую смену может быть максимально обработано два инцидента. Чтобы верхний предел не превышался, сопровождаемые экстренными уведомлениями события должны быть распределены стационарно, максимально равномерно, с ожидаемой медианой, равной нулю: если некоторая проблема или поведение некоторого компонента ежедневно становятся источником экстренных уведомлений (среднее количество инцидентов в день больше единицы), то велика вероятность, что в какой-то момент произойдет поломка в другом месте и количество инцидентов превысит норму.

Если лимит временно превышен, например по итогам квартала, необходимо принять ответные меры, чтобы вернуть эту нагрузку к допустимой величине и стабилизировать ее (см. подраздел «Операционная перегрузка» раздела «Избегание неуместной нагрузки» далее в этой главе и главу 30).

Компенсация

Для переработок (внеурочных часов), связанных с выполнением обязанностей дежурных, необходимо предусматривать адекватную компенсацию. Разные организации подходят к этому по-разному. Google предлагает отгулы в соответствии со временем переработки или прямую денежную компенсацию, ограниченную некоторым процентом от годовой зарплаты. На практике максимальная компенсация отражает лимит на количество часов дежурства в смене, которые может брать сотрудник. Структура компенсации гарантирует вовлечение в работу по сменам, если того требует команда, но также продвигает сбалансированное распределение работы и ограничивает возможные недостатки чрезмерной посменной работы. Компенсация строится таким образом, чтобы мотивировать к необходимой для команды работе в дежурных сменах, но в то же время обеспечить сбалансированное распределение обязанностей и ограничить такие негативные последствия чрезмерной работы «по вызовам», как неадекватное время работы над основным проектом или «выгорание» сотрудников.

Чувство безопасности

Как упоминалось ранее, большинство важнейших систем Google поддерживают SRE-команды.

Быть в SRE «дежурным» — значит брать на себя ответственность за то, с чем сталкивается пользователь, за доход от важнейших систем или за инфраструктуру, необходимую для стабильной работы всех систем. В SRE-командах тщательно обдумывать и оперативно решать проблемы — подход, жизненно важный для обеспечения успешного функционирования сервисов.

Современные исследования выделяют два различных образа действий, которым осознанно или неосознанно следует человек, когда сталкивается с проблемой [Kahneman, 2011]:

- интуитивное или автоматическое немедленное реагирование;
- рациональный, обдуманный подход к решению проблемы.

Если первый подход позволяет лучше справляться со сбоями в сложных системах, то второй с большей вероятностью дает лучшие результаты, благодаря чему снижается количество проблем в будущем.

Для того чтобы удостовериться, что у инженеров на протяжении дня будет правильный настрой, необходимо снизить стрессовую нагрузку, связанную с дежурством. Чем важнее и серьезнее конкретный сервис и последствия его сбоев, тем большее давление испытывают работающие с ним инженеры, что может привести к неуверенности и подтолкнуть команду к неверным решениям, а это может поставить под угрозу стабильную работу сервисов. Гормоны стресса кортизол и кортикотропин-рилизинг-гормон (КРГ) известны тем, что влияют на поведение, в том числе вызывают чувство страха, которое может нарушить стабильную работу организма и

заставить принимать не самые лучшие решения [Chrousous, 2009].

Под влиянием гормонов стресса тщательному изучению проблемы нередко предпочтается немедленное реагирование без обдумывания и обсуждения, что ведет к потенциальному злоупотреблению эвристикой. Эвристика очень соблазнительна для занятых в качестве дежурных. Например, если в течение недели возникают четыре одинаковые ошибки, причиной первых трех из них была некоторая внешняя система, то чрезвычайно заманчиво отнести и последнюю проблему к трем предыдущим без каких либо проверок.

Хотя интуиция и быстрота реакции могут оказаться полезными при решении проблем, у них есть обратная сторона. Интуиция может подвести, и она часто не основывается на реальных данных. Поэтому интуиция может завести работника в тупик, и он зря потратит время, следуя изначально неверным путем. Быстрая реакция основывается на привычке, на следовании стереотипам, а стереотипные реакции бездумны, и их последствия могут оказаться разрушительными. Идеальная методология обработки инцидентов должна обеспечить наиболее сбалансированный темп решения проблем, сочетая осознанные решения на основе достаточных исходных данных с критическим изучением возможных допущений.

Важно, чтобы находящиеся на дежурстве работники понимали, что они могут полагаться на некоторые ресурсы, способные упростить их работу. Важнейшие ресурсы для дежурного работника:

- хороший менеджмент внутри компании;
- хорошо определенные процедуры обработки инцидентов;

- безобвинительная культура написания отчетов [Loomis, 2010], [Allspaw, 2012].

Команды разработчиков систем, поддерживаемых SR-инженерами, обычно также участвуют в круглосуточных дежурствах и всегда могут при необходимости задействовать команды-партнеры.

Увеличение простоев (в допустимых пределах) — это, как правило, лучший подход к реагированию на серьезные сбои, когда отмечается существенная неопределенность.

Если при работе над инцидентом выясняется, что проблема достаточно сложна и требует привлечения нескольких команд, или после предварительного изучения невозможно сказать, сколько времени займет окончательное решение, уместно будет применить официальный протокол обработки (менеджмента) инцидента. Google SRE использует протокол, описанный в главе 14. Он предлагает простую и предельно понятную последовательность шагов, которые позволяют дежурному инженеру достичь удовлетворяющего решения проблемы, получая всю необходимую помощь. Этот протокол реализован в веб-приложении, которое автоматизирует большинство действий по менеджменту инцидентов, таких как назначение ролей, запись и изменение статуса проблемы. Данное приложение позволяет менеджеру сосредоточиться на решении самой проблемы и не тратить время и силы на такие рутинные действия, как заполнение электронных писем или прямое общение со многими участниками процесса.

И наконец, для каждого произошедшего инцидента очень важно понять, что же пошло не так, а что прошло гладко, и предпринять действия, которые не дадут повториться этим же ошибкам в будущем. По результатам значительных инцидентов SRE-команды должны составлять отчеты,

описывая полную хронологию произошедших событий. Ценность этих отчетов в том, что они фокусируются больше на событиях, чем на людях. Вместо обвинений кого-либо они отражают результаты систематического анализа инцидента. Ошибки случаются, и программное обеспечение должно быть создано так, чтобы свести эти ошибки к минимуму. Внедрение автоматизации и использование всех ее возможностей — лучший путь к сокращению количества ошибок, вызванных человеческим фактором [Loomis, 2010].

Избегание неуместной нагрузки

Как уже было сказано в разделе «Сбалансированная организация дежурств», на оперативную работу тратится обычно не более 50 % рабочего времени SRE-команд. Что произойдет, если этот лимит будет превышен?

Операционная перегрузка

SRE-команда и ее начальник, добавляя задачи в квартальный план работ, отвечают за равномерность нагрузки на команду.

Временное «одалживание» опытного SRE-работника в перегруженную работой команду, как это описано в главе 30, может дать достаточно времени для того, чтобы команда ощутимо продвинулась в решении проблем.

В идеале симптомы операционной перегрузки должны быть измеримы, а это означает, что и критерии оценки должны иметь количественное выражение (например, менее пяти «несрочных» запросов в день и менее двух экстренных уведомлений за смену).

Причиной перегрузок часто бывает неправильно настроенный мониторинг. Экстремальными оповещениями

должны сопровождаться только те события, которые соответствуют нарушениям работоспособности и доступности (SLO) сервисов. Необходимо также, чтобы все их можно было обработать. Оповещения с низким приоритетом, которые докучают инженеру каждый час (или даже чаще), изнуряют его, мешают нормальной работе и очень часто становятся причиной того, что более серьезные ошибки остаются без должного внимания (см. главу 29 для дальнейшего обсуждения).

Важно также контролировать количество оповещений, которые дежурный инженер получает в результате одного инцидента. Иногда одно ненормальное состояние может сгенерировать несколько оповещений, так что очень важно удостовериться в том, что взаимосвязанные оповещения группируются системой мониторинга или службой оповещений. Если по какой-то причине все-таки создаются повторяющиеся или неинформативные оповещения, возможность игнорировать их может помочь инженеру сосредоточиться на самой проблеме. Надоедливые оповещения, которые систематически порождаются по нескольку штук на один инцидент, необходимо настраивать, добиваясь соотношения ошибок и оповещений, равного 1:1. Это позволяет инженеру сосредоточиться на проблеме, а не на сортировке ошибок.

Иногда изменения, из-за которых возникает перегрузка, находятся вне ведома SRE-команды. Например, разработчики приложения могут внести изменения, система станет более «шумной», более нестабильной или и то и другое одновременно. В таком случае наилучшим решением будет работать совместно с ними для достижения общих целей по улучшению системы.

В крайнем случае у SRE-команд есть возможность «передать пейджер» — SRE могут потребовать от разработчиков заниматься исключительно функциями дежурных, пока система не будет соответствовать требованиям SRE. «Передача пейджера» — нечастое явление, потому что почти всегда есть возможность ограничиться взаимодействием с разработчиками, чтобы снизить операционную нагрузку и сделать систему более надежной. Хотя иногда комплексные или архитектурные изменения, затрагивающие несколько кварталов, могут быть необходимы для того, чтобы сделать систему более устойчивой с операционной точки зрения. Бывает, однако, что для достижения устойчивости системы (с точки зрения ее оперативного обслуживания) необходимы комплексные изменения или перестройка архитектуры, что длится несколько кварталов. В таких случаях на SRE-команду не должна ложиться слишком большая нагрузка. Вместо этого лучше всего обсудить реорганизацию дежурных обязанностей с командой разработчиков, возможно передавая несколько вызовов или их все находящемуся на дежурстве разработчику. Такое решение обычно временная мера, и команды SRE и разработчиков трудаются совместно, чтобы привести сервис в порядок и дать возможность SRE-команде снова обслуживать его самостоятельно.

Возможность переговоров о пересмотре дежурных обязанностей между SRE и командой разработчиков свидетельствует о равном вкладе обеих команд⁷¹. Такие рабочие отношения также служат примером того, как нормальное взаимодействие двух команд и их функции — устойчивость системы или добавление новой функциональности — обычно приносят пользу как сервису, так всей компании в целом.

Коварный враг: операционная недоработка

Быть дежурным при «тихой», стабильной системе — это блаженство, но что произойдет, если система слишком тихая или у SRE мало работы в качестве дежурных? Операционная недоработка для SRE-команд нежелательна. Оторванность от промышленной эксплуатации системы приводит к проблемам с уверенностью, причем это проявляется как в неуверенности, так и в излишней самоуверенности, в то время как пробелы в знаниях вскроются лишь при возникновении инцидентов. Избежать этого можно, если SRE-команды будут сформированы так, чтобы каждый инженер бывал дежурным в промышленной среде как минимум раз или два в квартал, таким образом гарантируя знакомство каждого из них с производственным процессом. Упражнения, известные как «Колесо неудачи» (описаны в главе 28), также полезны для командной работы — они помогают отточить и улучшить навыки решения проблем, а также глубже изучить сервис. Google проводит ежегодный стресс-тест для всей компании, который называется DiRT (Disaster Recovery Training). Он включает в себя теоретические и практические тренировки, чтобы успешно выполнять многодневное тестирование инфраструктуры систем или отдельных сервисов [Krishan, 2012].

Итоги главы

Подход к организации дежурств, описанный в этой главе, служит руководством для всех SRE-команд, работающих в Google, а также является ключевым фактором для создания устойчивой и управляемой рабочей среды. Подход Google позволил нам включить разработку как основное средство масштабирования функций по обеспечению промышленной

эксплуатации систем, поддерживая их высокую надежность и доступность, несмотря на рост количества систем и сервисов, за которые отвечает SRE, и их сложности. Хотя этот подход неприменим непосредственно к каждой сфере, где от IT-инженеров требуется работа в «дежурном» режиме, мы считаем, что он представляет собой проверенную модель, которую другие организации могут перенять, чтобы успешно справляться с растущим объемом таких работ.

[70](#) Ранняя версия этой главы появлялась в качестве статьи в ;login: (октябрь 2015 года, выпуск 40, № 5).

[71](#) Более подробное обсуждение распределения трудоемкости между разработкой и сопровождением продукта приводится в главе 1.

12. Эффективная диагностика и решение проблем

Автор — Крис Джоунс

Имейте в виду: чтобы быть экспертом, мало знать, как работает система. Настоящие знания приходят с изучением причин, почему система не работает.

Брайан Редман

Правильное функционирование чего-либо — особый случай среди многих случаев неправильного.

Джон Олспоу

Выявление и устранение проблем — необходимый навык для каждого, кто работает с распределенными компьютерными системами — особенно SRE, — однако он часто рассматривается как врожденный, которым кто-то обладает, а кто-то — нет. Причина в том, что это дело привычно для тех, кто занимается им регулярно. Объяснять, как решать проблемы, так же сложно, как объяснять, как ездить на велосипеде. Но мы уверены, что этому можно *научить* и можно *научиться*.

Новички часто ошибаются при решении возникших проблем. Этот процесс зависит от двух факторов: от знания хорошей стратегии решения проблем (вне зависимости от системы) и хороших знаний конкретной системы. Хотя проблему можно исследовать, пользуясь только общими методиками и правилами⁷², такой подход обычно менее

эффективен, чем основанный на понимании работы системы. Именно такие знания ограничивают эффективность SRE в новой системе. Не существует полноценной замены знаниям о том, как система построена и как она работает.

Рассмотрим общую модель процесса решения проблем. Читатели, уже имеющие опыт в этом деле, могут не согласиться с описанным далее. Если ваш метод для вас удобен и результативен, нет никаких причин изменять своим привычкам.

Теория

С формальной точки зрения процесс диагностики и решения проблем можно представить как применение гипотетико-дедуктивного метода⁷³. Используя результаты наблюдений за системой и теоретические основы для понимания ее поведения, мы последовательно выдвигаем и проверяем гипотезы о причинах произошедшей ошибки.

В идеализированной модели, как на рис. 12.1, мы бы начали с изучения отчета об ошибке, который бы сообщал, что с системой что-то не так. Затем мы проверили бы результаты телеметрии⁷⁴ и журналы системы, чтобы узнать о текущем состоянии. С этой информацией и со знанием того, как построена система, как она себя ведет и как распознает ошибки, уже можно предположить возможные причины проблемы.



Рис. 12.1. Процесс диагностики и решения проблемы

После этого можно начинать проверять свои предположения, используя один из двух возможных подходов. Мы можем сравнивать текущее состояние системы со своим предположением в поисках подтверждающих или опровергающих доказательств. Или в некоторых случаях можем специально изменить поведение системы, чтобы понаблюдать за результатом. Второй подход углубляет наше понимание состояния системы и возможных причин проблемы. Используя обе стратегии, мы постоянно тестируем систему, пока не найдем источник проблемы, после чего можно принять комплекс мер по устранению и предотвращению ее в будущем и написать отчет. Конечно, непосредственно устранение проблемы должно быть на первом месте, а поиск причин и написание отчета — уже после этого.⁷⁵

Типичные подводные камни

Случаи неэффективного решения возникающих проблем часто бывают результатом ошибок и недоработок на стадиях первичной обработки и сортировки, обследования и диагноза. Обычно эти ошибки и недоработки возникают от недостаточно глубокого понимания системы. Вот типичные подводные камни, которых следует избегать.

Мартышкин труд по изучению симптомов, которые не относятся к делу или создают неверную картину состояния системы.

Отсутствие понимания того, что и как можно изменить в системе, в ее входных данных и ее окружении для безопасной и эффективной проверки своих гипотез.

Рассматривание самых невероятных гипотез о причинах неполадок или, наоборот, зацикливание на причинах предыдущих ошибок в расчете, что они повторятся и на этот раз.

Поиск кажущихся зависимостей, которые могут оказаться случайными или же быть связаны с другими проблемами.

Избегание первой и второй ловушек – это вопрос знания системы и наличия опыта работы с общепринятыми паттернами, которые используются в распределенных системах. Третья ловушка – это набор заблуждений, которых можно избежать, раз и навсегда запомнив, что не все ошибки равновероятны, или, как говорят врачи, «если слышишь стук копыт, думай о лошадях, а не о зебрах»¹. Кроме того,

помните, что при прочих равных условиях нужно всегда предпочитать более простые объяснения¹.

Кроме того, всегда следует помнить, что корреляция между событиями не всегда означает наличие причинно-следственной связи²: некоторые взаимосвязанные события, например потеря пакетов и сбои жестких дисков в кластере, могут иметь одну причину (в данном случае перебой питания), но сами сетевые ошибки, очевидно, не будут ни причиной, ни следствием проблем с жесткими дисками. Более того, чем масштабнее и сложнее система, чем больше ее параметров мы анализируем, тем чаще будут встречаться случайно коррелирующие между собой события³.

Понимание этих ловушек в наших рассуждениях – это первый шаг к их избеганию и, как следствие, к более эффективному решению проблем. Методологический подход к осознанию того, что мы знаем, чего мы не знаем и что мы должны знать, облегчает понимание того, что пошло не так и как это исправить.

Практика [767778](#)

На практике, конечно, процесс диагностики и решения проблем никогда не бывает таким простым, как описано в нашей идеализированной модели. Есть несколько способов сделать его менее болезненным и более продуктивным и для

того, кто сталкивается с проблемами в системе, и для того, кто отвечает за их решение.

Отчет об ошибках

Любая проблема начинается с отчета: он может быть сгенерирован автоматически, либо ваш коллега может просто сказать, что «система стала медленней». Эффективный отчет должен содержать описание *ожидаемого поведения, реального поведения* и, если возможно, способов его воспроизведения⁷⁹. В идеале отчеты должны быть написаны в едином стиле и храниться в легкодоступном месте, например в системе баг-трекинга. У наших команд часто есть собственные формы или небольшие веб-приложения, которые запрашивают относящиеся к ошибке в сопровождаемой системе сведения, а затем автоматически генерируют так называемый баг (запись об ошибке, ее симптомах, причинах и мерах по устранению) и далее обеспечивает его «жизненный цикл». Хорошей идеей может быть также дополнение баг-репортера инструментами, которые бы пытались сами диагностировать и исправить проблему.

В Google нормальной практикой считается открытие бага по любой проблеме, даже для тех, которые получены по электронной почте или в корпоративном чате (например, IM). Таким образом, формируется журнал мероприятий по исследованию проблем и восстановлению, к которому можно обратиться в дальнейшем. Во многих командах по некоторым причинам не приветствуется передача отчетов непосредственно человеку: такая практика требует дополнительных действий для превращения отчета в баг, способствует появлению низкокачественных отчетов, которые не видны другим членам команды, и ведет к тому, что

основная нагрузка по решению проблемы ложится не на дежурного специалиста, а на того, кто знаком отправителю (также см. главу 29).⁸⁰

У Шекспира проблема

Вы находитесь на дежурстве по обслуживанию системы, занимающейся поиском произведений Шекспира, и получаете ошибку `ShakespeareBlackboxProbe_SearchFailure`: благодаря мониторингу методом «черного ящика» вы обнаружили, что результаты для запроса «виды существ неведомых¹» (ориг. *the forms of things unknown*) не были получены в течение пяти минут. Система оповещений заполнила баг, снабдив его ссылками на результаты мониторинга для поиска в Playbook, и направила его вам. Пора приступать к работе!

Первичная обработка и сортировка

После того как вы получили отчет об ошибке, следует выяснить, что с ней делать. Проблемы различаются по степени опасности: некоторые могут влиять только на одного пользователя с очень специфическими условиями (такие проблемы часто удается обойти), другие же могут повлечь за собой перебои в работе всего сервиса. Ваши дальнейшие действия должны быть адекватны проблеме: допустимо срочно мобилизовать всех инженеров для решения только что возникшей серьезной проблемы (см. главу 14), но не для исправления давно известного дефекта. Оценка опасности проблемы требует хорошей подготовки, опыта в

проектировании, рассудительности и очень часто умения сохранять спокойствие в любой ситуации.

Возможно, при сбоях в системе вы первым делом захотите найти источник проблемы. Не делайте так!

Вместо этого стоит попытаться хотя бы частично возобновить работу системы. Для этого может потребоваться перенаправить трафик неисправных кластеров на те, которые еще работают нормально, ограничить суммарный трафик, чтобы избежать перегрузки и последовательного отключения всей системы, или же отключить отдельные подсистемы, чтобы снизить нагрузку. Приоритетной задачей должна быть «остановка кровотечения»; вы никак не поможете пользователям, если будете искать причины неисправности, пока ваша система умирает. Конечно, быстрота оказания «первой помощи» не должна помешать сохранению информации о том, что произошло, например, в виде записей в журналах, которые позже помогут вам в поиске причины.

Молодых пилотов учат, что в случае ЧП они все равно должны пилотировать самолет [Gawande, 2009]; поиск проблемы вторичен по сравнению с тем, что нужно безопасно посадить самолет на землю. Такой подход справедлив и для компьютерных систем: например, если неполадка ведет к тому, что часть информации будет потеряна навсегда, «заморозка» системы для предотвращения дальнейшего ее распространения будет наилучшим решением.

Такой порядок нередко смущает неопытных SR-инженеров и кажется им противоестественным, особенно если ранее они занимались разработкой программных продуктов.

Обследование

У нас должна быть возможность подробно исследовать, что происходило с каждым компонентом системы, и установить, правильно ли он работал.

В идеале система мониторинга ведет запись показателей функционирования системы, как это было упомянуто в главе 10. Эти данные — хорошая отправная точка для выяснения, что именно пошло не так. Последовательная запись хронологии операций — эффективный путь к пониманию поведения отдельных компонентов системы, обнаружению зависимостей и, таким образом, к нахождению источника проблемы⁸¹.

Второй незаменимый инструмент — журналирование. Изучение информации о каждой операции и о состоянии системы позволяет понять, что делал каждый процесс в данный момент времени. Возможно, вам придется анализировать журналы одного или нескольких процессов. Трассировка запросов через всю иерархию процессов с использованием таких инструментов, как Dapper [Sigelman, 2010], облегчает понимание работы распределенной системы посредством так называемых диаграмм использования (use-case), что позволяет применять затем различные анализаторы [Sambasivan, 2014].

Журналирование

Текстовые, пригодные для чтения человеком журналы очень полезны для оперативной отладки в реальном времени, в то время как журналы в структурированном бинарном формате позволяют создавать и использовать инструменты, проводящие ретроспективный анализ на основе большого объема информации.

Очень полезно иметь несколько уровней детализации, а также возможность повышать уровень по мере надобности, на лету. Такой подход позволяет детально изучать любую операцию (или даже все) без перезапуска процесса, а также возвращать (понижать) уровень детализации журналов сервиса, не нарушая его работу. В зависимости от объема трафика, который обрабатывает ваш сервис, может быть полезно использовать «прореженную» выборку; например, вы можете выбирать для протоколирования каждую тысячную операцию.

Следующим шагом может стать добавление языка для правил фильтрации, например «Показать операции, которые соответствуют X», где X может быть любым правилом, например «выбрать вызовы удаленных процедур (RPC) с запросами менее 1024 байт, или операции, ожидание результата которых составило больше 10 мс, или только обращения к процедуре doSomeThingInteresting() в rpc_handler.py». Возможно, вы захотите построить свою систему журналирования, которую можно было бы задействовать в любой момент, быстро и с требуемой избирательностью.

Раскрытие текущего статуса — третий козырь в нашем рукаве. Например, у серверов Google есть интерфейсы, которые демонстрируют в качестве примеров недавно выполненные вызовы удаленных процедур, что позволяет понять, как сервер общается с другими серверами без обращения к описывающей архитектуру документации. Эти интерфейсы также

показывают гистограммы частоты ошибок и величин задержек для каждого типа УВП, что позволяет быстро узнать, где начались сбои. Некоторые системы имеют интерфейсы, показывающие их текущую конфигурацию или позволяющие анализировать их данные. Например, серверы Google's Borgmon (см. главу 10) могут вывести условия, по которым осуществляется мониторинг, или даже сделать пошаговую трассировку конкретных алгоритмов в поисках источника того или иного выходного значения.

Наконец, вам, возможно, понадобится создать тестовое приложение-клиент, чтобы понять, что тестируемый компонент возвращает в ответ на запросы.

Отладка Шекспира

Используя ссылку, предоставленную в отчете мониторинга методом черного ящика, вы обнаруживаете, что к интерфейсу `/api/search` был отправлен запрос HTTP GET:

```
{  
    'search_text' : 'the forms of things  
unknown'  
}
```

Ожидается, что он получит HTTP-ответ с кодом 200 и данными из JSON, которые выглядят следующим образом:

```
({"work": "A Midsummer Night's Dream",  
     "act": 5,  
     "scene": 1,  
     "line": 2526,
```

```
        "speaker": "Theseus"  
    })
```

Система настроена так, что тестовый запрос отсылается раз в минуту; за последние десять минут примерно половина из них была успешной, явной закономерности при этом незаметно. К сожалению, мониторинг не показывает коды ошибок; вы делаете пометку о том, что позже нужно это доработать.

Используя curl, вы вручную отправляете запрос к проверяемому интерфейсу и получаете HTTP-ответ с кодом ошибки 502 (Bad gateway – «ошибка шлюза») без какого-либо содержимого. У него есть HTTP-заголовок, поле `X-request-trace`, в котором показаны адреса серверов, участвовавших в выполнении этого запроса. С имеющейся информацией вы можете начать тестировать эти серверы на предмет того, правильно ли они реагируют на запросы.

Диагноз

Понимание того, как спроектирована конкретная система, действительно полезно для выдвижения реалистичных гипотез о сути и причинах неполадок, но есть и ряд общих подходов, применимых к любым системам.

Упрощайте и сокращайте. В идеале все компоненты в системе имеют хорошо определенные интерфейсы, которые выполняют преобразование их входных данных в выходные (в нашем случае, получив введенный для поиска фрагмент, компонент должен вернуть текст, в котором найдены

совпадения с введенным). После этого можно взглянуть на соединение между компонентами или, что равноценно, на передающуюся между ними информацию, чтобы определить, правильно ли работает компонент. Особенно эффективным бывает ввод заранее известных тестовых данных с последующей сверкой результата (тестирование по методу черного ящика) на каждом шаге обработки, равно как и ввод данных, подготовленных специально для наблюдения возможных причин ошибок. Наличие таких воспроизводимых наборов тестов существенно ускоряет отладку. Кроме того, появляется возможность использовать их и вне «промышленного» окружения, то есть в условиях, когда допустимы гораздо более агрессивные и рискованные воздействия на систему.

«Разделяй и властвуй» — принцип, применимый в решении самых разнообразных задач. В многоуровневой системе, где работа протекает через иерархию (стек) компонентов, обычно лучше начинать отладку с одного конца стека, методично анализируя работу компонентов, один за другим. Такая же стратегия подходит и для использования с системами конвейерной архитектуры. В очень больших системах линейное продвижение от компонента к компоненту может оказаться слишком длительным. Тогда хорошей альтернативой будет *разделить систему на две части* и начать тестирование с точки «перехода» между ними. Определив, какая часть системы работает правильно (и может быть исключена из рассмотрения), продолжайте деление, пока не дойдете до неисправного компонента.

Спрашивайте «что», «где» и «почему». Сбоящая система все еще пытается работать и делает что-то — но совсем не то, что от нее требуется. Нужно выяснить, что именно она делает, почему это происходит и, наконец, где расходуются ее ресурсы

и куда направляются результаты, — все это может помочь вам в понимании того, что же пошло не так [82](#) [83](#).

Вскрытие причин наблюдаемых симптомов

Симптом: в кластере Spanner очень большая задержка, поэтому вызовы удаленных процедур не успевают дойти до сервера и завершаются по тайм-ауту.

Почему: процессор сервера Spanner перегружен задачами и не успевает обслужить все запросы от клиентов.

Где расходуется время процессора? Изучение сервера показало, что он сортирует записи в журналах, записанных на диске.

Где именно код сортировки тратит время? Он сравнивает пути к файлам журналов на соответствие с регулярными выражениями.

Решения: переписать регулярные выражения, чтобы в них не использовались обратные ссылки (backtracking). Попробовать найти уже проверенное решение для похожих задач. Подумайте о переходе на новую версию регулярных выражений RE2, которая не использует обратные ссылки и гарантирует линейную зависимость времени обработки от объема входных данных².

Какое воздействие было последним. У систем есть инерция: мы выяснили, что не подверженная внешним воздействиям компьютерная система имеет тенденцию продолжать работать, например, пока не изменится конфигурация или тип нагрузки. Недавние изменения в системе могут быть отличным местом к расследованию того, что пошло не так. Поэтому хорошей отправной точкой для изучения проблемы будет анализ последних изменений в системе⁸⁴.

Хорошо спроектированные системы должны предусматривать средства для полномасштабного протоколирования всех изменений в версиях установленного ПО и в конфигурации на всех уровнях иерархии от файлов программ на сервере, которые обрабатывают пользовательский трафик, и до пакетов, установленных в отдельных узлах в кластере. Взаимосвязь изменений производительности системы и ее поведения с другими событиями в системе и ее среде также могут быть полезны для «табло» мониторинга. Например, вы можете снабдить график детализаций, показывая частоту появления ошибок в период установки новой версии ПО (рис. 12.2).

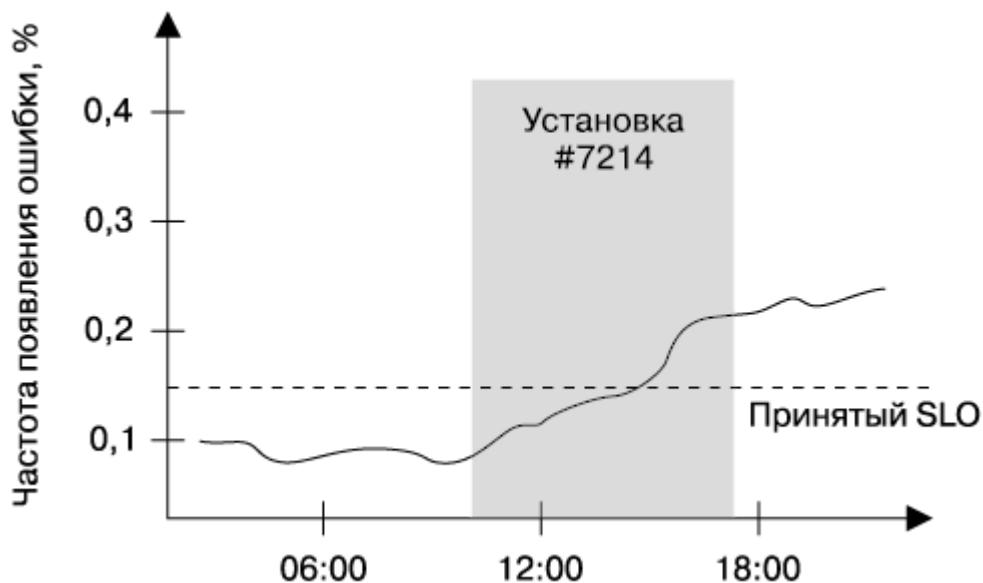


Рис. 12.2. Частота ошибок с момента установки и удаления новой версии

Отправляя вручную запросы интерфейсу `/api/search` (см. врезку «Отладка Шекспира» на с. 188) и анализируя списки серверов, участвовавших в передаче результатов, вы можете исключить из рассмотрения проблемы API первого сервера, принял запрос, или балансировщик загрузки: ответ вряд ли будет включать в себя такую информацию, если запрос вообще не дошел до поисковых сервисов и потерялся где-то по пути. Теперь вы можете сосредоточиться на бэкенде — анализируя журналы, отправляя тестовые запросы, проверяя полученные ответы и другие показатели работы системы.

Специфические средства диагностики. Рассматриваемые до сих пор универсальные методы и средства подходят для решения широкого круга задач, но вы, вероятно, считаете, что полезны могут быть и инструменты для диагностики ваших конкретных сервисов. Действительно, SR-инженеры Google тратят немало времени на создание таких инструментов. Хотя

многие из этих инструментов подходят только для конкретной системы, обязательно выясните, нет ли для других сервисов и у других команд чего-то подобного, чтобы не тратить время и усилия на ненужное дублирование.

Подтверждение диагноза

Когда у нас остался уже небольшой список возможных причин, пора попытаться найти, какая же именно из них стала источником проблемы. Подтвердить или отклонить гипотезы можно экспериментальным путем. Пусть, например, мы считаем, что проблема вызвана ошибками сети между сервером приложений и сервером базы данных или отказом в соединении со стороны базы данных. Попытки подключиться к БД с таким же набором параметров, как и у сервера приложений, позволяют опровергнуть вторую гипотезу, а проверка доступности сервера БД при известных топологии сети, настройках сетевых экранов и других факторах позволяет опровергнуть первую. Следуя за кодом программ и пытаясь шаг за шагом имитировать его выполнение, мы можем выйти на источник проблемы.

Разрабатывая тесты (которые могут быть и достаточно простыми, как, например, отправка пинга, и, наоборот, достаточно сложными, например отключение трафика в кластере и ввод специально сформированных запросов для выяснения условий возникновения «гонок»), надо руководствоваться рядом соображений.

- В идеальном случае варианты результатов тестов должны быть взаимоисключающими, чтобы с их помощью можно было однозначно подтвердить или опровергнуть группы гипотез. На практике достичь этого не так-то просто.

- Проверяйте в первую очередь очевидные вещи: выполняйте тесты, начиная с самых очевидных и переходя последовательно к более неочевидным, с учетом возможных рисков для системы. Например, есть смысл проверить сначала сетевое соединение между двумя машинами и лишь потом выяснить, не был ли закрыт доступ для этого пользователя в новой версии конфигурации.
- Результаты эксперимента могут ввести вас в заблуждение. Например, настройки сетевого экрана могут запрещать доступ только с конкретного IP-адреса, база данных может не пинговаться с вашего компьютера, но будет доступна для сервера приложений.
- Активное тестирование может привести к побочным эффектам, которые повлияют на результат будущих тестов. Например, если позволить процессу использовать больше ресурсов процессора, это ускорит выполнение операций, но может привести к проблемам с многопоточностью. Аналогично включение подробного журналирования может усугубить проблему с задержками и даже исказить результаты теста, и тогда возникает вопрос: проблема усугубляется сама по себе или же из-за журналирования?
- Результаты некоторых тестов бывают слишком ненадежны, чтобы однозначно полагаться на них; их можно лишь принимать к сведению. Например, проблемы многопоточности и «гонок» бывает очень сложно воспроизвести, поэтому вам иногда придется принимать решения без твердых доказательств, в чем именно состоит проблема.

Тщательно записывайте, какие идеи у вас были, какие тесты вы проводили и какие получили результаты⁸⁵. Эта информация может иметь решающее значение, особенно в сложных и запутанных случаях, напоминая, что именно происходило и какие шаги уже предпринимались, чтобы не пришлось их снова повторять. При активном внесении изменений в систему в ходе тестирования, например отдавая больше ресурсов процессу, такие заметки позволяют намного легче и быстрее вернуть систему в первоначальное состояние.

Волшебная сила отрицательных результатов

Автор — Рэндалл Босетти.

Под редакцией Джоан Вендт

Отрицательным называют такой результат, когда не был достигнут ожидаемый (положительный) результат, то есть эксперимент прошел не так, как планировалось. Сюда относятся новые технические решения, эвристики и выполняемые человеком операции, которые не дали ожидавшихся улучшений в системе.

Не игнорируйте отрицательные результаты. Понимание своей ошибки представляет большую ценность: очевидный отрицательный результат может помочь в решении самых сложных вопросов — касающихся проектирования системы. Очень часто у команды есть два пути проектирования, кажущиеся одинаково обоснованными, но продвижение по одному из них сопровождается сомнениями, что другой был бы лучше.

Отрицательный результат — окончательный вердикт. Он дает надежную и достоверную информацию о «промышленной» или «разработчикской» конфигурациях системы, о пределах ее производительности. Он может помочь

другим решить, стоит ли им тестировать (или проектировать) те или иные вещи. Например, команда разработчиков решила не использовать определенный веб-сервер, потому что он может поддерживать без блокировок лишь около 800 соединений, а требуется 8000. Когда следующая команда решит оценить этот сервер, вместо того чтобы все начинать с нуля, они смогут воспользоваться задокументированным отрицательным результатом и быстро решить, что: 1) им требуется менее 800 подключений или 2) необходимо решить проблемы с блокировками соединений.

Даже если отрицательный результат неприменим напрямую к чьему-либо эксперименту, та информация, которую он дает, может помочь планировать новые эксперименты или избегать повторения уже известных проблем. Это могут быть микротесты производительности, задокументированные неудачные решения или отчеты об авариях. Вам следует оценить и обсудить масштабы влияния отрицательных результатов при планировании эксперимента, поскольку затрагивающий многие системы и стабильно повторяющийся отрицательный результат для ваших коллег может оказаться даже важнее, чем для вас.

Инструменты и методы переживают конкретный эксперимент и помогают в будущих работах. Например, наличие готовых средств для тестирования и измерений или для генерации тестовой нагрузки позволяет достаточно легко сделать эксперимент более информативным и достоверным. Многие веб-мастера с пользой применяют Apache Bench (инструмент для тестирования нагрузочной способности веб-серверов), выполняющий большую, сложную и кропотливую работу, хотя поначалу его результаты скорее разочаровывали.

Польза от создания инструментов для повторяющихся экспериментов может проявиться позже: пусть сейчас какое-то

ваше приложение не выигрывает от размещения его базы данных на SSD или от ее индексации — от этого может выиграть другое приложение. Если вы написали скрипт, позволяющий легко опробовать эти конфигурации, он поможет оптимизировать и ваш будущий проект.

Публикация отрицательных результатов повышает культуру производства в области обработки данных. Накопление отрицательных результатов и, казалось бы, несущественных сведений способствует повышению качества набора контролируемых параметров и служит примером рационального отношения к ограждам в работе. Открывая другим как можно больше подробностей, вы стимулируете и остальных поступать так же, и это подталкивает всю отрасль в правильном направлении. Наши SR-инженеры уже усвоили это и создают качественные детальные отчеты, что благотворно повлияло на стабильность промышленно эксплуатируемых продуктов.

Публикуйте свои результаты. Если вам нужны результаты экспериментов, то с большой вероятностью они нужны и другим людям. Если вы поделитесь своими результатами, другим не придется проектировать с нуля или ставить похожие эксперименты. Мало кто хочет публиковать отрицательные результаты, поскольку они свидетельствуют о неудаче эксперимента. Некоторые эксперименты обречены изначально, но отчеты о них все равно просматривают. Сведения о множестве экспериментов так никогда и не публикуются, потому что люди думают, что получение отрицательного результата не является движением вперед.

Внесите свой вклад, рассказав всем о своих дизайне, алгоритмах и рабочем процессе в команде. Объясните своим коллегам, что отрицательные результаты — это часть необходимых и ожидаемых рисков разработки и в каждой

хорошо спроектированной системе есть и их заслуга. Будьте скептичны, когда видите конструкторский документ, анализ производительности или описание, в которых не упоминается о неудачах. Это говорит о том, что либо документ был сильно «вычищен», либо автор был недостаточно точен и честен.

И наконец, публикуйте результаты, которые кажутся вам удивительными, чтобы другие — даже вы сами в будущем — не удивлялись им заново.

Лечение

Итак, в идеале вы свели все причины к одной. Теперь мы должны доказать, что это действительно та самая причина. Исчерпывающее доказательство того, что конкретная причина привела к проблеме, предполагает искусственное воспроизведение ситуации, а это в условиях промышленной эксплуатации может быть затруднено. Из-за нескольких факторов нам часто удается найти только *вероятную* причину.

- *Сложность систем.* Скорее всего, было несколько факторов, из которых каждый отдельно взятый сам по себе причиной не был, но все вместе они привели к сбою⁸⁶. Кроме того, поведение реальных систем часто зависит от предыдущих состояний, поэтому для воспроизведения ошибки необходимо, чтобы система уже оказалась в соответствующем, возможно, редком состоянии.
- *Воспроизвести проблему в работающей промышленной системе может быть невозможно* как из-за сложности приведения ее в нужное состояние, так и из-за неприемлемого времени восстановления работоспособности. Наличие не участвующей в производственном процессе копии системы решает эту

проблему, но это требует затрат на поддержание ее в работоспособном и адекватном состоянии.

Как только вы нашли факторы, которые вызвали проблему, самое время все задокументировать: что происходило с системой, как вы выследили проблему, как вы справились с ней и как сделать так, чтобы она не повторилась в будущем. Другими словами, вам нужно написать отчет (*несмотря на то что система работает!*).

Пример: анализ реальной ситуации

App Engine («движок приложений»)⁸⁷ — часть облачной платформы Google. Это продукт типа «платформа как сервис», который позволяет разработчикам создавать свои сервисы поверх инфраструктуры Google. Один из наших внутренних клиентов прислал отчет об ошибке. В нем указывалось, что у них резко увеличились задержка, использование ЦП и количество процессов, нужных для обработки трафика в их приложении — системе управления контентом, используемой для создания документации для разработчиков⁸⁸. Клиент не находил в коде таких изменений, которые могли бы потребовать больше ресурсов, трафик приложения тоже не увеличивался (рис. 12.3), и они недоумевали: неужели это произошло из-за изменений в самом сервисе App Engine?

В ходе расследования мы узнали, что задержка действительно на порядок возросла (рис. 12.4) и одновременно вчетверо увеличились загрузка ЦП (рис. 12.5) и количество процессов (рис. 12.6). Что-то явно было не так. И мы начали искать проблему.

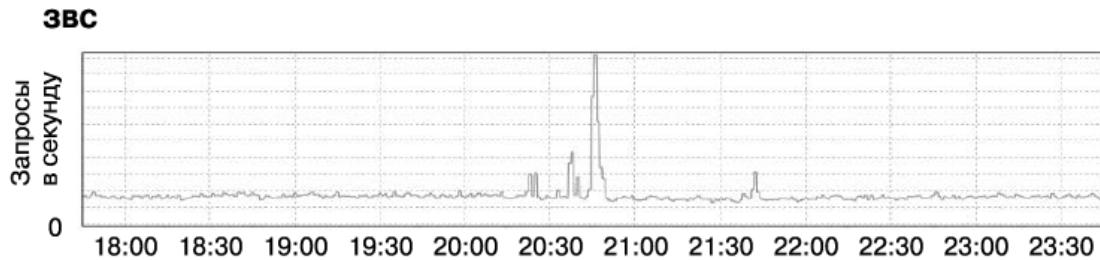


Рис. 12.3. Количество запросов, которое приложение получало в секунду. Видны моменты пиковой нагрузки, после чего она возвращается к нормальной

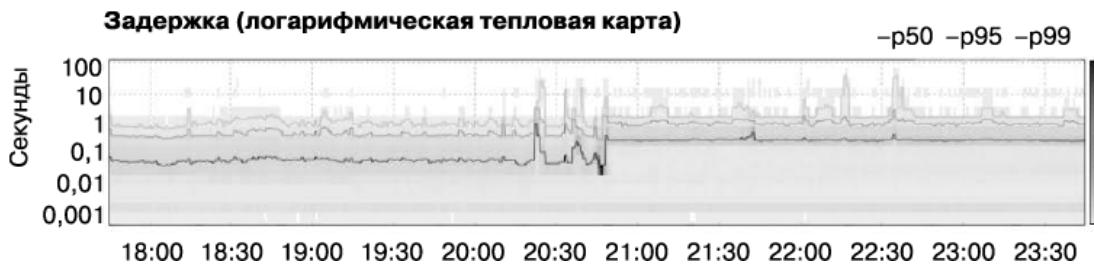


Рис. 12.4. Задержка приложения. Кривые на графике соответствуют величинам задержки, в которые укладывается выполнение соответственно 50, 95 и 99 % запросов. «Карта нагрева» показывает, сколько запросов попадает в каждый интервал

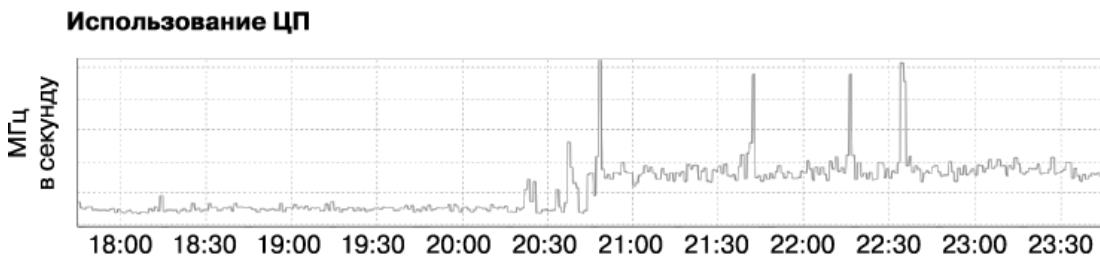


Рис. 12.5. Совокупное использование ЦП приложением

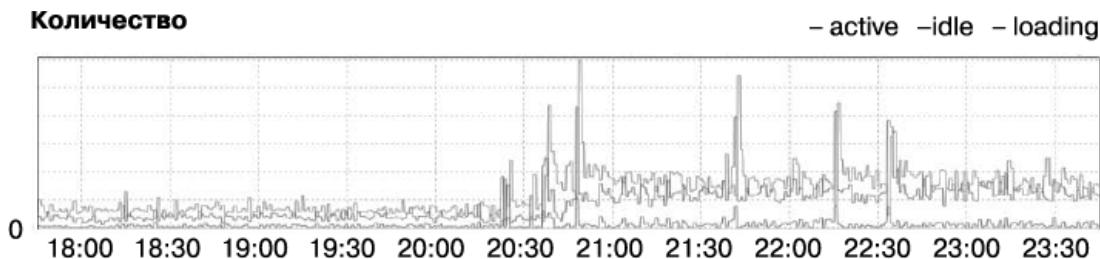


Рис. 12.6. Количество запущенных процессов (экземпляров приложения)

Резкое увеличение задержки и потребления ресурсов обычно указывает на то, что увеличилось количество

отсылаемого системе трафика или что изменения произошли в конфигурации системы. Однако здесь мы могли легко отбросить обе эти причины: поскольку рост потребления ресурсов вызван всплеском трафика приложения около 20:45, что может объяснить краткий всплеск в использовании ресурсов, следовало бы ожидать нормализации, как только количество запросов снова снизится до обычного. Такая загрузка не должна была продолжаться в течение нескольких дней, с тех самых пор, как разработчики заполнили отчет об ошибке, а мы начали разбираться с проблемой. Кроме того, перебои начались в субботу, когда некому было менять ни приложение, ни его окружение. Последние изменения кода и конфигурации были сделаны несколько дней назад. Однако, если бы источник проблемы был в сервисе, то такой же эффект наблюдался бы у всех приложений, использующих данную инфраструктуру. Но таких приложений не было.

Мы рассказали о проблеме нашим разработчикам App Engine, чтобы выяснить, не связаны ли проблемы клиента с какими-то особенностями этого сервиса. Они тоже не нашли ничего странного, однако один из разработчиков заметил совпадение между увеличением задержки и увеличением количества специфических обращений к API хранилища данных — вызовов `merge_join`, которые часто указывают на неоптимальную индексацию данных из базы. Включение в индексацию тех свойств объектов, которые приложение использовало при их выборке из хранилища, ускорило бы эти запросы и, скорее всего, производительность приложения в целом, но мы должны были понять, *какие именно* свойства подлежат индексации. Беглый анализ кода приложения явных «подозреваемых» не выявил.

Наступило время тяжелой артиллерии: используя Dapper [Sigelman, 2010], мы протрассировали прохождение HTTP-

запроса — начиная от его поступления на прокси клиентской стороны и до момента формирования ответа приложением, а затем просмотрели вызовы удаленных процедур всех серверов, участвующих в выполнении этого запроса. Это позволило нам увидеть, какие свойства были включены в выборку из хранилища данных, а затем создать нужные индексы.

В ходе этого исследования мы обнаружили, что запросы к статичным данным, например графическим файлам, в обслуживании которых хранилище не участвует, выполнялись намного медленнее, чем обычно. Обратившись к графикам обращений к отдельным файлам, мы увидели, что всего пару дней назад эти запросы выполнялись намного быстрее. Это означало, что найденная ранее корреляция между `merge_join` и увеличением задержки была ложной и что наша гипотеза о неоптимальной индексации в корне неверна.

Продолжив изучение аномально медленных запросов к статичному содержимому, мы нашли, что большинство вызовов из приложения проходят через сервис кэширования и поэтому должны выполняться очень быстро — за несколько миллисекунд. Выходило, что запросы должны быть очень быстрыми и вроде бы не могли стать источником проблемы. Однако задержка с момента начала обработки запроса приложением и до первого вызова RPC составляла примерно 250 миллисекунд, в течение которых приложение, скажем так, «что-то делало». Поскольку код, выполняемый сервисом App Engine, предоставляется пользователями, SR-инженеры его не изучают, и мы не могли узнать, что же делает приложение в это время. Dapper также не смог отследить происходящее, так как он видит только вызовы удаленных процедур, которых в этот период не было.

Мы решили, что пока не будем отгадывать эту загадку. У клиента было запланировано открытие доступа к приложению

на следующей неделе, и мы не были уверены, что сумеем быстро выяснить суть проблемы и решить ее. Вместо этого мы порекомендовали клиенту выделить приложению больше ресурсов, запросив наиболее мощный ЦП. Сделав это, клиент сократил задержку до приемлемого уровня, хоть и не до такого низкого, как нам хотелось бы. Было решено, что такого уменьшения задержки достаточно, чтобы команда могла считать запуск своего приложения успешным, а изучение проблемы продолжить в свободное время⁸⁹.

К этому моменту, мы начали подозревать, что приложение было жертвой другой распространенной причины внезапного увеличения задержки и потребления ресурсов: изменения в режиме работы. Мы заметили рост количества операций в хранилище данных из приложения как раз перед тем, как увеличилась задержка, но этот рост не был ни большим, ни длительным — и мы отбросили это совпадение как случайное. Однако такое поведение приводило к мысли об общем шаблоне: каждый экземпляр приложения при инициализации считывает объекты из хранилища данных, а затем хранит их в памяти. Таким образом приложение избегает повторного чтения редко изменяемых конфигураций из хранилища, вместо этого обращаясь к объектам в памяти. Далее, длительность обработки запросов часто зависит от объема конфигурационных данных⁹⁰. Мы не могли доказать, что именно это было причиной проблемы, но такое типичное решение обычно считается нежелательным⁹¹.

Разработчики приложения добавили инструментарий, чтобы понять, где приложение тратит свое время. Они выявили тот метод, который вызывался при каждом запросе, — он проверял, имеет ли пользователь доступ к данному пути в файловой системе. Метод использовал кэширование объектов «белого списка» в памяти приложения, чтобы минимизировать

обращения к хранилищу и к сервису кэширования. Как заметил один из разработчиков, «я не знаю, где тут огонь, но дым от закэшированного “белого списка” буквально ест глаза».

Некоторое время спустя проблема была найдена, это был давно существовавший дефект в системе управления доступом: всякий раз при обращении по конкретному пути выполнялось создание объекта «белого списка» и запись его в хранилище. При запуске приложений автоматический сканер безопасности проверял их на уязвимости и как побочный эффект в процессе проверки создавал «белый список» из тысяч объектов в течение примерно получаса. Это множество избыточных объектов затем участвовало в проверках доступа при каждом запросе, что и приводило к задержкам ответов, причем без вызовов удаленных процедур других сервисов. Исправление дефекта — удаление этих объектов после использования — восстановило производительность приложения до ожидаемого уровня [92](#).

Как облегчить решение проблем

Есть много подводов к тому, как упростить и ускорить диагностику и решение проблем. Вероятно, наиболее общими из них являются следующие.

- Обеспечьте возможность наблюдения за системой: в виде показателей белого ящика и структурированных журналов — снизу доверху, для каждого компонента.
- Разрабатывайте системы с предельно понятными и наблюдаемыми интерфейсами между компонентами.

Доступность информации в целостном виде на любой стадии ее обработки в системе — например, с использованием

уникальных идентификаторов запросов во всех вызовах удаленных процедур всех компонентов — дает возможность меньше заниматься гаданием, *какие места* в журналах прохождения запроса и ответа на него соответствуют друг другу. Это, в свою очередь, позволяет сократить время диагностики и восстановления.

Проблемы, которые приходится искать, часто бывают следствием неверного представления об изменениях в состоянии системы и ее окружения. Упрощение, контроль и журналирование этих изменений делает сами проблемы более редкими, а их диагностику и решение — более простыми.

Итоги главы

Итак, мы рассмотрели кое-какие меры, которые вы можете предпринять, чтобы сделать процесс диагностики и решения проблем ясным и понятным для новичков, в результате чего они тоже смогут стать квалифицированными специалистами. Применение систематического подхода к решению проблем — вместо того чтобы полагаться только на удачу или опыт — позволяет сократить время, необходимое для восстановления системы, что делает работу с ней более продуктивной, удобной и привлекательной для пользователей.

[72](#) Действительно, использование только общепринятых практик и личных навыков решения проблем часто может быть отличным способом понять, как работает система (см. главу 28).

[73](#) См https://en.wikipedia.org/wiki/Hypothetico-deductive_model.

[74](#) Например, экспортируемые переменные, описанные в главе 10.

[75](#) «Зебра», см. [https://en.wikipedia.org/wiki/Zebra_\(medicine\)](https://en.wikipedia.org/wiki/Zebra_(medicine)) — профессиональный сленг, обозначающий экзотический диагноз. Выражение приписывается Теодору Вудварду из университета Maryland School of Medicine. Это правило применимо не всегда, но для некоторых систем целые классы проблем могут быть исключены из

рассмотрения: например, в хорошо спроектированном кластере файловых систем задержка из-за неисправности одного диска крайне маловероятна.

[76](#) Бритва Оккама, см. https://en.wikipedia.org/wiki/Occam's_razor. Но помните также, что проблема может быть не единственной. Например, наблюдаемый в системе набор симптомов с большей вероятностью бывает следствием совместного действия ряда мелких дефектов, чем одной редкой ошибки, которая также объясняет все эти симптомы. См. https://en.wikipedia.org/wiki/Hickam's_dictum.

[77](#) Конечно же, см. <https://xkcd.com/552>.

[78](#) Например, у нас нет правдоподобной теории, которая бы объясняла, почему в период с 2000 по 2009 год в США количество защит диссертаций на степень PhD по информатике весьма точно ($r^2 = 0,9416$) коррелирует с потреблением сыра на душу населения (http://tylervigen.com/view_correlation?id=1099).

[79](#) Может быть полезно сослаться на перспективный баг-репортер, описанный в [Tatham, 1999], который способен помочь в составлении качественных отчетов об ошибках.

[80](#) В переводе Ф.И. Тютчева.

[81](#) Но будьте осторожны — случайная корреляция может направить вас по ложному следу.

[82](#) Во многих отношениях это похоже на метод «пяти почему» [Ohno, 1988], представленный Таichi Оно для понимания того, в чем корень проблем производственных ошибок.

[83](#) RE2 — новая версия библиотеки поддержки регулярных выражений, предложенная Google в 2010 году и использующая модель детерминированных конечных автоматов (DFA). По сравнению с RE2, более традиционная библиотека PCRE на основе недетерминированных конечных автоматов (NFA) допускает экспоненциальный рост затрат. RE2 доступна по ссылке <https://github.com/google/re2>.

[84](#) [Allspaw, 2015] отмечает, что это часто используемая эвристика в разрешении проблем, связанных с перебоями электропитания.

[85](#) Использование для заметок документа с совместным доступом или чата в реальном времени позволяет иметь временные метки для всех выполненных действий, что полезно для написания отчета. Одновременно вы также делитесь информацией с остальными, что ускоряет их работу и не мешает работать вам.

[86](#) См. [Meadows, 2008] о подходах к анализу систем, а также [Cook, 2010] и [Dekker, 2014] об ограниченности возможностей найти единственную причину ошибок вместо

исследования системы на предмет комплексно действующих факторов.

[87](https://cloud.google.com/appengine) См. <https://cloud.google.com/appengine>.

[88](#) Для облегчения понимания мы сократили и упростили описание этого примера.

[89](#) Хотя запуск продукта с неидентифицированным дефектом не лучшее решение, на практике часто бывает нецелесообразно устранять абсолютно все дефекты. Вместо этого иногда приходится довольствоваться «лучшим из возможного» и по возможности минимизировать риски, руководствуясь здравым смыслом.

[90](#) Поиск в хранилище данных для ускорения может применять индексы, но часто используемая реализация с кэшированием в памяти проста и удобна для просмотра по списку всех хранящихся так объектов. Однако выигрыш от линейного времени обработки списка незначителен при немногочисленных объектах, а с увеличением их количества может сильно возрасти задержка.

[91](#) Еще важнее влияние такого решения на логику программы: постоянно храня в памяти копию конфигурационных данных, программа не видит изменений их оригинала в хранилище или в файлах вплоть до явного повторного считывания. Хорошо это или плохо — решать разработчикам. — *Примеч. пер.*

[92](#) Описанная ситуация, скорее всего, разрешилась бы быстрее или не возникла бы вовсе, если бы разработчики не злоупотребляли «экстенсивным» подходом к разработке и больше следили за потреблением ресурсов. В частности, выявлять перерасход памяти для хранения структур данных можно и нужно. — *Примеч. пер.*

13. Реагирование в критических ситуациях

Автор — Кори Адам Бэй

Под редакцией Дайаны Бэйтс

Жизнь так устроена, что все когда-нибудь ломается.

Как бы ни были высоки ставки и как бы ни была велика организация, важная характеристика, от которой зависит, как долго проживет организация и что выделяет ее среди других, — это то, как ее сотрудники реагируют на критические ситуации. Лишь некоторые из нас от природы хорошо ориентируются в таких ситуациях. Поэтому правильный подход состоит в заблаговременной подготовке к ним и периодических тренировках. Налаживание и поддержание системы тренировок и контроля требует, помимо добросовестного отношения со стороны персонала, также и соответствующего управления. Все эти составляющие важны для создания благоприятной атмосферы, в которой команды смогут вкладывать свою энергию, деньги и, возможно, сверхурочное время работы, чтобы быть уверенными в том, что в критической ситуации системы, процессы и люди будут вести себя эффективно.

Заметьте, что особенности составления отчетов, которые позволяют почертнуть новые знания из инцидентов, потребовавших экстренного реагирования, обсуждаются в главе 15. Эта же глава более конкретно рассматривает сами такие инциденты.

Что делать, когда система сломалась

Прежде всего — не паникуйте! Вы не одни, и небо не падает вам на голову. Вы профессионал и тренировались для таких

ситуаций. Обычно никому не грозит прямая опасность, на грани смерти только несчастные электроны. В худшем случае не будет работать половина Интернета. Так что сделайте глубокий вдох... и продолжайте.

Если вам кажется, что вы не справитесь, привлекайте к решению больше людей. Иногда может потребоваться привлечь целую компанию. Если в вашей компании есть установленные процедуры реагирования на инциденты (см. главу 14), удостоверьтесь, что вы знакомы с ними, и действуйте согласно инструкции.

Авария, вызванная тестированием

В Google взяли на вооружение упреждающий подход к тестированию поведения в аварийных и иных критических ситуациях (см. [Krishan, 2012]). SR-инженеры «ломают» наши системы, наблюдают их крах и вносят необходимые изменения для повышения устойчивости и предотвращения подобных отказов в будущем. Большинство таких «управляемых аварий» проходят в соответствии с планом, и системы демонстрируют поведение более или менее в рамках ожидаемого. Мы находим какие-то слабые места или неявные зависимости и документируем действия, необходимые для исправления выявленных недостатков. Однако иногда реальность расходится с нашими ожиданиями. Рассмотрим один пример теста, который вскрыл несколько неожиданных зависимостей.

Подробности

Нам необходимо было удалить скрытые зависимости от тестовой базы данных в одной из наших крупных распределенных баз данных, реализованной на MySQL. Планировалось заблокировать любой доступ всего к одной базе

из сотни. Того, что из этого получилось, не мог предположить никто.

Развитие ситуации

Через пару минут после начала теста многие зависимые сервисы сообщили, что как внешние, так и внутренние пользователи не могут получить доступ к ключевым системам. Некоторые системы были доступны частично или с перебоями.

Придя к выводу, что в этом виноват тест, SR-инженеры немедленно прервали его. Мы попытались откатить изменения, но не добились успеха. Не поддаваясь панике, мы немедленно приступили к поиску возможности восстановить нормальный доступ. Мы пошли проверенным путем: переключили доступ на копии и резервные ресурсы. Одновременно мы связались с ключевыми разработчиками, чтобы те устранили дефекты в библиотеке, обеспечивающей взаимодействие приложений с БД.

Доступ был полностью восстановлен в течение часа, и все сервисы смогли опять подключиться. Столь масштабные последствия теста мотивировали нас как на быструю и полную корректировку всех библиотек, так и на включение в планы периодического повторного тестирования, чтобы предотвратить повторение подобной ситуации.

Подведение итогов

Что оказалось хорошо. Зависимые сервисы, которые были затронуты этим инцидентом, мгновенно распространили проблему в масштабах всей компании. Мы правильно предположили, что наш эксперимент вышел из-под контроля, и немедленно его прекратили. Всего за час мы полностью восстановили доступ. Некоторые команды избрали другой путь

и перенастроили свои системы, исключив использование тестовой базы. Эти параллельные усилия помогли восстановить сервис так быстро, как это было возможно. Затем мы удостоверились в том, что система функционирует нормально, и решили периодически проводить подобные тесты, чтобы гарантировать, что ситуация больше не повторится.

Чему это нас научило. Хотя тест был предварительно полностью рассмотрен и изучен, реальность показала, что нам не хватило знаний того, как зависимые системы взаимодействуют между собой в такой ситуации.

Мы не смогли следовать процессу реагирования на инциденты, который создали всего несколько недель назад и который не был окончательно внедрен. Этот процесс гарантировал бы, что все сервисы и все пользователи будут предупреждены о перебоях в работе. Чтобы избежать подобного развития событий в будущем, команда SRE постоянно совершенствует и тестирует наши инструменты и процессы, ответственные за реагирование на инциденты, вдобавок удостоверяясь в том, что эти нововведения понятны всем заинтересованным сторонам.

Поскольку процедуры отката не были протестированы в тестовой среде, они оказались неработоспособны, из-за чего восстановление затянулось. Теперь мы каждый раз тестируем системы отката перед началом крупномасштабного тестирования.

Авария, вызванная изменениями конфигурации

Как вы уже можете себе представить, системы Google имеют множество конфигураций, причем сложных конфигураций, которые мы постоянно изменяем и дорабатываем. Для

предотвращения поломки наших систем мы проводим множество тестов на этих конфигурациях, чтобы удостовериться в том, что они не приведут к неожиданным и нежелательным результатам. Однако масштаб и сложность инфраструктуры Google практически не позволяет предвидеть каждую зависимость или каждое взаимодействие. Иногда изменения в конфигурациях не идут по плану. И вот пример такого случая.

Подробности

В пятницу было выполнено глобальное изменение конфигурации внутренней подсистемы, защищающей наши сервисы от перегрузки и некорректных обращений. Она взаимодействует со всеми нашими внешними системами, в которых изменение вызвало повторяющиеся отказы, и весь парк систем был парализован практически одновременно. Поскольку внутренняя инфраструктура Google зависит от собственных сервисов, множество внутренних приложений тоже стали недоступны.

Развитие ситуации

Через несколько секунд начали поступать сообщения мониторинга, указывающие, что многие сайты перестали работать. Некоторые дежурные инженеры независимо друг от друга восприняли это как отказ корпоративной сети и перешли в специальные защищенные комнаты («комнаты паники») с резервным доступом к системам, находящимся в «промышленной» эксплуатации. К ним присоединились и другие инженеры, безуспешно пытавшиеся получить доступ к своим системам.

Через пять минут после обновления конфигурации ответственный за него инженер был оповещен о перебоях внутрикорпоративного доступа, но еще не знал о глобальном характере проблемы, и запустил повторное изменение, отменяющее предыдущее. С этого момента сервисы начали восстанавливаться.

Через десять минут после первого обновления дежурные инженеры объявили об инциденте и продолжили действовать согласно процедурам реагирования. Они начали уведомлять остальную компанию о возникшей ситуации. Инженер, который обновил конфигурацию, информировал дежурных о том, что проблема, скорее всего, была в обновлении, которое он откатил назад. Тем не менее, у некоторых сервисов сбои и нарушения конфигурации, спровоцированные первоначальным изменением и не позволяющие полностью восстановить работоспособность, продолжались до часа.

Подведение итогов

Что оказалось хорошо. В данном случае сработало несколько факторов, которые предотвратили усугубление ситуации до долговременного отключения множества внутренних систем Google.

Начнем с того, что в процессе мониторинга проблема была почти моментально обнаружена и система уведомила нас о ней. Однако стоит заметить, что в этом случае наша система мониторинга не была идеальной: уведомления «вылетали» многократно, непрерывно заваливая дежурных инженеров и забивая обычные и экстренные каналы связи.

Как только проблема была найдена, процедура разрешения инцидента прошла в целом как полагается и оповещение обо всех предпринимаемых действиях и событиях были

регулярными и качественными. Наши средства коммуникаций держали всех на связи, даже когда более сложное программное обеспечение отказалось. Этот опыт напомнил нам о том, зачем служба SRE продолжает использовать высоконадежные, не перегруженные функционалом резервные средства коммуникаций, к которым они регулярно прибегают.

В дополнение у Google есть инструменты, доступные из командной строки, и средства резервного доступа, которые позволили выполнять обновления и откатывать изменения, даже когда другие интерфейсы оказались недоступны. Они отлично сработали в критический момент, напоминая, что инженеры должны лучше знать и чаще тестировать инструменты, с которыми они работают.

В инфраструктуре Google был добавлен еще один уровень защиты: в системах, затрагиваемых изменениями, ограничивается быстрота доставки полных обновлений для очередных клиентов. Такое поведение могло бы остановить цепную реакцию отказов и предотвратить полное отключение, позволяя обрабатывать хотя бы несколько запросов между отказами. Наконец, не стоит сбрасывать со счетов элемент удачи в быстром разрешении этого инцидента: так случилось, что инженер, занимавшийся обновлением, был на связи в реальном времени — высокий уровень ответственности, чего обычно не бывает во время подобных изменений. Инженер заметил большое количество жалоб об отказах в доступе сразу же после того, как выполнил обновление, и практически сразу же откатил изменения. Если бы процесс отката задержался, кто знает, сколько бы продлились перебои и насколько бы усложнилась ликвидация их последствий.

Чему это нас научило. Предыдущие обновления, уже прошедшие тщательную проверку, не приводили к такой проблеме, так как в них не было комбинации одной редкой и

специфичной опции конфигурации и нового функционала. Поэтому то изменение, которое вызвало аварию, не рассматривалось как рискованное и, следовательно, проходило менее строгий процесс проверки. Когда изменение применили ко всей системе, оно задействовало непротестированную связку опции конфигурации и функционала, что и спровоцировало отказ.

По иронии судьбы, усовершенствование средств проверки и автоматизации этого процесса планировалось как первоочередная задача на следующий квартал. Случившееся немедленно повысило ее приоритет и укрепило инженеров в решении о необходимости полного тестирования независимо от ожидаемой степени рискованности обновления.

Как и можно было ожидать, оповещения во время инцидента были голосовыми, потому что за пару минут практически все ушло в офлайн. Это нарушило работу дежурных инженеров и усложнило общение между ними, тем самым усложнив ситуацию.

Google полагается на свои инструменты. Но большинство инструментов, которые обычно используются в таких ситуациях, оказались подвержены цепной реакции отказов. Если бы перебой в работе сервисов затянулся, отладка стала бы намного сложнее.

Авария, вызванная процессом

Мы вложили немало времени и сил в автоматизацию управления нашим парком машин. Просто невероятно, сколько задач можно запускать, останавливать или перенастраивать на всех машинах с минимальными усилиями! Но если что-нибудь идет не по плану, такая эффективность автоматизации может обернуться угрозой.

В следующем примере быстрота исполнения показала себя не с лучшей стороны.

Подробности

В ходе рутинного тестирования средств автоматизации на сервер, готовящийся к выводу из эксплуатации, были отправлены два последовательных запроса на отключение. На втором из них коварная ошибка в процедурах автоматизации отправила все машины аналогичных серверов по всему миру в очередь на выполнение Diskerase (полная очистка жестких дисков: см. врезку «Автоматизация: появление сбоев при масштабировании» в конце главы 7).

Развитие ситуации

Сразу после запуска второго запроса на отключение дежурные инженеры получили оповещение об отключении небольшого сервера — того, который и планировалось отключить. Исследование показало, что машины поставлены в очередь Diskerase, поэтому, следуя обычной процедуре, дежурные инженеры убрали весь трафик данной площадки. Это было сделано для того, чтобы избежать отказов в обслуживании, поскольку у этих машин содержимое дисков было удалено и они не могли обрабатывать запросы. Трафик был перенаправлен на площадки, которые могли должным образом обрабатывать запросы.

Но вскоре стали поступать сообщения от аналогичных серверов по всему миру. Предотвращая дальнейшие разрушения, инженеры отключили автоматизацию во всех командах. Вскоре также были прекращены или приостановлены дополнительная автоматизация и процедуры обслуживания серверов промышленной эксплуатации.

В течение часа весь трафик был разведен по другим площадкам. И хотя пользователи могли сталкиваться с увеличившимися задержками, их запросы выполнялись. Сбой был официально устранен.

Теперь началось самое сложное — восстановление. По части ссылок наблюдались заторы запросов, и сетевые инженеры предпринимали меры по снижению нагрузки в выявляемых узких местах. На одной из таких площадок был выбран сервер, которому первым из многих предстояло восстать из пепла. Через три часа после сбоя благодаря упорству нескольких инженеров система была пересобрана и смогла вернуться в строй, радостно принимая запросы пользователей.

Американские команды передали эстафету своим коллегам из Европы, и службы SRE набросали порядок восстановления с использованием «конвейерной» ручной переустановки. Команда была разделена на три части, каждая из которых отвечала за один свой конкретный этап. Через три дня большая часть серверов была запущена, но восстановление некоторых затянулось на месяц или два.

Подведение итогов

Что оказалось хорошо. Управление обратными прокси-серверами в больших и в малых системах организовано по-разному, поэтому более крупные системы не пострадали. Дежурные инженеры смогли быстро перенаправить трафик от маленьких серверов к большим, которые спроектированы так, что могут без проблем выдерживать полную нагрузку. Однако некоторые узлы были перегружены, что потребовало от сетевых инженеров искать пути обхода. Для того чтобы уменьшить негативный эффект для пользователей, дежурные инженеры выделили перегруженные сети как свою самую приоритетную задачу.

Процесс отключения для небольших систем работал правильно и эффективно. Менее чем за час было отключено и надежно «зачищено» множество таких систем.

Хотя отключение средств автоматизации остановило также и мониторинг этих небольших систем, дежурные инженеры смогли возобновить его из командной строки. Это помогло им оценить масштабы нанесенного ущерба.

Инженеры оперативно следовали протоколу реагирования на инциденты, который к тому моменту был намного яснее, чем во времена первого сбоя, описанного в текущей главе. Взаимодействие и сотрудничество между командами и во всей компании в целом было превосходным — реальное свидетельство того, насколько эффективными были тренинги и вся программа по управлению в критических ситуациях. Никто не оставался в стороне, внося вклад в общее дело.

Чему это нас научило. В основе возникшей проблемы лежало то, что ответственный за отключение систем сервер автоматизации недостаточно проверял отправляемые команды на «стерильность». Попытавшись выполнить процедуру повторно после неудавшегося первого отключения, он получил пустой ответ от серверной стойки. Вместо того чтобы отфильтровать этот ответ, он передал в базу данных серверов запрос на удаление содержимого дисков с пустым фильтром, то есть всех машин, содержащихся в базе. Да, иногда «ничего» означает «все». База данных скомпилировала запрос, и процесс разрушения закрутился на всех серверах так быстро, как это было возможно.

Переустановка была медленной и ненадежной. Это по большей части было вызвано использованием протокола передачи файлов TFTP (Trivial File Transfer Protocol) с низшим показателем качества обслуживания (QoS) для удаленных площадок. BIOS этих машин не могла адекватно справиться с

ошибками передачи⁹³. В зависимости от установленных сетевых карт, BIOS или приостанавливалась работа, или уходила в бесконечный цикл перезагрузок. Раз за разом передача загрузочных файлов обрывалась, и это все сильнее нагружало установщики. Дежурные инженеры смогли решить проблему, слегка подняв приоритет для трафика перезагрузки систем и используя автоматизацию для перезапуска зависших машин.

Инфраструктура переустановки машин не смогла справиться с одновременной инсталляцией тысяч машин. Отчасти это вызвано тем, что процедуры восстановления позволяли запустить с одного рабочего места не более двух установок одновременно. Кроме того, в ходе восстановления использовались неверные параметры QoS для передачи файлов и плохо настроенные тайм-ауты. Это вынудило переустанавливать ядро системы даже там, где очистка еще не была выполнена и ядро оставалось работоспособно. В попытке исправить эту ситуацию дежурные инженеры связались с командами, которые были ответственны за эту инфраструктуру и могли бы быстро перенастроить ее на поддержку такой необычной нагрузки.

У всех проблем есть решение

Время и опыт показывают, что системы могут не просто ломаться — они могут ломаться так, как никто и представить себе не мог. Один из самых важных выученных нами уроков — это то, что решение есть всегда, даже если оно неочевидно, особенно для человека, у которого пейджер разрывается от вызовов. Если вы не можете придумать решение, забросьте свою сеть дальше. Подключите больше коллег, ищите помощи, делайте то, что должны, но делайте это быстро. Главное — быстро взять проблему под контроль. Зачастую больше всего

знает о ситуации тот, кто как-то причастен к ее возникновению. Обратитесь к этому человеку.

Как только проблема будет решена, очень важно не забыть выделить время для наведения порядка, описания инцидента и...

Учитесь на прошлом опыте. И не повторяйте его

Храните историю перебоев в работе

Нет лучшего способа научиться, чем задокументировать то, что ранее ломалось. Это позволяет учиться на ошибках других. Будьте внимательны и честны, но самое главное — задавайте сложные вопросы. Ищите новые способы предотвратить повторный сбой не только тактически, но и стратегически. Убеждайтесь в том, что все члены команды могут узнать из отчетов все то, что узнали вы.

Оставляйте указания по действиям в такой ситуации для себя и других ответственных лиц. Это предотвратит будущие перебои, которые будут схожи с вашим случаем и будут вызваны теми же причинами, что и задокументированный. Наличие достаточно полного списка уже изученных сбоев позволит увидеть, что можно сделать, чтобы предотвратить такие случаи в будущем.

Задавайте себе сложные и даже неправдоподобные вопросы: «А что если?..»

Нет лучшей проверки, чем реальность. Задавайте себе сложные, не имеющие готового ответа вопросы.

А что если в здании пропадет электропитание? Что если стойка с сетевым оборудованием окажется в двух футах под водой? А что если главный дата-центр внезапно выключится? А что если кто-то атакует ваш веб-сервер? Что тогда делать? Кому звонить? Кто за это заплатит? У вас есть план действий? Знаете ли вы, что делать? Знаете ли вы, как отреагируют системы? Сможете ли вы минимизировать ущерб, если это произойдет сейчас? Смогут ли другие люди, сидящие рядом, сделать то же самое?

Поощряйте упреждающее тестирование

Когда дело дошло до аварии, теория и практика — это две большие разницы. Пока ваша система не сломается по-настоящему, вы до конца не узнаете, как поведут себя эта система, зависящие от нее системы и пользователи.

Не надейтесь на то, что вы предполагаете, или на то, что не смогли или не стали тестировать. Какое окончание недели вы предпочтете — ошибку в два часа ночи в субботу, когда большая часть компании еще не вернулась с тимбилдинга в Шварцвальде (горный массив в Германии. — Примеч. пер.), или наблюдение за безукоризненным выполнением заключительных тестов, которые они тщательно гоняли всю эту неделю?

Итоги главы

Мы рассмотрели три разных случая, когда компоненты нашей системы выходили из строя. Несмотря на то что причины всех трех происшествий были различны — упреждающее тестирование, изменение конфигурации, отказ оборудования, — дальнейшее развитие событий имело много общего. Ответственные не запаниковали. Они запросили помошь,

когда посчитали это необходимым. Они были научены опытом предыдущих инцидентов. Впоследствии они построили свои системы для лучшего реагирования на такие виды инцидентов. Каждый раз новые найденные проблемы были задокументированы. Потом эти сведения помогали другим командам научиться устранять неполадки и защищать свои системы от схожих сбоев. Выполнялось упреждающее тестирование систем, которое гарантировало, что внесенные изменения решают проблемы, и показывало новые узкие места, до того как они станут проблемой.

Циклы эволюции наших систем продолжаются — с каждым сбоем или новым результатом тестирования постепенно улучшаются и процессы, и система в целом. И хотя рассматриваемые в этой главе примеры относятся к Google, те же подходы к разрешению критических ситуаций могут быть применены в любых организациях любого масштаба.

[93](#) BIOS (Basic Input/Output System) — базовая система ввода/вывода. Встроенное программное обеспечение компьютера, управляющее им на низком уровне и обеспечивающее ввод/вывод до того, как запустится операционная система.

14. Управление в критических ситуациях

Автор — Эндрю Стриблхилл [94](#)

Под редакцией Кавиты Джулиани

Эффективное управление во время аварии является ключом к минимизации причиненного ущерба и скорейшему восстановлению нормальной работы. Если вы не отрепетируете собственный ответ на потенциальные инциденты, теоретические механизмы управления инцидентами в реальных ситуациях могут пойти насмарку.

В этой главе анализируется пример инцидента, который выходит из-под контроля из-за бессистемного управления, описывается корректный подход к инцидентам, а также рассматривается итог того же инцидента в случае с применением отлаженных методов управления.

Неуправляемый инцидент

Поставьте себя на место Мэри, дежурного программиста фирмы. Пятница, на часах два часа пополудни, и тут пейджер внезапно взрывается. Мониторинг методом черного ящика говорит, что ваш сервис прекратил обработку какого бы то ни было трафика во *всем* дата-центре. Вздохнув, вы отставляете чашку с кофе и принимаетесь решать проблему. Через несколько минут приходит оповещение об остановке второго дата-центра. Затем отказывает уже и третий из ваших пяти дата-центров. Ситуация усугубляется тем, что объем трафика превосходит возможности оставшихся дата-центров, и это приводит к перегрузке. Когда вы это обнаруживаете, сервис уже оказывается перегружен и больше не способен отвечать на запросы.

Вы напряженно всматриваетесь в файлы журналов, и кажется, что это продолжается уже целую вечность. В них тысячи строк, и, судя по ним, ошибка кроется в одном из недавно обновленных модулей, поэтому вы решаете вернуть серверы к предыдущей устойчивой версии. Когда выясняется, что откат не помогает, вы звоните Жозефине, которая написала большую часть кода агонизирующего в данный момент сервиса. Напомнив вам, что в ее часовом поясе сейчас полчетвертого утра, она все же сонным голосом соглашается авторизоваться и взглянуть. Ваши коллеги Сабрина и Робин начинают копаться в проблеме с собственных терминалов. «Мы просто смотрим», — говорят они.

Тем временем один из служащих позвонил вашему боссу и рассерженно требует ответа на вопрос, почему ему не сообщили о «полной катастрофе этого жизненно необходимого для работы сервиса». Вице-президенты порознь донимают вас по поводу срока устранения проблемы, постоянно спрашивая: «Как это могло случиться?» Вы бы рады были отнести к их вопросам со всем вниманием, но это потребовало бы когнитивных усилий, которые вы бережете для работы. Они вспоминают свой прежний опыт разработки и высказывают бесполезные советы вроде: «Увеличь размер этой страницы!» — которые тем не менее нельзя оставить без внимания.

Проходит время, и два оставшихся центра обработки данных отказывают окончательно. Не ставя вас в известность, сонная Жозефина звонит Малколму. У него неожиданно возникает блестящая идея: что-то насчет привязки потоков к процессорам. Он убежден, что сможет оптимизировать процессы на оставшихся серверах (в промышленной среде!), если поставит на них вот это простое изменение, — и делает это. В течение пары секунд серверы перезапускаются, принимая изменения. А затем умирают.

Анатомия неуправляемого инцидента

Заметьте, что в вышеизложенном сценарии каждый исполнял свои обязанности так, как понимал их. Почему же все пошло так плохо? Этот инцидент вышел из-под контроля из-за нескольких типичных неблагоприятных факторов.

Излишняя сосредоточенность на технической стороне проблемы

Мы стремимся набирать таких людей, как Мэри, из-за их технического мастерства. Поэтому неудивительно, что она была занята внесением функциональных изменений в систему, мужественно пытаясь решить проблему. Мэри была не в том положении, чтобы более масштабно думать о минимизации последствий аварии, поскольку ее непосредственная техническая задача оказалась непосильной.

Низкий уровень взаимодействия

По той же причине Мэри была слишком занята, чтобы поддерживать четкий обмен данными. Никто не знал, какие действия предпринимают его коллеги. Руководство было в ярости, клиенты дезориентированы, а прочие инженеры-программисты, которые могли бы подставить плечо в поиске и исправлении ошибки, не были задействованы в полной мере.

«Вольный стрелок»

Малколм внес изменения в систему из лучших побуждений. Он, однако, не согласовал свои действия с коллегами, даже с Мэри, которая технически была ответственна за устранение неполадок. Его действия сделали эту и без того плохую ситуацию еще хуже.

Элементы процесса управления в критических ситуациях

Навыки и методы управления в критических ситуациях существуют для того, чтобы направлять силы мотивированных индивидуумов. Используемая в Google система управления в критических ситуациях⁹⁵ основана на системе управления инцидентами (Incident Command System, ICS) — системе концепций, рекомендаций и стандартов, отличающейся прозрачностью и масштабируемостью.

Правильно спроектированный процесс управления инцидентами имеет следующие особенности.

Рекурсивное разделение обязанностей

Необходимо убедиться, что каждый участвующий в разрешении инцидента знает свою роль и не заходит на чужую территорию. Это может показаться парадоксальным, но четкое разделение обязанностей предоставляет отдельным лицам больше автономии, поскольку не приходится угадывать поведение коллег и подстраиваться под них.

Если нагрузка на отдельного члена команды становится чрезмерной, ему нужно запросить у ответственного за планирование больше исполнителей. Те, в свою очередь, тоже могут передать работу следующим сотрудникам, и это может повлечь за собой появление субинцидентов. Или же выполняющий функции «лидера инцидента» может передать различные компоненты системы коллегам, которые затем сообщают лидеру результаты исследования.

Есть несколько вполне обособленных ролей, которые следует распределить между лицами, участвующими в работе над инцидентом.

- *Управление инцидентом.* «Начальник штаба» контролирует общее состояние инцидента, формирует команду для работы с инцидентом и распределяет обязанности в соответствии с текущими потребностями и с приоритетом задач. На практике он же исполняет и все нераспределенные функции. При необходимости он может устранять препятствия, которые мешают оперативной группе работать наиболее эффективно.
- *Оперативная группа.* Лидер оперативной группы взаимодействует с «начальником штаба» и работает над разрешением инцидента, применяя имеющиеся средства для выполнения поставленных задач. Только оперативная группа может обладать правом на внесение изменений в систему на протяжении всего инцидента.
- *Информирование.* Лицо всей команды. В его обязанности непременно входит держать в курсе происходящего (обычно посредством электронной почты) всю команду и иных заинтересованных лиц, а также, например, поддерживать актуальность и достоверность документации об инциденте.
- *Планирование.* Цель планирования — оказывать оперативной группе поддержку, занимаясь более длительными делами, такими как регистрация найденных ошибок, заказ обеда, планирование и организация передачи задач между исполнителями, а также отслеживание и фиксацию аномалий в системе, чтобы ее можно было вернуть в исходное состояние после разрешения инцидента.

Выделенный центр управления

Заинтересованные стороны должны знать, где они могут общаться с «начальником штаба». Во многих ситуациях

уместно компактное расположение всех членов команды в месте, обозначенном как центр управления. Другие команды могут предпочесть работать за собственными столами, отслеживая обновления по инциденту посредством электронной почты или IRC (Internet Relay Chat).

По опыту корпорации Google, IRC приносит огромную пользу в реагировании на инциденты. IRC работает очень надежно и может быть использован для протоколирования всех переговоров в ходе работы, чем оказывает неоценимую помощь в удержании в памяти всех подробностей о происходящих в системе изменениях. Мы написали боты, которые регистрируют относящийся к инциденту трафик (что пригодится для последующего изучения в ходе «разбора полетов») либо фиксируют такие события, как оповещения в канале. IRC — это также удобное средство общения, способное координировать территориально распределенные команды.

Обновляемый документ о состоянии инцидента

Одна из наиболее важных обязанностей «начальника штаба» — ведение обновляемого документа об инциденте и обеспечение его актуальности. Он может существовать в виде редактируемой веб-страницы, но лучше всего — в виде документа с коллективным доступом, редактируемого несколькими людьми одновременно. Большинство наших команд используют Google Docs, хотя сама SRE-команда Google Docs пользуется сервисом Google Sites: в конце концов, если система зависит от программного средства, которое вы и пытаетесь исправить в рамках данного инцидента, то вряд ли она в это время будет работоспособна.

В приложении В вы найдете образец документа об инциденте. Обновляемый документ может выглядеть неряшливо, но он должен быть функциональным.

Использование шаблонов должно облегчить создание подобной документации, а запись наиболее важной информации в самом верху документа делает его более практическим. Сохраните его для последующего «разбора полетов» и, если необходимо, для метаанализа.

Четкая и оперативная передача полномочий

Принципиально важно, чтобы функции «начальника штаба» были полностью и однозначно переданы в конце рабочего дня. Если вы передаете управление кому-то с другим местоположением, сообщите ему об этом посредством телефонного или видеозвонка. Как только новый начштаба полностью введен в курс дел, покидающий пост должен четко обозначить передачу, явно говоря: «Сейчас ты управляешь инцидентом, понятно?» — и оставаться на связи, пока не получит такого же четкого подтверждения того, что пост принят. Передача управления должна быть доведена до сведения других лиц, работающих над инцидентом, чтобы в любое время всем было известно, кто возглавляет работы по управлению инцидентом.

Управляемый инцидент

Посмотрим, чем обернулась бы та же ситуация, если бы к ней применили принципы управления инцидентами.

На часах два пополудни, и Мэри пьет третью чашку кофе за день. Ее застает врасплох резкий сигнал пейджера, и она залпом допивает напиток. Возникла проблема: центр обработки данных прекратил обслуживание трафика. Мэри начинает разбираться. Вскоре приходит новое оповещение: вышел из строя второй из пяти центров обработки данных. Таким образом, ситуация развивается очень быстро, и это

говорит, что пора воспользоваться системой управления инцидентами.

Она подтягивает Сабрину. «Ты сможешь принять управление?» Кивнув в знак согласия, Сабрина получает от Мэри краткое изложение нынешнего положения дел. Подробности она заносит в письмо, которое рассыпает по заранее подготовленному списку контактов. Сабрина понимает, что пока не может оценить последствия аварии, и просит содействия Мэри. Та отвечает: «Пользователей уже должно было затронуть, но будем надеяться, что не потеряем и третий центр». Сабрина записывает ответ Мэри в обновляемый документ инцидента.

Когда приходит третье оповещение, Сабрина видит его в отдельной ветке IRC и дублирует информацию в цепочке электронных писем. Письма держат вице-президентов в курсе ситуации без заваливания лишними подробностями. Сабрина просит уполномоченного по связям с общественностью начать составление обращений к пользователям, а затем обращается к Мэри, чтобы узнать, не следует ли им связаться с дежурным разработчиком (в данный момент это Жозефина). Получив утвердительный ответ от Мэри, Сабрина подключает Жозефину к работе.

К тому времени как Жозефина авторизуется в системе, уже вызвался помочь и Робин. Сабрина напоминает Робину и Жозефине, что они должны уделять первоочередное внимание любым задачам, которые поставит им Мэри, и держать Мэри в курсе любых дополнительных действий, которые они предпринимают. Робин и Жозефина быстро знакомятся с текущей ситуацией, прочитав документ инцидента.

Тем временем Мэри уже попробовала вернуть предыдущую версию сервиса (бинарные файлы) и убедилась, что это не помогло. Она быстро пожаловалась на это Робину, и тот

сообщил в IRC о неудаче этой попытки исправления. Сабрина скопировала эту информацию в обновляемый документ о состоянии инцидента.

В пять часов вечера Сабрина начинает искать персонал, который взял бы на себя работу над инцидентом, поскольку она и ее коллеги собираются идти домой. Она обновляет документ об инциденте. В 5:45 проходит короткая телефонная конференция, в результате все владеют ситуацией. В шесть вечера члены команды передают обязанности коллегам в следующем офисе.

Следующим утром Мэри вернулась на работу и узнала, что ее коллеги по другую сторону океана, приняв эстафету, локализовали проблему, позаботились о минимизации ущерба, закрыли инцидент и начали работать над отчетами. Проблема решена. Мэри заваривает свежий кофе и принимается за планирование структурных улучшений, чтобы аналогичные проблемы не побеспокоили команду снова⁹⁶.

Когда следует сообщать об инциденте

Лучше объявить об инциденте как можно раньше, найти простое решение и закрыть инцидент, чем часами развертывать структуру управления инцидентами в условиях нарастания проблемы. Определите четкие условия для объявления инцидента. Моя команда руководствуется следующими общими правилами. Если выполняется любое из этих условий, событие является инцидентом.

- Вам необходимо привлекать вторую команду, чтобы решить проблему.
- Перебой в работе заметили клиенты.

- Проблема не решена даже после нескольких часов сосредоточенной работы.

Компетентность в управлении инцидентами быстро утрачивается, если не находит применения постоянно. Но как программистам поддерживать на должном уровне свои навыки по управлению инцидентами — работать с большим их количеством? К счастью, принципы и методы управления инцидентами можно применять к другим функциональным изменениям, которые требуют задействовать несколько команд, особенно в разных часовых поясах. Если вы используете эти методы как привычную часть процедур управления изменениями в системах, вы легко примените их и при возникновении настоящего инцидента. Если ваша организация проводит тестирование аварийного восстановления (если нет, то начните, это весело: см. [Krishan, 2012]), то управление инцидентами должно быть частью этого процесса. Мы часто проводим ролевую игру — реагирование на критическую ситуацию, которая уже была разрешена (возможно, коллегами на другой площадке), чтобы лучше освоить управление инцидентами.

Подведем итоги

Итак, предварительно выработав стратегию управления инцидентами, имея структурированный, гибко масштабируемый план и регулярно применяя его на практике, мы смогли существенно сократить среднее время восстановления и снизили стрессовую нагрузку на персонал в критических ситуациях. Внедрение аналогичной стратегии могло бы быть полезно для любой организации, озабоченной обеспечением бесперебойной работы.

Практические рекомендации по управлению инцидентом

Расставляйте приоритеты. Остановите развитие аварии, возобновите работу сервиса и сохраните данные для последующего анализа истинных причин аварии.

Готовьтесь заранее. Заранее разрабатывайте и документируйте ваши методы управления инцидентами, согласовывая их со всеми участниками.

Доверяйте. Предоставьте всем участникам работ полную самостоятельность, но в рамках отведенных им ролей.

Используйте самоанализ. Уделяйте внимание своему эмоциональному состоянию во время работы над инцидентом. Если вы начинаете впадать в панику или чувствуете перегруженность, запросите помошь.

Рассматривайте альтернативы. Время от времени взвешивайте варианты и анализируйте: целесообразно ли продолжать работу над текущей задачей или же стоит взяться за другую, более неотложную.

Практикуйтесь. Применяйте эти подходы постоянно, чтобы это вошло в привычку.

Проводите ротацию. В прошлый раз вы были начальником штаба? Тогда сейчас возьмите на себя другую роль. Стимулируйте всех участников команды ознакомиться с каждой ролью.

94 Более ранняя версия этой главы появилась в виде публикации в ;login: (апрель 2015 года, выпуск 40, № 2).

95 Подробнее см. <http://www.fema.gov/national-incident-management-system>.

96 Стоит отметить, что в обоих примерах в этой главе рассматриваются только первые шаги реагирования на возникшую аварию, а ее причины и действия по ликвидации последствий остаются за кадром. — *Примеч. пер.*

15. Культура постмортема: учимся на ошибках

Авторы — Джон Лунни и Сью Льюдер

Под редакцией Гэри О'Коннора

Ошибки — плата за знания.

Девин Кэрреуэй

Будучи SR-инженерами, мы работаем с очень большими, сложными, распределенными системами. Мы постоянно дополняем наши сервисы новыми функциями и добавляем новые системы. Перебои и внештатные ситуации неизбежны, учитывая масштаб и интенсивность этих изменений. Когда это происходит, мы устраним причину проблемы и сервис снова функционирует в нормальном режиме. И если бы у нас не было какого-либо формализованного процесса извлечения опыта из этих происшествий, они могли бы повторяться бесконечно. Если не делать из них выводы, то сбои будут становиться сложнее и даже порождать новые сбои, перегружая систему и ее операторов, вплоть до помех пользователям. Поэтому анализ завершившегося инцидента — «разбор полетов» — является важным инструментом SR-инженера.

Концепция «разбора полетов» и так называемого постмортема хорошо известна в индустрии высоких технологий [Allspaw, 2012]. Постмортем представляет собой письменный отчет об инциденте, его последствиях, действиях, предпринятых для его смягчения или устранения, его первопричине (или первопричинах), а также о последующих действиях для предотвращения его повторения. Эта глава описывает критерии для выбора момента проведения «разбора полетов» и составления постмортема, излагает практику организации этого процесса, а также советы по введению

соответствующей культуры на основании нашего опыта, полученного за годы работы.

Философия постмортема от Google

Главная цель написания постмортема — гарантировать документирование инцидента, тщательно изучить все имевшие место его причины и, что особенно важно, принять эффективные профилактические меры для уменьшения вероятности его повторения и/или возможных последствий. Подробный обзор методов анализа первопричин выходит за рамки этой главы (вместо нее см. [Rooney, 2004]); впрочем, по теме качества систем можно найти большое количество публикаций, практических рекомендаций и инструментов. Наши команды используют для анализа первопричин различные методы, выбирая наиболее подходящие для их сервисов. Проведение «разбора полетов» и составление постмортема ожидается после любого значительного нежелательного события. Это не наказание, а возможность научиться чему-то новому для всей компании. Стоимость этого процесса, однако, достаточно высока в плане затраченных усилий и времени, поэтому мы сами вольны решать, когда приступить к отчету. Команды обладают некоторой внутренней свободой действий, однако есть общие правила, когда постмортем необходим:

- обнаруженнное пользователями время простоя или ухудшение производительности ниже определенного порога;
- потеря данных любого типа;
- потребовалось вмешательство дежурного инженера (отмена и откат изменений, перенаправление трафика и т.п.);

- время разрешения ситуации превысило определенный лимит;
- ошибка системы мониторинга (обычно это подразумевает обнаружение инцидента пользователем или инженером).

Важно определить критерии для составления постмортема до возникновения инцидента, чтобы каждый понимал их. Помимо вышеуказанных объективных причин, любая из заинтересованных сторон может затребовать постмортем для какого-либо конкретного события.

Безобвинительный «разбор полетов» является краеугольным принципом для всей службы SRE. По-настоящему безобвинительный постмортем должен быть направлен на то, чтобы выявить причины возникновения инцидента, а не указать отдельным сотрудникам или командам на их ошибочные или нецелесообразные действия. Такой постмортем предполагает, что каждый, кто был вовлечен в инцидент, имел благие намерения и правильно распорядился той информацией, которой обладал. В атмосфере выискивания виноватых и обвинений отдельных лиц или команд за «неправильные» действия люди не станут рассказывать о проблемах просто из-за страха наказания.

Безобвинительная культура берет начало в области здравоохранения и авиации, где ценой ошибки порой может стать чья-то жизнь. Именно там культивировалась такая рабочая среда, где каждая ошибка рассматривается как возможность улучшить систему. Фокусируясь не на поиске виновных, а на расследовании системных причин того, почему отдельное лицо или команда получили неполную или некорректную информацию, можно применить эффективные профилактические меры. Нельзя «исправить» людей, зато

можно исправить системы и процессы, чтобы они помогали людям, проектирующим и обслуживающим сложные системы, принимать правильные решения.

Когда в системе происходит реальный сбой, постмортем не должен быть просто отпиской, которую вскоре забудут. Напротив, инженеры смотрят на постмортем как на возможность не только исправить недочет, но и в целом сделать системы Google более надежными. Безобвинительный постмортем призван не изливать досаду, тыкая в кого-то пальцем, а ясно указывать, где и как сервисы могут быть улучшены. Вот два примера.

- *Тыканье пальцем.* «Бэкенд-часть системы слишком сложная, ее надо переписать! Она выходила из строя каждую неделю в течение трех последних кварталов, и я уверен, что мы все устали исправлять все по мелочам. Серьезно, если меня вызовут еще хотя бы раз, я перепишу ее сам...»
- *Безобвинительный тон.* «Принятие решения о переписывании бэкенд-части системы в самом деле может предотвратить дальнейшее возникновение этих досадных ситуаций, а руководство по обслуживанию текущей версии довольно объемное и слишком сложное, чтобы его можно было полноценно изучить. Я уверен, что наши будущие дежурные инженеры скажут нам спасибо!»

Практическая рекомендация: избегайте обвинений и будьте конструктивными

Безобвинительный постмортем может показаться непростым для написания, поскольку сам формат постмортема

предписывает явно указывать действия, которые привели к инциденту. Устранение порицаний из постмортема придаст людям уверенности, чтобы безбоязненно обсуждать проблему. Важно также не заставлять отдельных сотрудников или команды слишком часто заниматься составлением постмортемов, превращая это в наказание. Атмосфера страха перед обвинением создает почву для замалчивания проблем и инцидентов, что ведет к возрастанию рисков для всей организации [Boysen, 2013].

Сотрудничайте и делитесь знаниями

Мы ценим сотрудничество, и «разбор полетов» здесь не исключение. Процесс составления постмортема включает сотрудничество и обмен знаниями на каждом этапе.

Мы пишем наши постмортемы в Google Doc с использованием внутрикорпоративных шаблонов (см. приложение Г). Вне зависимости от того, какое средство вы используете, обращайте внимание на следующие ключевые свойства.

- *Совместная работа в реальном времени.* Обеспечивает максимально быстрый сбор данных и идей. Незаменима на ранней стадии работы над постмортемом.
- *Открытая система комментариев и примечаний.* Упрощает поиск решений методом краудсорсинга (то есть «народной стройки». — Примеч. пер.) и расширяет их покрытие.
- *Уведомления по электронной почте.* Могут быть направлены участникам работы над документом или использованы для

того, чтобы держать в курсе остальных.

Подготовленный постмортем должен пройти одобрение и быть опубликован. На практике это означает, что команды создают заготовку постмортема собственными силами, а затем просят группу старших инженеров оценить его на предмет завершенности.

Критерии оценки могут включать следующее.

- Собраны ли ключевые данные по инциденту для последующего исследования?
- Является ли оценка влияния полной и завершенной?
- Была ли найдена основная причина?
- Целесообразны ли план действий и очередность последующих исправлений ошибок?
- Поставлены ли заинтересованные стороны в известность о результатах?

После предварительной оценки постмортем распространяют более широко — как правило, среди большой группы разработчиков или по внутреннему списку рассылки. Наша цель — распространить документы постмортемов среди максимально широкой аудитории, которая могла бы извлечь пользу из заложенных в них сведений или уроков. При этом Google руководствуется строжайшими правилами касательно доступа к любой информации, которая может идентифицировать пользователя⁹⁷, и даже такие внутренние

документы, как постмортемы, никогда не включают в себя подобные данные.

Практическая рекомендация: не оставляйте ни одного постмортема без оценки

Нерассмотренный постмортем все равно как несуществующий. Чтобы убедиться, что каждая заготовка прошла рассмотрение, мы поощряем регулярные семинары для разбора постмортемов. На этих встречах важно доводить до завершения все текущие дискуссии, фиксировать возникшие идеи и придавать окончательный вид документам.

Как только все участники будут удовлетворены документом и заключенным в нем планом действий, проект помещается в архив завершенных инцидентов команды или организации⁹⁸. К документам открыт коллективный доступ, что позволяет легко отыскать среди них нужный и изучать его.

Внедрение культуры постмортема

Говорить о культуре постмортемов куда проще, чем реализовать ее. Внедрение потребует планомерной работы и постоянного подкрепления. Мы подкрепляем культуру совместной работы над постмортемами, активно привлекая старших руководителей к участию в их рецензировании и совместному написанию. Руководство может содействовать развитию этой культуры, но безвинительные постмортемы в идеале предполагают самомотивацию инженеров. Для воспитания культуры постмортемов SR-инженеры организуют

превентивные мероприятия, которые призваны распространить полученные знания об архитектуре и инфраструктуре систем. В число подобных мероприятий, к примеру, входят следующие.

- *Постмортем месяца.* В ежемесячной рассылке по всей организации распространяется наиболее содержательный и качественный постмортем.
- *Группа Google+, посвященная постмортемам.* В этой группе распространяют и обсуждают примеры постмортемов — свои и других организаций, эффективные методы работы и комментарии по поводу постмортемов.
- *Книжные клубы по постмортемам.* Команды ведут регулярные собрания (читательские клубы), на которых за освежающими напитками рассматриваются особенно интересные или значимые постмортемы и ведется открытый диалог с вовлеченными и не вовлеченными в событие лицами, а также с новыми сотрудниками Google о том, что произошло и какие уроки можно извлечь из инцидента, какими будут его последствия. Инцидент может рассматриваться по прошествии нескольких месяцев или даже лет!
- «*Колесо неудачи*». Новые SR-инженеры часто проходят через упражнение «Колесо неудачи» (см. подраздел «Катастрофа: ролевая игра» на с. 484), в ходе которого разыгрываются события какого-либо постмортема с группой инженеров, исполняющих соответствующие роли. Это происходит в присутствии инженера, работавшего над изначальным инцидентом. Он помогает сделать обстановку максимально приближенной к реальности.

Одна из главных трудностей при внедрении культуры постмортемов в организации состоит в том, что некоторые могут оспорить их значимость, ссылаясь на затраты на подготовку. Следующие меры помогут преодолеть это затруднение.

- Включайте составление постмортемов в рабочий процесс постепенно. Пробный период, в течение которого будут подготовлены несколько успешно завершенных постмортемов, поможет доказать их ценность и помочь определить критерии, по которым устанавливается необходимость постмортема.
- Убедитесь, что эффективные постмортемы не остаются без внимания и вознаграждения: как публично, посредством упомянутых ранее социальных методов, так и в порядке индивидуального и внутрикомандного управления.
- Поощряйте осведомленность и участие высшего руководства. Даже сам Ларри Пейдж говорит об огромной значимости постмортемов!⁹⁹

Практическая рекомендация: явно вознаграждайте людей за правильные поступки

Основатели Google Ларри Пейдж и Сергей Брин проводят TGIF (из «Википедии»: TGIF – акроним к английской фразе Thank God it's Friday – «Слава Богу, сегодня пятница, празднование последнего рабочего/учебного дня недели». — Примеч. пер.), еженедельное собрание сотрудников в прямом эфире в нашей штаб-квартире в Маунтин-Вью, Калифорния,

которое транслируется в офисы Google по всему миру. Одно из TGIF 2014 года было посвящено теме «Искусство постмортема» и включало дискуссию SR-инженеров о событиях с серьезными последствиями. Один из них говорил о недавно загруженному им обновлении. И хотя оно было заранее тщательно протестировано, непредусмотренное взаимодействие программ на четыре минуты вывело из строя важнейший сервис. Инцидент продолжался всего четыре минуты, потому что инженер не растерялся и немедленно отменил изменения, предотвратив куда более обширные и далеко идущие последствия. Этот инженер не только немедленно получил два партнерских бонуса¹ в награду за быстрые и грамотные действия по разрешению инцидента, но и сорвал шквал аплодисментов от присутствующих на пятничном мероприятии, включая основателей компании и многотысячную аудиторию

сотрудников Google. В дополнение к таким публичным мероприятиям Google имеет ряд внутрикорпоративных соцсетей, которые стимулируют коллег отмечать хорошо составленные отчеты и отличное управление инцидентами. Это один из многих примеров того, как признание вклада в работу приходит со стороны всех сотрудников – от рядовых специалистов до директоров¹.

Практическая рекомендация: интересуйтесь мнениями об эффективности постмортема

В Google стремятся оперативно решать возникающие проблемы и делиться инновациями в рамках всей корпорации. Мы постоянно интересуемся у наших команд, насколько постмортемы способствуют решению их задач и что в них можно усовершенствовать. Мы задаем следующие вопросы. Помогают ли постмортемы в вашей работе? Не слишком ли изнурительным вам кажется написание постмортемов (см. главу 5)? Какие приемы и методы ваша команда могла бы порекомендовать остальным? Какие из инструментов вы хотели бы усовершенствовать? Результаты опроса позволяют находящимся «на передовой» SR-инженерам спросить об улучшениях, которые повышали бы эффективность постмортемов.

Помимо их влияния на управление инцидентами и последующими мерами по разрешению последствий, постмортемы вплетены в общую культуру производства в Google: сейчас уже считается нормой сопровождать исчерпывающим постмортемом каждый значимый инцидент.[100](#)

Итоги главы: непрерывные улучшения

Мы с уверенностью можем говорить, что благодаря постоянным усилиям по внедрению культуры постмортемов системы Google реже подвергаются сбоям и это делает их более эффективными и комфортными для пользователей. Наша

рабочая группа «Постмортемы в Google» служит примером нашей приверженности культуре безобвинительных постмортемов. Эта группа координирует действия по составлению постмортемов во всей компании: централизованно собирает шаблоны постмортемов, автоматизирует создание постмортемов на основании данных от программных инструментов, используемых при работе над инцидентом, и помогает автоматизировать извлечение данных из постмортемов, чтобы мы могли проанализировать имеющиеся тенденции. Мы смогли объединить лучшие методы работы над такими различными продуктами, как YouTube, Google Fiber, Gmail, Google Cloud, AdWords и Google Maps. Хотя эти разработки и различаются, все они используют постмортемы с одной и той же целью — извлекать уроки из наших самых сложных и неприятных событий.

Учитывая большое количество постмортемов и других отчетов, создаваемых каждый месяц во всей компании Google, все более актуальными становятся средства для их сбора и систематизации. Эти средства помогают нам определить наиболее часто затрагиваемые темы и наметить направления совершенствования наших продуктов. Чтобы облегчить понимание и автоматический анализ постмортемов, мы недавно ввели дополнительные поля метаданных в наш шаблон (см. приложение Г). Дальнейшая работа в этой области предполагает внедрение технологий машинного обучения, которое сможет заранее сообщить о слабых местах, облегчить исследование инцидентов в реальном времени, а также сократить количество дублирующихся инцидентов.

⁹⁷ См. <http://www.google.com/policies/privacy/>.

⁹⁸ Если вам нужно собственное хранилище, то Etsy выпустили Morgue — инструмент для работы с постмортемами.

[99](#) Программа партнерских бонусов в Google призвана отмечать и поощрять сотрудников, проявивших особое усердие; бонусы предполагают денежное вознаграждение.

[100](#) Более подробно этот инцидент рассмотрен в главе 13.

16. Контроль неисправностей

Автор — Гейб Крэбби

Под редакцией Лиза Кэри

Совершенствование в направлении повышения надежности предполагает, что мы, начав с некоторого «исходного уровня», можем постоянно наблюдать за прогрессом. Именно для этого мы используем Outalator — наше средство контроля сбоев и неисправностей. Outalator — это система, которая пассивно принимает все оповещения, отправляемые нашими системами слежения, и позволяет эти данные хранить, помечать, группировать и анализировать.

Систематическое обучение на опыте прошлых проблем — одна из основ эффективного управления сервисом. Постмортемы (см. главу 15) предоставляют детальную информацию о конкретных сбоях, но это лишь часть решения. Они создаются только для инцидентов с серьезными последствиями, а те происшествия, которые сами по себе наносят небольшой ущерб, зато происходят часто и имеют тенденцию повторяться, в поле зрения не попадают. Аналогично постмортемы чаще предоставляют информацию, полезную для совершенствования одного или нескольких сервисов, но могут упускать из виду те возможные изменения, эффект которых в конкретных случаях невелик либо они невыгодны по соотношению эффекта и затрат, но имеют большое «горизонтальное» влияние [101](#).

Мы также можем получить полезную информацию, задавая вопросы вроде: «Сколько оповещений получает каждая команда в свою дежурную смену?», «Каким было соотношение требующих и не требующих принятие мер оповещений за последний квартал?», и даже самый очевидный — «Какой из

сервисов, которыми занимается эта команда, потребовал от них усилий?».

Escalator

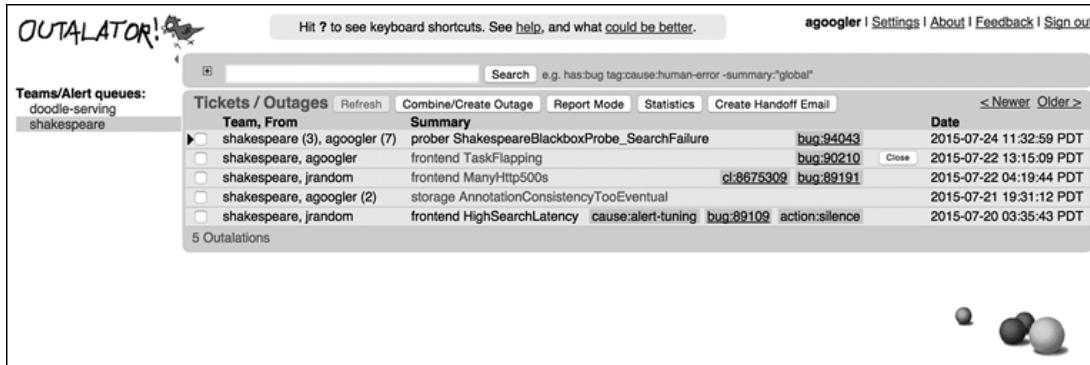
В Google все адресованные SR-инженерам оповещения проходят через единую реплицируемую систему, которая контролирует, подтверждено ли человеком их получение. Если в течение установленного времени подтверждение не получено, система обращается к следующему заданному адресату (или адресатам) — к примеру, от главного дежурного к замещающему. Эта система, названная The Escalator, изначально была разработана как максимально открытое средство, получающее копии электронных писем, отправляемых дежурным сотрудникам. Такой функционал позволил легко интегрировать Escalator в существующие рабочие потоки без необходимости менять поведение пользователя (или в тот период времени — поведение системы мониторинга).

Outalator

По примеру Escalator, когда мы добавляли полезные элементы к уже действующей инфраструктуре, мы создали систему, работающую не только с отдельными «восходящими» уведомлениями, но и с новым уровнем абстракции: сбоями (или дефектами).

Outalator позволяет пользователю просматривать список уведомлений с временным разделением, поступающих сразу из нескольких очередей, не требуя переключаться между ними вручную. На рис. 16.1 показано несколько очередей из списка Outalator. Такой подход удобен, поскольку первичный

получатель оповещений от многих сервисов — это одна и та же SRE-команда, и лишь при необходимости они перенаправляются другим получателям, обычно различным командам разработчиков.



Team, From	Summary	Date
shakespeare (3), agoogler (7)	prober ShakespeareBlackboxProbe_SearchFailure	bug:94043 2015-07-24 11:32:59 PDT
shakespeare, agoogler	frontend TaskFlapping	bug:90210 2015-07-22 13:15:09 PDT
shakespeare, jrandom	frontend ManyHttp500s	cl:8675309 bug:89191 2015-07-22 04:19:44 PDT
shakespeare, agoogler (2)	storage AnnotationConsistencyTooEventual	2015-07-21 19:31:12 PDT
shakespeare, jrandom	frontend HighSearchLatency cause:alert-tuning	bug:89109 action:silence 2015-07-20 03:35:43 PDT

5 Outalations

Рис. 16.1. Просмотр очередей Outalator

Outalator сохраняет копию первоначального сообщения и позволяет комментировать инциденты. Для удобства программа также молча принимает и сохраняет копию любого ответного электронного письма. Поскольку в цепочках писем есть менее информативные (к примеру, письмо, сгенерированное как «ответ всем», чтобы добавить больше адресов в список получателей копий), аннотации могут помечаться как важные. В таком случае интерфейс скрывает другие части сообщения, чтобы не создавать неразбериху. Все вместе это более информативное представление инцидента, чем просто поток писем, возможно разрозненных.

Outalator позволяет объединить несколько сообщений (оповещений) в единое целое (инцидент). Они могут относиться к одному и тому же инциденту, а могут быть не связанными между собой и не представляющими интереса событиями вроде привилегированного доступа к базе данных или даже просто ошибками системы мониторинга. Функция группировки, показанная на рис. 16.2, разгружает интерфейс и

позволяет изучать инциденты, а не просто отдельные оповещения.

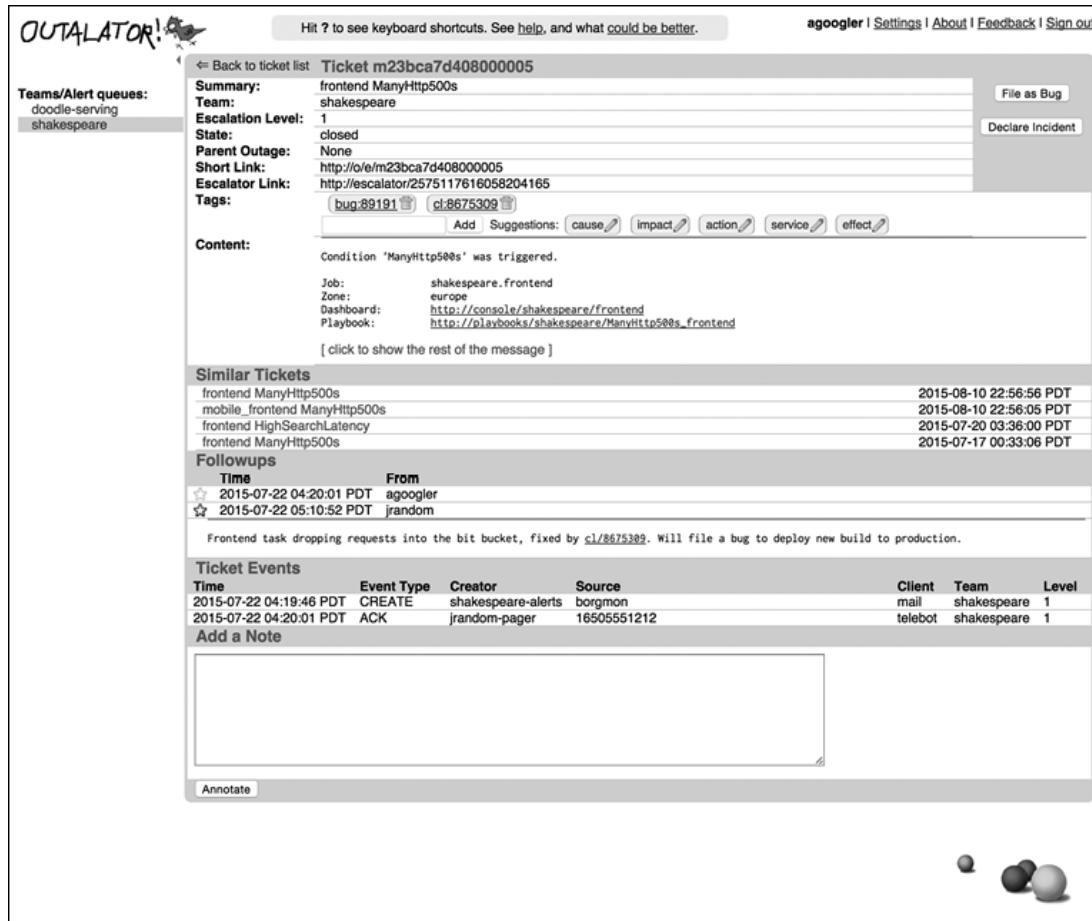


Рис. 16.2. Функция группировки

Если вы делаете свой Outalator

Многие организации используют такие системы обмена сообщениями, как Slack, Hipchat или даже IRC для внутреннего общения и/или обновления статуса информационных страниц. Все эти средства прекрасно подходят для включения в системы вроде Outalator.

Агрегирование

Одно и то же событие может порождать и часто порождает несколько оповещений. Например, сбои в работе сети становятся причинами превышения лимитов ожидания и недоступности сервисов и все соответствующие команды получат собственные оповещения, включая владельцев сервисов. А это значит, что в центре управления сетями одновременно зазвучат отдельные сирены. Впрочем, даже незначительные происшествия, затрагивающие отдельный сервис, могут породить несколько оповещений, если они сопровождаются неоднократными ошибками. Конечно, нужно стараться свести к минимуму количество оповещений, вызываемых одним и тем же событием, но полностью избежать их обычно не удается, поскольку необходимо обеспечивать баланс между ложноположительными и ложноотрицательными срабатываниями.

Возможность объединять несколько оповещений в единый инцидент весьма важна, чтобы справляться с таким дублированием. Письмо с текстом «Это то же самое, что и вон то, и все это признаки одного инцидента» подойдет для одного определенного оповещения: оно предотвратит ненужную повторную отладку или лишнюю панику. Однако сопровождать письмом каждое оповещение — непрактичное и плохо масштабируемое решение, если требуется устраниТЬ дублирующиеся оповещения внутри даже одной команды, не говоря уже о более продолжительных периодах времени или о взаимодействии нескольких групп.

Маркировка

Разумеется, не каждое оповещение о событии означает инцидент. Бывают ложные срабатывания, а также тестовые

события или письма, ошибочно отправленные человеком. Сам сервис Outalator не различает эти события, но позволяет делать универсальную маркировку — добавлять к сообщениям метаданные на всех уровнях. Маркеры (теги) записываются в произвольной форме, в виде отдельных слов. Двоеточия, впрочем, распознаются как семантические разделители, которые естественным образом дают возможность использовать иерархические пространства имен и применять некоторые средства автоматической обработки. Для поддержки пространств имен служат также рекомендованные префиксы маркеров, в первую очередь `cause` и `action`, но в целом используемый набор маркеров зависит от конкретной команды и обусловлен «историческими причинами». Например, для одних команд пометка `cause:network` может содержать достаточно большое количество информации, в то время как другие могут предпочесть более подробные формулировки, например `cause:network:switch` или `cause:network:cable`. Некоторые команды могут часто использовать метки типа `customer:132456`, поэтому можно закрепить за ними префикс `customer`, но для других потребуется придумать что-то другое.

Маркеры могут быть распознаны и преобразованы в ссылку, пригодную для дальнейшего использования (например, `bug:76543` ссылается на систему контроля ошибок — «багтрекер»). Другие метки представляют собой единственное слово (например, `bogus` обычно применяется для пометки сообщений, сгенерированных в результате ложного срабатывания). Конечно, некоторые маркеры могут быть записаны с ошибками (`cause:netwrok`), а какие-то — вообще не содержать сколько-то полезной информации (`problem-went-away`), но лучше избегать жестко заданных предопределенных списков, позволяя командам искать

собственные стандарты и предпочтения, в результате чего мы получим более эффективное средство и в конечном итоге — надежные данные. В целом метки оказались весьма мощным инструментом, позволяющим командам выявлять и описывать проблемные места отдельно взятого сервиса, причем с минимальной формализацией этого анализа или даже совсем без нее. Каким бы простым ни казался механизм маркировки, он, вероятно, является одной из наиболее полезных уникальных возможностей Outalator.

Анализ

Разумеется, в обязанности SR-инженеров входит куда больше, чем просто устранение проблем и последствий инцидентов. Архивные данные полезны при реагировании на инциденты — вопрос «А что мы делали в прошлый раз?» всегда будет хорошей отправной точкой. Но архивная информация куда более полезна, если мы имеем дело с проблемами системными, повторяющимися или иными столь же широкого характера. Применение подобного анализа — одна из наиболее важных составляющих системы контроля неисправностей.

Нижний уровень анализа включает подсчет и базовое обобщение статистических данных для отчетности. Подробности зависят от конкретной команды, но такая информация, как количество инцидентов за неделю/месяц/квартал и количество оповещений в течение инцидента, присутствует обязательно. Следующий уровень более важен, но при этом прост в реализации: сравнения между командами/сервисами в динамике для раннего выявления закономерностей и тенденций. Этот уровень позволяет командам определить, является ли данная нагрузка «нормальной» относительно их собственной истории практики и других сервисов. Фраза «Это уже третий раз за неделю»

может быть как плохим, так и хорошим знаком, но если «это» обычно случается пять раз в день или, например, пять раз в месяц, то это уже дает почву для анализа.

Следующим шагом в анализе данных является поиск более сложных и масштабных проблем, что уже потребует не просто подсчетов, а более глубокого анализа. Прозрачный доступ к такой информации наряду с данными об инцидентах требуется, например, для определения компонента инфраструктуры, являющегося причиной наибольшего количества инцидентов, и затем оценки выигрыша от повышения стабильности или производительности данного компонента¹⁰². Простой пример: у разных команд действуют специфические для сервисов условия оповещений, такие как «устаревшие данные» или «большая задержка». Оба состояния могут быть вызваны перегрузкой сети, которая приведет к задержке репликации базы данных и потребует вмешательства. Они также могут объективно не превышать номинальный для данного сервиса уровень, но уже не соответствовать возросшим субъективным требованиям пользователей. Изучение этой информации с участием нескольких команд позволяет выявлять системные проблемы и выбирать правильное решение, особенно если речь идет о наиболее изощренных сбоях, чтобы предотвратить перерасход вычислительных мощностей.

Отчетность и общение

Среди наиболее востребованных SR-инженерами функций можно также назвать возможность отбирать несколько (возможно, ни одного) объектов из Outalator и включать их темы, метки и важные аннотации в письмо следующему дежурному инженеру (с произвольным списком адресов для

копий письма), чтобы сообщать о текущем состоянии при передаче смены. Для периодических проверок сервисов в промышленной эксплуатации (которые большинство команд проводит еженедельно) Outalator поддерживает «режим отчета», когда важные примечания разворачиваются параллельно с основным списком для обеспечения быстрого просмотра.

Неочевидная польза

Эффект от возможности определить, что оповещение или поток оповещений связаны с другим, уже известным сбоем, очевиден: ускоряется диагностика и снижается нагрузка на другие команды, которые будут знать, что данный инцидент уже учтен. Но есть и другие, менее очевидные преимущества. Возьмем, например, Bigtable: если в некотором сервисе выявлено нарушение функционирования, однозначно вызванное проблемами именно в Bigtable, но вы видите, что соответствующая SRE-команда не была оповещена, то, вероятно, следует сообщить им об этом самостоятельно. Улучшение взаимодействия между командами может внести и вносит значительный вклад в разрешение инцидентов или хотя бы смягчает их последствия.

Некоторые команды в компании зашли настолько далеко, что настраивают Escalator на простое перенаправление сообщений в Outalator, где они могут быть промаркированы, снабжены аннотациями и проанализированы. Одним из примеров применения такой «системы учета» является регистрация и проверка использования привилегированных или служебных учетных записей (стоит, однако, отметить, что это простейший функционал, пригодный скорее для технических внутренних проверок, чем для полноценного внешнего аудита). Другой вариант использования — запись и

снабжение аннотациями хода выполнения периодических задач, которые могут производить существенные изменения — например, автоматическое применение изменений схем, исходящее из системы контроля версий и достигающее баз данных.

[101](#) Например, некоторое изменение в сервисе Bigtable, дающее небольшой положительный эффект в случае одной конкретной неисправности, может требовать слишком больших усилий разработчиков. Однако если улучшение будет проявляться и во многих других случаях, эти усилия будут оправданными.

[102](#) С одной стороны, «порождает наибольшее количество инцидентов» — хороший критерий выбора при поиске путей сокращения числа происшествий и совершенствования системы в целом. С другой — эта метрика может говорить и просто об избыточной чувствительности средств мониторинга или о работе некоторых клиентских систем вне заданных параметров. И наконец, с третьей стороны, само по себе количество инцидентов не отражает ни сложность устранения их причин, ни степень тяжести их последствий.

17. Тестирование надежности систем

Авторы — Алекс Перри и Макс Люббе

Под редакцией Дианы Бейтс

*Пока вы не проверили сами,
считайте, что это сломано.*

Неизвестный

Одной из основных обязанностей SR-инженеров являются измерение и оценка надежности обслуживаемых ими систем. Для этого SR-инженеры применяют классические методы тестирования программ, адаптируя их для своих систем в требуемом масштабе [103](#). Надежность можно оценить через ее текущий уровень и через будущий (ожидаемый). Текущий уровень надежности определяется путем анализа данных мониторинга системы, а ожидаемый прогнозируется на основе имеющихся сведений о поведении системы в прошлом. Чтобы эти прогнозы были достаточно точными и полезными, должно выполняться одно из следующих условий.

- Сайт не претерпевает никаких изменений с течением времени, для него не выпускается новое ПО и не меняется парк серверов; следовательно, будущее поведение будет аналогично текущему.
- Все изменения, происходящие с сайтом, могут быть исчерпывающим образом описаны; следовательно, вносимые ими неопределенности могут быть проанализированы и учтены.

Тестирование — это механизм, позволяющий выявить отдельные области эквивалентного поведения системы в ходе

внесения в нее изменений¹⁰⁴. Каждый тест, проходящий успешно как до, так и после внесения изменений, сокращает неопределенность, которую необходимо изучать и анализировать. Тщательное (в идеале — исчерпывающее) тестирование помогает спрогнозировать будущую надежность для заданного сайта с достаточной для практического использования степенью детализации.

Необходимый объем тестирования зависит от требований к надежности системы. По мере повышения степени покрытия тестами вашего кода сокращается неопределенность и уменьшается вероятность снижения надежности в результате каждого изменения. Адекватное покрытие тестами позволяет внести больше изменений, прежде чем надежность снизится до неприемлемого уровня. Если вы вносите слишком много изменений и делаете это слишком быстро, прогнозируемый уровень надежности будет снижаться вплоть до допустимого порога. С этого момента вы, вероятно, предпочтете приостановить изменения, пока не накопятся новые данные мониторинга. Эти данные дополняют покрытие тестами, которые подтверждают надежность уже протестированных сценариев выполнения. Учитывая, что для обслуживаемых клиентов сценарии распределены случайным образом [Wood, 1996], благодаря статистической обработке наблюдаемых показателей можно экстраполировать их и на новые (еще не протестированные) сценарии. Эта статистика позволяет также определить области, для которых нужно провести более качественное тестирование или внести иные доработки.

Связь между тестированием и средним временем восстановления

Успешное прохождение теста или серии тестов еще не доказывает надежность программы. С другой стороны, если тест провалился, то это обычно доказывает ее ненадежность.

Ошибки может обнаруживать и система мониторинга, но не быстрее, чем срабатывает система оповещения о них. *Среднее время восстановления* (Mean Time to Repair, MTTR) показывает, сколько времени требуется команде эксплуатации, чтобы ликвидировать последствия ошибки вне зависимости от способа восстановления — путем отката изменений или как-то иначе.

Тестирующая система может обнаруживать ошибки с нулевым временем восстановления¹. Это происходит при тестировании уже собранной системы, когда тест находит ту же проблему, которая была бы обнаружена средствами мониторинга. При таком тестировании можно заблокировать проблемный код, и в «промышленную» версию эта ошибка не попадет (хотя ее все еще нужно исправить в исходном коде). Восстановление с нулевым временем восстановления путем блокирования конкретного обновления — это быстрый и удобный способ решения проблемы. Чем больше ошибок удастся найти и исправить при нулевом времени восстановления, тем больше будет *среднее время между сбоями* (Mean Time Between Failures, MTBF) для ваших пользователей.

Рост MTBF по мере повышения степени оттестированности системы позволяет разработчикам ускорять выпуск новых функций. Некоторые из них, конечно, будут содержать ошибки. Новые ошибки снова приведут к замедлению выпуска новых версий, поскольку потребуется исправление обнаруженных ошибок.

Авторы, пишущие о тестировании ПО, обычно сходятся в том, каково должно быть покрытие тестами. Большая часть разногласий происходит от противоречивой терминологии и различий в расстановке акцентов при оценке влияния тестирования в разных фазах жизненного цикла ПО или на разные характеристики тестируемых систем. Обсуждение тестирования в компании Google в целом вы можете найти по ссылке [Whittaker, 2012]. Следующие разделы знакомят с относящейся к тестированию терминологией, используемой в этой главе.[105](#)

Виды тестирования ПО

Тесты для ПО делятся на две большие группы: традиционные и производственные. Традиционные тесты обычно применяются в ходе разработки ПО для оценки корректности обособленных программ. Производственные тесты выполняются на веб-сервисе, работающем в «промышленной» среде для того, чтобы оценить, насколько корректно ведет себя развернутая система в реальных условиях.

Традиционное тестирование

Как показано на рис. 17.1, традиционное тестирование начинается с модульного (юнит-тестов)[106](#). Тестирование более сложной функциональности располагается в иерархии выше модульного.



Рис. 17.1. Иерархия традиционных тестов

Модульное тестирование

Модульные тесты — это наиболее простая и наименее масштабная разновидность тестирования ПО. Они используются для того, чтобы можно было убедиться в корректности отдельного модуля или компонента («юнита») программы, например класса или функции, независимо от более крупной программы, содержащей этот компонент. Они также могут использоваться в качестве спецификации компонента, чтобы гарантировать, что функция или модуль соответствует требованиям системы. Именно модульные тесты

обычно применяются для организации процесса «разработки через тестирование» (test driven development, TDD).

Интеграционное тестирование

Программные компоненты, прошедшие индивидуальное тестирование, объединяются в более крупные единицы. Для них выполняются *интеграционные тесты*, позволяющие убедиться, что эти компоненты корректно функционируют. Так называемая инъекция зависимостей (dependency injection), выполняемая с помощью инструментов вроде Dagger¹⁰⁷, — это очень мощный метод создания так называемых «моков» (англ. mock)¹⁰⁸, моделирующих сложные зависимости и упрощающих тестирование программы. Распространенным примером этого приема служит подмена полноценной базы данных облегченным «моком», реализующим явно заданное поведение.

Системное тестирование

Системные тесты — это наиболее масштабные тесты, которые запускаются для всей развернутой системы. В систему объединяются все модули, относящиеся к определенным ее компонентам (например, серверам), которые уже прошли интеграционные тесты. Далее необходимо тестировать функциональность всей системы в целом. Системные тесты имеют множество разновидностей.

- *Тесты на общую доступность* (*smoke tests*). Это одни из самых простых системных тестов, в рамках которых проверяются очень простые, но очень важные аспекты поведения. Тесты на общую доступность также иногда называются *sanity*

testing («тестирование исправности»), и они создают основу для более сложных тестов.

- *Тесты производительности (performance tests).* Как только мы убедились в работоспособности базовой функциональности системы, следующим шагом станет написание очередного системного теста, который позволит убедиться, что производительность системы будет оставаться на приемлемом уровне в течение всего жизненного цикла ПО. Поскольку время отклика для взаимосвязанных компонентов или используемых ресурсов может значительно изменяться в течение всего периода разработки, систему нужно протестировать и убедиться, что она не станет замедляться на глазах пользователей. Например, некая программа может потребовать 32 Гбайт оперативной памяти, хотя раньше она обходилась восемью, или же время ответа, равное 10 миллисекундам, увеличится сначала до 50, а затем и до 100 миллисекунд. Тесты производительности позволяют гарантировать, что с течением времени система не деградирует или не станет слишком дорогостоящей из-за необходимости наращивать ее мощность.
- *Регрессионные тесты (regression tests).* Еще один вид системных тестов призван предотвратить повторное проникновение ошибок в код. Регрессионные тесты можно представить как галерею ошибок, которые когда-либо привели к сбоям системы или к неверным результатам. Документирование этих ошибок как тестов системного или интеграционного уровня позволяет при переписывании кода убедиться, что в нем снова не появились те ошибки, на поиск и исправление которых уже были потрачены время и силы.

Важно отметить, что все эти тесты имеют свою стоимость как с точки зрения затрат времени, так и с точки зрения необходимых вычислительных ресурсов. На одном конце шкалы находятся модульные тесты, которые очень дешевы с обеих точек зрения, поскольку их можно выполнить в считанные миллисекунды на лэптопе. На другом — создание полноценного сервера со всеми взаимосвязанными компонентами (или замещающими их эквивалентами), необходимыми для запуска соответствующих тестов, может потребовать значительно больше времени — от нескольких минут до нескольких часов — и, возможно, выделения дополнительных вычислительных ресурсов. Понимание этого важно для повышения продуктивности труда разработчиков, а также способствует более эффективному использованию средств тестирования.

Тестирование в промышленном окружении

Тесты в промышленном окружении (тесты в рабочей среде, *production tests*), в противоположность тестам в изолированной тестовой среде, взаимодействуют с реально работающей системой. Они очень похожи на мониторинг по методу черного ящика (см. главу 6), поэтому такой подход иногда называют *тестированием методом черного ящика*. Такие тесты критически важны для обеспечения надежной работы сервиса под реальной нагрузкой.

Путаница из-за обновлений

Часто говорят, что тестирование выполняется (или должно выполняться) в изолированной (герметичной) среде [Narla, 2012]. Это утверждение подразумевает, что промышленная

среда – не изолированная. Конечно, она обычно не бывает изолированной, поскольку установка обновлений в виде небольших, хорошо изученных и протестированных фрагментов регулярно вносит изменения в действующую систему и ее окружение.

Чтобы справиться с неопределенностью и оградить пользователей от рисков, изменения можно устанавливать в действующую систему не в том порядке, в котором они появляются в системе контроля версий. Обновления часто проходят в несколько этапов с помощью механизмов, которые последовательно модифицируют пользовательскую среду, сопровождая это мониторингом на каждом шаге, чтобы гарантировать отсутствие неожиданных, хотя и предсказуемых проблем в новом окружении. В результате все «промышленное» окружение в целом заведомо не соответствует полностью какой-либо конкретной версии, хранящейся в системе контроля версий.

Система контроля версий позволяет иметь несколько версий исполняемых файлов и связанных с ними конфигураций, ожидающих переноса на действующую систему. В такой ситуации могут возникнуть проблемы, если тесты выполняются в этой системе. Например, тест может использовать последнюю версию конфигурационного

файла из системы контроля версий в сочетании с работающей более старой версией файла программы. Или же

вы можете запустить тест с более старой версией конфигурационного файла и увидеть ошибку, которая уже исправлена в более новой версии этого файла.

Аналогично системный тест может использовать конфигурационные файлы для сборки своих модулей перед запуском теста. Если этот тест проходит успешно, но конфигурационный тест (их мы рассмотрим в следующем разделе) для этой версии – неуспешно, то результат нашего теста будет корректен в изолированной среде, но не для реально функционирующей системы. Такой результат нас не устраивает.

Тестирование конфигураций

В компании Google конфигурации веб-сервисов описаны в файлах, которые хранятся в нашей системе контроля версий. Для каждого конфигурационного файла отдельный *тест конфигурации* проверяет, как реально настраивается конкретная программа, и сообщает о найденных несоответствиях. Такие тесты изначально не являются изолированными, поскольку работают за пределами «песочницы»¹⁰⁹ тестового окружения.

Тесты конфигурации создаются (и тоже тестируются!) для каждой конкретной версии конфигурационного файла, сохраненного в системе контроля версий. Сопоставляя то, какая версия теста проходит относительно целевой версии, можно косвенно судить о том, насколько текущая «промышленная» версия системы отстает от «разработчикской».

Эти негерметичные конфигурационные тесты оказываются особенно полезны как часть распределенной системы мониторинга, поскольку закономерности их успешного и неуспешного прохождения в промышленном окружении помогает выявлять в иерархии сервисов сценарии, для которых нет адекватной комбинации локальных конфигураций. Правила мониторинга пытаются найти эти нежелательные сценарии среди сценариев выполнения реальных запросов пользователей, взятых из журналов трассировки. Любые найденные совпадения становятся оповещениями о небезопасности текущей версии и/или выполняемой модификации, и нужно это каким-то образом исправить.

Конфигурационные тесты могут быть очень простыми, когда при развертывании в промышленной среде используется реальное содержимое файла и выполняется запрос в реальном времени для получения копии содержимого. В таком случае код теста просто выполняет этот запрос и сравнивает полученный ответ с файлом. Тесты становятся более сложными, если конфигурации:

- неявно содержат (то есть учитывают) значения по умолчанию, присутствующие в файле программы (это означает, что тесты соответствуют только «своим» версиям программ);
- проходят через препроцессор (например, скрипт bash), превращаясь в параметры командной строки (происходит преобразование тестовой конфигурации по заданным правилам);
- относятся к разделяемой среде выполнения (это делает тесты зависимыми от графика установки обновлений других компонентов в ней).

Нагрузочное тестирование

Для того чтобы безопасно эксплуатировать систему, SR-инженерам необходимо знать ограничения как самой системы, так и отдельных ее компонентов. Компоненты под нагрузкой в определенный момент вместо корректного прекращения работы нередко терпят аварии или порождают катастрофические сбои. *Нагрузочные тесты* (или *стресс-тесты*) служат для того, чтобы определить допустимые границы нагрузки для веб-сервиса. Нагрузочное тестирование отвечает на следующие вопросы.

- Насколько заполненной может быть база данных, прежде чем она начнет давать сбои?
- Сколько запросов в секунду может быть отправлено на сервер, прежде чем начнутся его отказы из-за перегрузки?

Канареечное тестирование

Как можно заметить, *канареечные тесты* (*canary*) отсутствуют в списке тестов промышленного окружения. Термин *canary* происходит от выражения *canary in a coal mine* («канарейка в угольной шахте») и изначально относился к практике использования живой птицы для обнаружения ядовитых газов в шахте до того, как ими отравятся люди.

В ходе канареечного теста обновление версии программ или конфигурации устанавливается на определенную часть серверов, которые остаются в этом состоянии на время «инкубационного периода». Если за это время не случается никаких неожиданностей, установка обновлений продолжается для остальных серверов прогрессирующими темпами^{[110](#)}. Если же что-то идет не так, то каждый отдельно взятый сервер можно откатить до известного предыдущего стабильного

состояния. Такой прием с инкубационным периодом мы обычно называем «просушкой» (англ. baking the binary).

Канареечный тест — это не столько тест, сколько спланированная и организованная приемка версии пользователями. В то время как конфигурационные и нагрузочные тесты подтверждают наличие особых состояний в конкретных программах, канареечный тест решает другую задачу. Он лишь показывает, как программа работает с менее предсказуемым реальным трафиком, при этом он не всегда обнаруживает новые ошибки, и в этом его несовершенство¹¹¹.

Рассмотрим конкретный пример канареечного теста: возьмем некую ошибку, которая относительно редко влияет на пользовательский трафик, и посмотрим, как она себя проявляет в экспоненциально распространяющем обновлении. Накапливающееся количество зафиксированных отклонений ожидается равным $CU = RK$, где U — это порядок ошибки (эта величина будет определена позже), R — частота появления ошибок, а K — период, в течение которого объем трафика возрастает в e раз (или, что то же самое, на 172 %)¹¹².

Чтобы избежать негативной реакции пользователей, версию с нежелательными отклонениями в работе нужно быстро откатить к предыдущей конфигурации. За то короткое время, которое потребуется для обнаружения проблемы и исправления, будет, скорее всего, сгенерировано еще несколько отчетов. Когда пыль осядет и все успокоится, с помощью этих отчетов можно будет оценить накопленное количество ошибок C и частоту R .

Путем деления и масштабирования на величину периода K получим примерное значение U — порядок ошибки¹¹³. Рассмотрим несколько примеров.

- $U = 1$. Пользовательский запрос столкнулся с кодом, который попросту не работает.
- $U = 2$. Пользовательский запрос случайным образом повреждает данные, которые может увидеть один из последующих пользовательских запросов.
- $U = 3$. Данные, поврежденные случайным образом, также являются корректным идентификатором для предыдущего запроса.

Большинство дефектов имеют первый порядок: они масштабируются линейно относительно роста пользовательского трафика [Perry, 2007]. Как правило, их можно отследить, преобразовав в регрессионные тесты журналы всех запросов с ненормальными результатами. Для дефектов более высокого порядка эта стратегия не работает: запрос, который регулярно выдает ошибку, выполняясь по порядку после ряда предыдущих, внезапно начинает работать нормально, если опустить некоторые из них. Важно выявить эти и подобные дефекты еще в процессе установки обновлений, поскольку эксплуатационная нагрузка возрастает очень быстро — экспоненциально.

Учитывая соотношение дефектов высокого и низкого порядков, при использовании стратегии экспоненциального распространения обновлений не обязательно добиваться полностью сбалансированного деления пользовательского трафика. Пока каждый метод для заданной части трафика применяет один и тот же интервал K , примерное значение U будет корректным, даже если вы не можете определить, какой именно метод помог обнаружить ошибку. Последовательно применяя множество методов и допуская при этом некоторые перекрытия, удается поддерживать значение K небольшим.

Такая стратегия минимизирует общее количество встречаемых пользователем отклонений C , позволяя получить раннюю оценку величины U (конечно же, вы надеетесь получить 1).

Окружения сборки и тестирования проекта

Было бы здорово продумывать эти тесты и прорабатывать сценарии сбоя с первого же дня работы над проектом, но зачастую SR-инженеры присоединяются к разработчикам уже на стадии реализации — когда выбранная модель проверена и утверждена, библиотеки проверены на масштабируемость алгоритмов и, возможно, уже готовы макеты всех пользовательских интерфейсов. Но база кода команды все еще остается в состоянии прототипов, а полноценное тестирование еще не внедрено и, вероятно, даже не спланировано. С чего вы должны начинать тестирование проекта в таких ситуациях? Строить полное покрытие модульными тестами всех ключевых функций и классов — удручающая перспектива, если текущее тестовое покрытие невелико или вовсе отсутствует. Лучше начинать тестировать так, чтобы с минимальными усилиями получить наибольшую отдачу.

Вы можете начать работу, задав следующие вопросы.

- Можно ли каким-либо способом выделить наиболее важное место в базе кода? Экстраполируя принцип, известный из разработки и управления проектами, если все задачи имеют одинаково высокий приоритет, то высокого приоритета не имеет ни одна из них. Можно ли отсортировать по важности компоненты тестируемой системы?
- Существуют ли функции и классы, которые однозначно необходимы для конкретной задачи или для потребительских качеств? Например, код, который отвечает

за биллинг, обычно важен для заказчика. Код биллинга также зачастую отделен от других частей системы.

- С какими API будут работать другие команды? Даже если ошибки будут выявлены финальными тестами и не затронут пользователей, они способны дезориентировать другую команду разработчиков, которые из-за этого могут написать неверные (или неоптимальные) приложения для вашего API.

Передача в эксплуатацию некорректно работающего ПО — это главный смертный грех разработчика. Создать набор тестов на общую доступность для каждой выпускаемой версии совсем не трудно. Давая большой эффект при малых затратах, это может стать первым шагом на пути к созданию хорошо протестированного, надежного ПО.

Одним из способов создания строгой культуры тестирования¹¹⁴ является документирование всех обнаруженных ошибок и дефектов как «тестовых случаев». Если каждый дефект превратится в тест, то ожидаемо, что каждый тест поначалу будет проходить неуспешно, поскольку дефект еще не устранен. По мере того как инженеры будут исправлять ошибки, программа начнет успешно проходить тесты и вы встанете на путь создания полного набора регрессионных тестов¹¹⁵.

Еще один ключевой фактор для создания хорошо протестированного ПО — настройка инфраструктуры тестирования. Основой для мощной инфраструктуры тестирования служит система контроля версий, которая отслеживает каждое изменение в базе кода.

Как только настроите систему контроля версий, вы сможете добавить систему непрерывной сборки¹¹⁶, которая будет

автоматически выполнять сборку программ при каждом изменении в базе кода. По нашим наблюдениям, инженеров следует сразу же оповещать, если очередные изменения привели к ошибкам. Я рисую озвучить очевидное, но очень важно, чтобы последняя версия программного проекта в системе контроля версий всегда была полностью работоспособной. Когда система, отвечающая за сборку проекта, оповещает разработчиков о неработающем коде, они должны приостановить выполнение всех других задач и заняться решением этой проблемы. Относиться к дефектам с такой серьезностью важно по нескольким причинам.

- Как правило, новые изменения, внесенные после появления дефекта, затрудняют его исправление.
- Дефектное ПО замедляет команду, поскольку приходится искать способы временно обойти проблему.
- Регулярный выпуск новых версий, например ночные и еженедельные сборки, становятся бесполезными.
- Намного проблематичнее становится выпускать срочные обновления (например, для устранения уязвимости в системе безопасности).

Так сложилось, что в мире SRE концепции стабильности и гибкости обычно противоположны друг другу. В последнем пункте показан интересный случай, когда стабильность приносит с собой гибкость. Когда сборка работает надежно, разработчики могут быстрее создавать новые версии!

Некоторые системы для сборки ПО, вроде Bazel¹¹⁷, имеют ценную функциональность, которая позволяет более четко управлять процессом тестирования. Например, Bazel создает

графы зависимостей для программных проектов. Когда в некий файл вносится изменение, Bazel выполняет повторную сборку только той части проекта, которая зависит от этого файла. Такие системы обеспечивают возможность воспроизводимых сборок. Вместо запуска всего набора тестов при каждом изменении в проекте тесты будут выполняться только для изменившихся файлов. В результате тестирование становится дешевле и быстрее^{[118](#)}.

Существует множество инструментов, которые помогут вам рассчитать и визуализировать необходимый уровень тестового покрытия [Cranmer, 2010]. Используйте их для формулирования целей и расстановки приоритетов тестирования: вам нужно подойти к процессу создания хорошо протестированного кода как к инженерному проекту, а не к философскому упражнению. Вместо того чтобы повторять двусмысленную фразу «Нам нужно больше тестов», установите явные цели и сроки.

Помните, что не все программы создаются одинаково. Критически важные системы требуют гораздо более качественного тестирования и гораздо лучшего покрытия тестами, чем скрипт, не устанавливаемый на «промышленный» сервер и имеющий небольшой срок жизни.

Масштабирование тестирования

Теперь, когда мы рассмотрели основы тестирования, взглянем, как служба SRE, применяя системный подход к тестированию, обеспечивает сохранение надежности при масштабировании систем.

Небольшой тест отдельного модуля может иметь короткий список зависимостей: один файл исходного кода, тестирующая библиотека, динамически подключаемые библиотеки,

компилятор и аппаратная платформа, на которой запускаются тесты. Правильно построенная среда тестирования требует наличия тестовых покрытий для всех этих зависимостей, и их тесты должны проверять все специфические варианты использования, ожидаемые другими компонентами среды. Если реализация теста зависит от участка кода внутри подключаемой библиотеки, не покрытого тестами, то не связанное с ним напрямую изменение среды^{[119](#)} может привести к тому, что модуль будет проходить тест успешно независимо от наличия в нем ошибок.

В противоположность этому, тест готового продукта может зависеть от огромного множества других компонентов системы, потенциально — от каждого объекта кода. Если тест должен выполняться в чистом промышленном окружении, то каждое небольшое изменение потребует полного восстановления среды. На практике для тестовой среды стараются выбирать точки ветвления версий. Это позволяет протестировать как можно больше зависимостей за минимальное количество итераций. Конечно, если в какой-либо ветви обнаруживается ошибка, необходимо выбрать дополнительные точки ветвлений.

Тестирование масштабируемых инструментов

Инструменты SRE нуждаются в тестировании так же, как и любые другие программы^{[120](#)}. Инструменты, разработанные в отделе SRE, могут выполнять следующие задачи.

- Съем показателей производительности баз данных.
- Прогнозирование показателей использования для планирования рисков, связанных с емкостью.

- Рефакторинг данных «теневого» экземпляра сервиса, недоступного пользователю.
- Изменение файлов на сервере.

Инструменты SRE имеют две общие характеристики.

- Их побочные эффекты остаются скрытыми внутри протестированного основного API.
- Они изолированы от запущенных пользовательских систем существующим барьером валидации процедур выпуска обновлений.

Барьеры для защиты от небезопасного ПО

Использование программ, работающих в обход обычного хорошо протестированного API (даже если это делается без злого умысла), может обернуться хаосом на промышленном сервисе. Например, реализация ядра СУБД может позволить администраторам временно отключить механизм транзакций, чтобы сократить затраты времени на обслуживание. Если эта же реализация используется и для ПО, выполняющего запись в базу в пакетном режиме, изоляция пользователей может оказаться нарушенной при любом обращении к доступной многим пользователям действующей копии базы. Предотвратить риск таких разрушительных последствий позволяют проектные решения.

1. Используйте отдельный инструмент для того, чтобы настроить барьер в конфигурации процедуры репликации таким образом, чтобы копия не проходила проверку работоспособности. В результате эта копия не будет доступна для пользователей.
2. Конфигурируйте небезопасное ПО так, чтобы оно при запуске проверяло наличие барьера. Позволяйте небезопасному ПО работать только с неактивными копиями базы.
3. Используйте средства проверки работоспособности копий базы при мониторинге методом черного ящика для корректного преодоления барьера.

Средства автоматизации — это тоже программы. Поскольку связанные с ними риски оказываются на другом уровне в иерархии сервисов, они не нуждаются в столь тщательном тестировании. Средства автоматизации выполняют следующие задачи.

- Выбор индексов базы данных.
- Балансировка нагрузки между дата-центрами.
- Выборка и сортировка содержимого журналов активных узлов сети для быстрого ремастеринга.

Все средства автоматизации обладают двумя общими свойствами.

- Работают с устойчивым, предсказуемым и хорошо протестированным API.
- Результаты их работы можно рассматривать как побочный эффект, невидимый для других клиентов этого API.

Тестирование позволяет продемонстрировать желаемое поведение других уровней иерархии сервисов до и после внесения изменений. Часто также можно проверить, изменяется ли внутреннее состояние инструмента за время его работы, насколько это видно посредством API. Например, базы данных продолжают выдавать корректные ответы на запросы, даже если для них нет подходящих индексов. С другой стороны, некоторые сущности API, документированные как неизменяемые (например, кэш DNS, содержимое которого сохраняется в течение времени TTL), могут изменяться от операции к операции. Скажем, если изменение уровня (режима) исполнения сервиса заменяет локальный сервер имен кэширующим прокси-сервером, в обоих случаях результаты запросов должны сохраняться сервером на протяжении достаточно длительного времени, однако вряд ли состояние кэша будет при переключении передано от одного сервера другому.

Если средства автоматизации подразумевают дополнительные тесты исполняемых файлов для проверки поведения в различном окружении, то как вы определите, в каком именно окружении они работают? В конце концов, инструмент автоматизации, реализующий перемещение программных контейнеров для оптимального перераспределения нагрузки, скорее всего, попытается в какой-то момент оптимизировать сам себя, если он тоже работает как контейнер. И было бы крайне неприятно, если бы

новая версия его внутреннего алгоритма заполняла страницы памяти так быстро, что полоса пропускания сети и промежуточных зеркал оказалась бы исчерпана, не позволяя завершить перемещение работающего кода. Даже при наличии интеграционного теста, для которого бинарный файл целенаправленно перетасовывает сам себя, он, скорее всего, использует гораздо меньший набор контейнеров, чем в реальном промышленном окружении. Ему вряд ли дадут использовать дефицитные и обладающие большой задержкой глобальные каналы связи для тестирования таких ситуаций.

Интереснее бывает, когда один инструмент автоматизации может изменять окружение, в котором работает другой. Или же два инструмента одновременно меняют окружение друг друга! Например, инструмент для установки обновлений парка машин скорее всего постарается забрать себе как можно больше ресурсов в ходе такой установки обновлений. В результате балансировщик контейнеров решит переместить контейнер с этим инструментом. Но в этот момент очередь обновляться может дойти до самого балансировщика. Такая круговая зависимость может и не причинить вреда, но только если для используемых API реализована возможность перезапуска, если эту логику не забыли протестировать и если есть независимая проверка работоспособности инструментов.

Тестируем катастрофы

Многие средства восстановления после аварий сознательно проектируются так, чтобы они работали автономно (оффлайн). Они выполняют следующие задачи.

- Определение состояний в контрольных точках, соответствующих нормальной остановке сервиса.

- Сохранение состояний в контрольных точках для использования их существующими средствами проверки, применяемыми в нормальном (не аварийном) режиме.
- Поддержка обычных средств организации *барьеров*, которые инициируют процедуру *корректного старта*.

Во многих случаях вы можете реализовать эти фазы так, чтобы связанные с ними тесты было удобно писать и они покрывали большую часть программы. Если же приходится нарушать какие-либо из ограничений (работа в режиме оффлайн, контрольные точки, загружаемость, барьеры или корректный старт), то вам будет гораздо труднее обеспечить уверенность в том, что соответствующий инструмент сработает в любой момент по первому требованию.

Онлайн-инструментам восстановления свойственно работать вне рамок основных API, поэтому их тестирование более интересно. Одна из проблем, с которой вы можете столкнуться при тестировании распределенной системы, — выявление ситуации, когда нормальное поведение, корректное по своей сути, неправильно взаимодействует с процедурами восстановления. Например, рассмотрим попытку анализировать условия возникновения гонок с помощью онлайн-инструментов. Оффлайн-инструменты в общем случае рассчитаны на более простую проверку корректности состояния в конкретный отдельно взятый момент, а не в процессе функционирования в течение некоторого промежутка времени. Положение осложняется тем, что участвующие в гонках исполняемые файлы (работающая промышленная версия и предназначенная для восстановления) обычно собираются отдельно друг от друга. Следовательно, для использования в этих тестах вам понадобится специальный

исполняемый файл с предусмотренным унифицированным интерфейсом для инструментов, чтобы они могли отслеживать транзакции.

Использование статистических тестов

Статистические методы вроде Lemon (Ana07) для «нечеткого тестирования», Chaos Monkey¹ и Jepsen² для состояния распределенных систем не обязательно будут повторяемыми. Простой перезапуск таких тестов после изменения кода не может однозначно доказать, что наблюдаемая ошибка исправлена³. Однако в ряде случаев они могут быть полезны.

Они могут обеспечить журналирование всех случайно выбранных операций, выполнявшихся при заданном прогоне, — иногда просто записывая в журнал начальное значение генератора случайных чисел.

Если такой журнал служит для тестирования готового продукта, может быть полезно выполнить несколько прогонов до создания отчета об ошибке. Процент успешно завершившихся тестов покажет вам, насколько сложно будет в дальнейшем доказать, что ошибка исправлена.

Различные варианты проявления ошибки могут помочь вам более точно определить подозрительные области вашего кода.

При последующих запусках могут обнаружиться более серьезные сбойные ситуации. Поэтому, возможно, вам

потребуется усугубить влияние этой ошибки в тестах.

В погоне за скоростью [121122123](#)

Для каждой версии или обновления-заплатки (патча) в репозитории кода каждый отдельно взятый тест показывает, пройден он или нет. Эти данные могут меняться при повторяющихся и, казалось бы, идентичных запусках. Вы можете оценить вероятность прохождения теста, взяв среднее значение для достаточно большого количества запусков, а также вычислить статистическую неопределенность этой оценки. Однако трудоемкость таких расчетов делает их выполнение для каждого теста и каждой версии практически невозможным.

Вместо этого лучше определить предположительное количество интересующих сценариев и запустить заданное количество прогонов каждого теста каждой версии, чтобы можно было сделать обоснованные выводы. Некоторые из этих сценариев будут исполняться благополучно (с точки зрения качества кода), а другие потребуют принять меры. Эти сценарии сказываются во всех тестах в разных масштабах, и, поскольку они связаны с гипотезами, надежное и быстрое получение списка требующих дальнейшей работы гипотез (например, о компонентах, которые сейчас неисправны) означает одновременно и получение оценки всех сценариев.

Обращаясь к инфраструктуре тестирования, разработчики хотят выяснить, работает ли их код — обычно это только небольшая часть исходного кода, которая покрыта данным тестом. И если он исправен, то зачастую подразумевается, что ответственность за все наблюдаемые сбои можно переложить на чей-то другой код. Другими словами, инженер хочет знать,

возникают ли в его коде непредвиденные ситуации гонок, из-за чего прохождение теста становится нестабильным (или более нестабильным, чем оно было под действием других факторов).[124](#)

Ограничение длительности теста

Большинство тестов являются простыми в том смысле, что они запускаются как замкнутый («герметичный») исполняемый файл, который за считаные секунды упаковывается в небольшой программный контейнер. Эти тесты обеспечивают разработчикам интерактивную обратную связь, которая успевает сообщить об ошибках раньше, чем инженер переключится на работу над следующей проблемой или новой задачей.

Время старта тестов, которые требуют взаимодействия между несколькими исполняемыми файлами и/или всего множества контейнеров, может измеряться секундами. Такие тесты обычно не обеспечивают достаточно малое время отклика, поэтому их можно назвать уже не интерактивными, а пакетными. Вместо того чтобы сказать выполняющему тест инженеру: «Не закрывайте вкладку редактора», такие тесты завершаются как неуспешные и говорят: «Этот код не готов к проверке».

Неформально длительность теста ограничивается тем моментом, когда инженер переключается на другую задачу. Результаты теста надо давать инженеру до того, как он (или

она) переключится, иначе новая задача может быть как та компиляция из ХСКД¹.

Предположим, инженер работает над сервисом, для которого есть около 21 000 простых тестов, и время от времени вносит исправления в кодовую базу сервиса. Чтобы протестировать каждое такое исправление, нужно сравнить векторы результатов прохождения/непрохождения теста всем содержащимся в базе кодом до и после поправки. Совпадение этих двух векторов позволяет считать текущую версию кода в базе «предварительно допущенной» к выпуску. За этим последует запуск множества финальных и интеграционных тестов, а также прочие распределенные тесты исполняемых файлов, проверяющих систему на масштабируемость (если дополнение использует значительно больше локальных вычислительных ресурсов) и на сложность (если оно порождает сверхлинейный рост нагрузки где-либо еще).

Как часто вы можете ошибаться, помечая пользовательские обновления как деструктивные из-за неверной оценки нестабильности среды тестирования? Скорее всего, пользователи будут бурно возмущаться отклонением каждого десятого обновления, но отказ в одном случае из 100 выражений вызвать не должен.

Это значит, что вам нужен корень степени 42 000 (по одному прогону каждого из 21 000 тестов до и после обновления) из 0,99 (доля обоснованно отклоняемых тестов). Формула:

$$0,99^{\frac{1}{2 \times 21000}}$$

говорит, что каждый отдельный тест должен работать корректно в 99,9999 % случаев. Хм-м.

Передача в промышленную эксплуатацию

Управление конфигурациями промышленных экземпляров систем, как правило, тоже возлагается на репозиторий системы контроля версий, но конфигурации чаще хранятся отдельно от исходного кода, с которым работают программисты. Аналогично тестирующая инфраструктура часто не может «видеть» конфигурацию реальной промышленной системы. Даже если они находятся в одном репозитории, изменения в конфигурациях выполняются в разных его ветках и/или в изолированном дереве каталогов, которые в сложившейся практике игнорируются средствами автоматизации тестирования.

Традиционно программисты разрабатывают программы в своем окружении и затем переправляют их администраторам для установки на серверы. При этом разграничение конфигураций тестирования и промышленной эксплуатации в лучшем случае раздражает, а в худшем — вредит надежности и гибкости системы. Это может также приводить к дублированию программных инструментов. В результате ухудшается устойчивость и предсказуемость находящегося в эксплуатации окружения, которое должно было бы быть интегрированным, из-за неочевидных несоответствий между поведением двух наборов программ. Наконец, конфликты между конкурирующими обращениями к системам контроля версий ограничивают скорость разработки проекта.

В рамках концепции SRE также можно наблюдать негативное влияние разграничения тестовой и промышленной конфигураций — оно нарушает соответствие между моделями, которые описывают промышленную систему и поведение

приложений. Это мешает выявлению еще на этапе разработки статистических несоответствий планируемым показателям. Однако вред от замедления разработки оказывается не так велик, как польза от предотвращения повреждения структуры системы, поскольку невозможно полностью избавиться от рисков при миграции.

Рассмотрим порядок унифицированного управления версиями и унифицированного тестирования, при которых применима методология SRE. Как повлияет ошибка при миграции распределенной архитектуры? Вероятно, будет выполнено какое-то достаточно большое количество тестов. Далее, предположим, что программист, скорее всего, согласится с тем, что система тестирования ошибается один раз из десяти или около того. На какие риски вы готовы пойти при миграции, зная, что тесты могут ошибаться и ситуация может стать действительно тревожной, причем очень быстро? Очевидно, некоторые области покрытия должны тестироваться более параноидально, чем другие. Эту особенность можно обобщить: ошибки, найденные одними тестами, более критичны для системы, чем найденные другими.

Ожидание сбоя тестов

Не так давно программный продукт мог выпускаться всего раз в год. Его исполняемые файлы компилировались на протяжении нескольких часов или дней, и большая часть тестирования выполнялась людьми вручную и согласно написанным вручную инструкциям. Такой процесс был неэффективным, но не было практической необходимости автоматизировать его. Затраты на выпуск новой версии в основном были связаны с документацией, миграцией данных, переобучением пользователей и другими факторами. Среднее

время между сбоями (Mean Time Between Failure, MTBF) составляло один год, независимо от того, какой объем тестирования был выполнен. В новой версии появлялось так много изменений, что некоторые сбои, видимые пользователям, оказывались скрыты внутри ПО. По сути, надежность, достигнутая в предыдущей версии, не имела отношения к следующей.

Эффективные инструменты для управления API/ABI и интерпретируемые языки, масштабируемые для крупных программных систем, теперь обеспечивают возможность сборки и запуска новой версии продукта каждые несколько минут. В принципе, достаточно большая армия людей¹²⁵ тоже может обеспечивать тестирование с тем же уровнем качества каждой новой версии (в том числе промежуточных) с помощью методов, описанных ранее. И хотя в конечном счете для одного и того же кода будут выполняться одни и те же тесты, качество финальной ежегодной версии продукта будет выше, поскольку в дополнение к ежегодным версиям будут протестированы и промежуточные версии. Благодаря промежуточным версиям вы можете однозначно соотносить найденные при тестировании проблемы с их причинами и быть уверенными в том, что устранили именно проблему, а не только ее симптом. Точно так же сокращение цикла обратной связи эффективно и применительно к автоматическому тестированию.

Если вы позволите пользователям испытать за год больше версий продукта, MTBF ухудшится, поскольку появится больше возможностей для возникновения сбоев, заметных пользователям. Однако вместе с тем вы сможете выявить области, которым пойдет на пользу дополнительное тестовое покрытие. Если эти тесты будут реализованы, каждое улучшение будет защищать от каких-то сбоев в будущем. При продуманном управлении надежностью соблюдается баланс

достоверности тестового покрытия и допустимого количества видимых пользователям сбоев, а также соответствующим образом корректируется частота выпуска новых версий. Это позволяет максимизировать полезную информацию, получаемую в результате тестирования и от конечных пользователей. В итоге можно управлять тестовым покрытием и, в свою очередь, скоростью выпуска версий продукта.

Когда SR-инженеры модифицируют конфигурационный файл или оптимизируют стратегию средств автоматизации, то эта инженерная работа (противоположная реализации пользовательских функций) соответствует той же концептуальной модели. Когда вы определяете частоту выпуска версий, исходя из надежности, зачастую имеет смысл разделить «бюджет надежности» по функциональности или (что более удобно) по командам. При таком сценарии команда, разрабатывающая функциональность, стремится достичь заданного уровня достоверности тестирования, который влияет на частоту инициированных ими выпусков версий. Команда SRE, имея свой отдельный бюджет и свои критерии достоверности, может выпускать новые версии с большей частотой.

Для того чтобы оставаться надежным и при этом избегать линейного увеличения количества SR-инженеров, поддерживающих сервис, промышленное окружение должно функционировать практически без участия человека. Для этого оно должно быть устойчивым к небольшим сбоям. В случае же более серьезного происшествия, требующего ручного вмешательства SR-инженеров, используемые ими инструменты должны быть хорошо протестированы. В противном случае такое вмешательство только снижает уверенность в том, что уже накопленные данные останутся доступными и актуальными и в ближайшем будущем. Когда происходит такое

снижение надежности, необходимо дождаться результатов анализа данных мониторинга, чтобы устраниить неопределенность. Если подраздел «Тестирование масштабируемых инструментов» выше был в основном о том, как добиться с помощью инструментария SR-инженеров требуемого тестового покрытия, то здесь вы видите, как определить, насколько часто следует применять эти инструменты в промышленной среде.

Конфигурационные файлы существуют в общем потому, что вносить в них изменения можно гораздо быстрее, чем заново пересобирать программы. Такая оперативность зачастую бывает важным фактором поддержания низкого MTTR. Однако частые изменения в этих файлах могут иметь и другие причины. В частности, с точки зрения надежности.

- Конфигурационный файл, служащий для обеспечения малого MTTR и модифицируемый только после сбоев, имеет период обновления больший, чем MTBF. Это может привести к существенной неопределенности, является ли такое изменение (сделанное вручную) действительно оптимальным без тех изменений, которые снижали общую надежность сайта.
- Конфигурационный файл, который меняется несколько раз за время между выпусками новых версий приложения (например, потому, что в нем хранится текущее состояние приложения), может представлять серьезный риск, и к ним нужно относиться так же, как и к изменениям приложения. Такой конфигурационный файл будет снижать надежность сайта, если только покрытие тестами и средствами мониторинга для него не будет даже лучше, чем для приложения.

Один из подходов к конфигурационным файлам состоит в том, что вы должны или убедиться, что каждый конфигурационный файл попадает только в одну из описанных выше категорий, или каким-то образом обеспечить это. Если вы выберете второе, то убедитесь в следующем.

- Каждый конфигурационный файл имеет достаточное тестовое покрытие для поддержки рутинных изменений.
- Перед выпуском новой версии изменения файла каким-то образом приостанавливаются, пока не будут получены результаты финального тестирования.
- Предоставьте механизм «красной кнопки» для принудительной отправки файла на установку до завершения тестирования. Поскольку «красная кнопка» снижает надежность, то, как правило, хорошей идеей будет оповещать о сбое как можно более заметно, например создавая отчет об этой ошибке с требованием более надежного решения в следующий раз.

«Красная кнопка» и тестирование

Вы можете предусмотреть механизм «красной кнопки» для прерывания или отключения тестов собранной версии перед установкой. Это будет означать, что все, кто в последний момент вносил изменения вручную, не узнают об ошибках, пока система мониторинга не сообщит о проблемах у реальных пользователей. Лучше дать тесту продолжать выполняться, связать событие досрочной передачи версии приложения на установку с будущими событиями

отсроченного тестирования и как можно оперативнее отметить и описать все неуспешные тесты. Таким образом, за принудительно установленной дефектной версией быстро последует другая (можно надеяться, более качественная). В идеале такая реализация «красной кнопки» автоматически повышает приоритет финальных тестов так, чтобы они могли опережать уже выполняющиеся рутинные инкрементальные проверки и нагрузочные тесты.

Интеграция

После тестирования отдельно взятого конфигурационного файла с целью снижения его возможного негативного влияния на надежность также важно рассмотреть интеграционное тестирование конфигурационных файлов. Содержимое конфигурационных файлов (с точки зрения тестирования) представляет собой потенциально вредоносное содержимое для интерпретатора, читающего конфигурацию [126](#). Интерпретируемые языки вроде Python часто используются в конфигурационных файлах, поскольку их интерпретаторы могут быть встроены в программные системы; для защиты от непреднамеренных ошибок в них существуют простые «песочницы».

Написание конфигурационных файлов на интерпретируемом языке рискованно, так как такой подход чреват скрытыми сбоями, которые трудно обнаружить. Поскольку загрузка конфигурации предполагает и выполнение прочитанного содержимого, невозможно определить естественный предел трудоемкости этого процесса. Этот вид интеграционного тестирования необходим в дополнение ко

всем прочим тестам. При этом следует тщательно контролировать длительность выполнения всех тестов, и тесты, которые не завершаются за заданное время, считаются неуспешными.

Если же конфигурация представляет собой текст с произвольным синтаксисом, каждая категория тестов нуждается в создании с нуля отдельного покрытия. Использование существующего синтаксиса, например YAML, в сочетании с хорошо протестированным парсером, скажем `safe_load` для Python, избавляет вас от части рутинной работы, связанной с конфигурационным файлом. Тщательный выбор синтаксиса и парсера может гарантировать, что у вас будет выдерживаться жесткий верхний предел времени выполнения всех операций загрузки. Однако тому, кто пишет конфигурации, приходится иметь дело с ошибками в данных и в их структуре, и наиболее простое решение состоит в отмене верхнего предела во время выполнения программы. К сожалению, эти стратегии бывают, как правило, плохо покрыты модульными тестами.

Использование ранее упоминавшихся двоичных протоколов обмена — протокольных буферов (protocol buffers, protobuf)^{[127](#)} — обеспечивает важное преимущество: структура данных (схема) определена заранее и во время загрузки проверяется автоматически, избавляя вас от еще большего количества рутины. При этом среда выполнения сохраняет ограничения.

Работа SR-инженера предполагает в том числе и написание служебных программных инструментов^{[128](#)} (если это не делает уже кто-то другой) и обеспечение тестового покрытия для проверки устойчивости системы. Все инструменты могут вести себя непредсказуемо из-за ошибок, не обнаруженных при тестировании, поэтому защиту рекомендуется строить многоуровневой. И когда один из них ведет себя неожиданно,

инженеры должны быть максимально уверены в том, что большинство остальных их инструментов работает корректно, чтобы иметь возможность справиться с последствиями такого поведения. Ключевую роль в обеспечении надежности сайта играют поиск всевозможных видов неправильного поведения и обеспечение того, что какой-либо тест (или протестированный валидатор входных данных другого инструмента) сообщает об этом поведении. Инструмент, который обнаружил проблему, может и не иметь возможности ее исправить или хотя бы остановить ее развитие, но он обязан как минимум сообщить о проблеме до того, как произойдет катастрофический сбой.

Рассмотрим сконфигурированный список всех пользователей (например, `/etc/passwd` на работающей на базе Unix машине, не подключенной к сети) и представим, что внесенное изменение непреднамеренно приводит к остановке парсера после обработки всего половины файла. Поскольку в результате не будут загружены лишь пользователи, созданные недавно, машина, скорее всего, продолжит работать без проблем и многие пользователи не заметят ошибки. Инструмент, который обслуживает домашние каталоги пользователей, может легко найти несоответствие списка существующих каталогов загруженному неполному списку пользователей и оперативно сообщит об этом. Роль этого инструмента заключается именно в том, чтобы сообщить о проблеме, и он не должен исправлять ее самостоятельно (путем удаления данных пользователей).

Зондирование системы в промышленной эксплуатации

Если тестирование обеспечивает корректность поведения для известных данных, а система мониторинга подтверждает ее для неизвестных, может показаться, что большинство источников рисков — как известных, так и неизвестных —

будут покрыты этой совокупностью тестирования и мониторинга. К сожалению, реальные риски зачастую куда более сложны.

Известные «хорошие» (корректные) запросы должны выполняться успешно, а известные «плохие» — завершаться с ошибкой. Удачное решение — реализовать покрытия и тех и других в виде интеграционных тестов. Тот же набор тестовых запросов может быть повторен и как финальный тест. Разбиение известных «хороших» запросов на группы, которые могут быть воспроизведены в условиях реальной промышленной эксплуатации и которые так воспроизводить нельзя, приводит к выделению трех категорий, таких как:

- известные «плохие» запросы;
- известные «хорошие» запросы, которые можно воспроизвести в промышленной среде;
- известные «хорошие» запросы, которые нельзя воспроизвести в промышленной среде.

Вы можете использовать запросы из каждой категории в качестве как интеграционных, так и финальных тестов. Большая часть этих тестов также может быть использована для мониторинга в качестве зондов.

Может показаться избыточным и, в принципе, бессмысленным разворачивать такую систему мониторинга, поскольку точно такие же запросы уже были испытаны путем выполнения их двумя другими способами. Однако эти два способа отличаются по некоторым критериям.

- При финальном тестировании, вероятно, собранный сервер был снабжен фронтендом и искусственным «тестовым»

бэкендом.

- При нагрузочном тестировании, возможно, проверяемый исполняемый файл работал с фронтеном, балансирующим нагрузку, и с отдельным масштабируемым устойчивым бэкендом.
- Фронтенды и бэкенды могут иметь независимые циклы выпуска новых версий. И, скорее всего, запланированные версии будут выходить с разной частотой (из-за адаптивных интервалов).

В результате зондирующие запросы мониторинга, выполняемые в промышленной среде, будут работать в конфигурации, которая не была протестирована ранее.

Эти запросы всегда должны работать корректно, но что будет означать их ошибка, если она все-таки произойдет? Либо API фронтенда (балансировщика нагрузки), либо API бэкенда (постоянного хранилища данных) для промышленного и тестового окружения оказались неэквивалентными. Если только вы не знаете заранее, почему эти окружения не эквивалентны, сайт считается нерабочим.

Та же программа для обновления промышленного окружения, которая постепенно заменяет приложение, постепенно заменяет и зондирующие запросы мониторинга, поэтому все четыре комбинации старых или новых зондов, которые отправляются старым или новым приложениям, продолжат генерироваться. Эта программа обновления может обнаружить, что одна из четырех комбинаций генерирует ошибки, и откатиться к последнему известному корректному состоянию. Обычно программа обновления предполагает, что каждое только что запущенное приложение является

неработоспособным в течение некоторого небольшого времени, пока не будет готово принимать большой объем пользовательского трафика. Если анализ зондов мониторинга включен в проверку готовности, обновление будет постоянно считаться неработоспособным и пользовательский трафик к новой версии приложения отправляться не будет. Обновление будет приостановлено, пока у инженеров не появится время и желание диагностировать сбой, а затем указать программе обновления выполнить откат.

Такой производственный тест с помощью зондов мониторинга защищает сайт и дает прямую, понятную обратную связь для инженеров. Эта обратная связь тем полезнее, чем раньше она доставляет инженерам информацию. Предпочтительно также, чтобы тесты были автоматизированы для масштабируемости механизмов оповещения инженеров.

Предположим, что каждый компонент имеет более старую версию, которая была заменена, и новую, которая выпускается (сейчас или очень скоро). Новая версия одного компонента может взаимодействовать со старой версией другого, что заставляет ее использовать устаревший API. Или же старая версия может взаимодействовать с новой версией хоста, используя API, который на момент выпуска старой версии еще не работал должным образом. Но теперь он работает, честно! Вам остается надеяться на лучшее — что эти тесты для будущей совместимости (которые работают как зонды мониторинга) имеют хорошее покрытие API.

Имитация бэкендов

При создании финальных тестов имитаторы бэкендов зачастую поддерживаются командой разработчиков сервиса,

с которым необходимо взаимодействовать, и просто включаются в список зависимостей сборки. Герметичный тест, выполняемый инфраструктурой тестирования, всегда объединяет имитационный бэкенд и тестовый фронтенд в одной точке сборки в истории контроля версий.

Эта зависимость может представлять собой герметичный исполняемый файл, и в идеале команда, которая его поддерживает, выпускает этот имитатор бэкенда вместе с выпуском основного приложения бэкенда и его тестов-зондов для мониторинга. Когда такая версия бэкенда собрана, может иметь смысл включить герметичные финальные тесты фронтенда (без имитатора бэкенда) в пакет новой версии фронтенда.

Ваша система мониторинга должна знать обо всех выпускаемых версиях с обеих сторон заданного интерфейса сервиса. Это позволяет гарантировать, что воссоздание любой из комбинаций двух версий и проверка прохождения теста, не потребуют трудоемкого дополнительного конфигурирования. Такой мониторинг не должен быть постоянным – вам нужно запустить лишь новые комбинации, являющиеся результатом выпуска новой версии обеими командами. Такие проблемы сами по себе не должны блокировать эту новую версию.

С другой стороны, средства автоматизации установки в идеале должны блокировать соответствующее промышленное обновление до тех пор, пока проблемные комбинации не

будут исключены. Аналогично средства автоматизации команды взаимодействующего сервиса могут постараться отвести трафик от экземпляров сервиса, имеющих проблемные комбинации, и обновить их.

Итоги главы

Тестирование — это одно из наиболее эффективных вложений в надежность программного продукта. Тестирование выполняется не раз и не два, а непрерывно на протяжении всего жизненного цикла проекта. Написание хороших тестов требует достаточно больших усилий, равно как и построение и поддержание инфраструктуры, которая поощряет хорошую культуру тестирования. Вы не можете исправить проблему до тех пор, пока не поймете ее, а в технике вы можете понять проблему, только измерив ее. Методологии и приемы, рассмотренные в этой главе, дают серьезную основу для количественной оценки ошибок и уровня достоверности в программных системах, чем помогают инженерам делать выводы о надежности ПО, когда оно уже выпущено и передано пользователям.

[103](#) В этой главе объясняется, как получить наибольший эффект от усилий, затрачиваемых на тестирование. Как только инженер сможет представить тесты, требуемые для заданной системы, в некотором обобщенном виде, остальная часть процесса будет одинакова для всех SRE-команд и к ней можно будет относиться как к общей инфраструктуре. Эта инфраструктура состоит из планировщика (для совместного доступа не связанных друг с другом проектов к общим ресурсам) и ряда «исполнителей» (которые testируют бинарные файлы в «песочнице», что позволяет не беспокоиться об их безопасности). Оба этих компонента инфраструктуры можно рассматривать как обычные службы, поддерживаемые SR-инженерами и наиболее похожие на хранилище масштаба кластера. Здесь эта тема подробно рассматриваться не будет.

[104](#) Более подробно об эквивалентности можно прочитать по следующей ссылке: <http://stackoverflow.com/questions/1909280/equivalence-class-testing-vs-boundary-value-testing>.

[105](#) Ошибки, выявленные на этапе тестирования, не требуют восстановления как такового, так как система еще не работает с реальными данными. — *Примеч. пер.*

[106](#) Этот уровень тестирования называют также «тестированием компонентов», но в данной книге термин «компоненты» применяется обычно к более крупным структурным единицам. — *Примеч. пер.*

[107](#) См. <https://google.github.io/dagger/>.

[108](#) «Мок» (англ. mock) — используемая при тестировании и отладке вставка-имитатор для замещения более сложного компонента, обращение к которому неудобно или невозможно. При этом «мок» воспроизводит упрощенную логику имитируемого компонента, чем отличается от простейшей заглушки — «стаба» (англ. stub). — *Примеч. пер.*

[109](#) «Песочница» (англ. sandbox) — искусственное замкнутое окружение, создаваемое для выполнения некоторой программы или программ. Все результаты работы этих программ ограничиваются рамками «песочницы». — *Примеч. пер.*

[110](#) Типичная практика состоит в том, что обновление затрагивает сначала только 0,1 % пользовательского трафика, а затем эта величина возрастает на порядок каждые 24 часа с одновременным изменением географии обновлений (во второй день 1 %, в третий — 10 %, в четвертый — 100 %).

[111](#) Практика канареичного тестирования выглядит не вполне честно по отношению к пользователям, которые в роли «канареек» рискуют временем и данными. Однако в ряде случаев воспроизвести все особенности реальных сценариев в изолированной тестовой среде практически невозможно. — *Примеч. пер.*

[112](#) Например, предполагая, что мы имеем 24-часовой интервал непрерывного экспоненциального роста от 1 до 10 %, получим

$$\frac{86\,400}{\ln \frac{0,1}{0,01}} = 37\,523$$

секунды, или примерно 10 часов 25 минут.

[113](#) Мы используем понятие «порядок» в смысле нотаций «“O” большое» и «“o” малое». Для получения более подробной информации пройдите по ссылке https://en.wikipedia.org/wiki/Big_O_notation.

[114](#) Если вы хотите получить более подробную информацию по этой теме, рекомендуем вам прочесть книгу [Bland, 2014], написанную нашим бывшим коллегой Майком Бландом.

[115](#) Можно отметить парадоксальный эффект: чем больше ошибок в программе изначально, тем лучше ее покрытие тестами такого рода. — *Примеч. пер.*

[116](#) Сейчас чаще говорят о «непрерывной интеграции», имея в виду практически то же самое: сборка и затем тестирование инициируются не как отдельные стадии, а после любых изменений в файлах проекта. Для сложного проекта это позволяет более оперативно выявлять дефекты. — *Примеч. пер.*

[117](#) См. <https://github.com/google/bazel>.

[118](#) С графиками зависимостей работали еще утилита make и ей подобные, но здесь речь идет о более высоком уровне автоматизации процесса и о включении в него также и тестирования. Под воспроизводимостью, вероятно, авторы имеют в виду возможность повторять такую частичную сборку (и тестирование) быстрее и чаще. — *Примеч. пер.*

[119](#) Например, тестируемый код служит оберткой для нетривиального API и обеспечивает более простую абстракцию с обратной совместимостью. Этот API, который привыкли считать синхронным, становится асинхронным и возвращает future — особую разновидность результата, значение которого не всегда определено. Ошибка аргументов вызова по-прежнему приводит к исключению, но лишь после того, как значение future перестанет быть неопределенным. Тестируемый код передает возвращенный из API результат вызывающей стороне напрямую, не анализируя его. При этом многие случаи ошибочных аргументов могут остаться незамеченными.

[120](#) В этом разделе рассматриваются инструменты службы SRE, которые должны быть масштабируемыми. Однако SR-инженеры разрабатывают и используют также и инструменты, которым масштабируемость не нужна. Такие инструменты тоже необходимо тестировать, но они в этом разделе не рассматриваются. Поскольку по характерным проблемам они схожи с пользовательскими приложениями, для них могут применяться аналогичные стратегии тестирования.

[121](#) См. <https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey>.

[122](#) См. <https://github.com/aphyr/jepsen>.

[123](#) Даже если запуск теста повторяется с одинаковым начальным состоянием генератора случайных чисел и сигналы для процессов генерируются в одном и том же порядке, синхронизация между этими сигналами и тестовым пользовательским трафиком не соблюдается. Поэтому нельзя гарантировать, что воспроизведены те же пути выполнения кода, которые наблюдались ранее.

[124](#) См. <http://xkcd.com/303/>.

[125](#) Возможно, набранных с помощью Mechanical Turk или похожих служб.

[126](#) Имеется в виду, что файл содержит не данные (числовые, текстовые и т.п.), а фрагменты кода на некотором языке. Фактически это можно рассматривать как часть программного кода системы, который, однако, устанавливается в обход правил и ограничений, действующих для исполняемых файлов. — *Примеч. пер.*

[127](#) См. <https://github.com/google/protobuf>.

[128](#) Не потому, что программисты не должны их писать. Инструменты, которые нарушают обычную иерархию программных средств и охватывают несколько уровней абстракции, с командами системных инженеров обычно соотносятся несколько лучше, чем с большинством команд разработчиков.

18. Разработка ПО службой SRE

*Авторы — Дейв Хелструм и Триша Вейр
при участии Эвана Леонарда и Курта Делимона
Под редакцией Кавиты Джулиани*

Попросите кого-либо назвать программный продукт компании Google, и вы услышите название одного из пользовательских продуктов вроде Gmail или Maps; кто-то даже может упомянуть инфраструктурные решения вроде Bigtable или Colossus. В действительности же существует также огромный объем ПО «за кулисами», которое пользователи никогда не видят. Некоторые из этих программных продуктов разработаны внутри Google SRE.

Производственная среда компании Google по некоторым показателям одна из самых сложных систем, построенных человеком. SR-инженеры имеют из первых рук опыт работы с хитросплетениями производственной среды. Это делает их уникальным образом подходящими для разработки необходимых инструментов для решения внутренних задач в ситуациях, связанных с обеспечением доступности систем в промышленной эксплуатации. Большая часть этих инструментов направлена на обеспечение доступности сервисов и низкой задержки отклика, но они могут принимать различные формы, например механизмы развертывания исполняемых файлов, системы мониторинга или среды разработки на основе динамической компоновки сервера. В целом, эти разработанные службой SRE инструменты представляют собой законченные программные проекты, которые отличаются от одноразовых решений и быстрых «костылей», а в основе образа мыслей разработавших их SR-инженеров лежит программный продукт, и это заставляет

принимать в расчет как запросы внутренних потребителей, так и перспективные направления развития.

Почему так важна разработка ПО внутри службы SRE

Для большинства промышленных систем компании Google требуется внутреннее ПО, поскольку лишь немногие сторонние инструменты могут работать в необходимых нам масштабах. История успешных проектов нашей компании убедительно показала нам преимущества разработки ПО непосредственно внутри службы SRE.

SR-инженеры находятся в уникальном положении, позволяющем эффективно разрабатывать внутреннее ПО, по нескольким причинам.

- Широта и глубина знаний SR-инженеров о продуктах Google позволяет им разрабатывать и создавать ПО с учетом масштабируемости, предсказуемой и управляемой деградации во время сбоев, а также возможностей взаимодействия с другой инфраструктурой или инструментами.
- Поскольку SR-инженеры сами работают в той же области, они легко могут понять постановку задачи и требования для разрабатываемого инструмента.
- Прямая связь с потенциальными пользователями — коллегами по SRE — позволяет получать честную и подробную обратную связь. Создавая инструмент для внутренней аудитории, хорошо знакомой с предметной областью, команда разработки может быстрее выпустить продукт и затем обновлять его. Внутренние пользователи с

бо́льшим пониманием относятся к минималистичному интерфейсу и другим недостаткам альфа-версии продукта.

С чисто прагматической точки зрения компания Google только выигрывает от того, что SR-инженеры разрабатывают ПО. Согласно плану темпы роста сервисов, поддерживаемых службой SRE, опережают темпы расширения ее штатов; один из основных принципов SRE заключается в том, что «численность команды не должна увеличиваться прямо пропорционально росту сервисов». Линейный рост команды при экспоненциальном росте поддерживаемых ею сервисов требует постоянного внедрения решений по автоматизации и усилий по упрощению инструментов, процессов и других аспектов сервиса для эффективного повседневного функционирования систем. Имеет смысл участие людей с опытом эксплуатации промышленных систем в разработке инструментов, служащих достижению требуемых целевых показателей доступности и задержки отклика.

С другой стороны, отдельные SR-инженеры, а также вся служба SRE тоже выигрывают от использования такого подхода.

Полноценные проекты по разработке ПО внутри службы SRE дают возможности карьерного роста SR-инженерам, а также возможность реализовать себя тем из них, кто желает поддерживать свои навыки программирования. Работа над долгосрочными проектами служит столь необходимым противовесом для дежурств и авралов, а также позволяет почувствовать удовлетворение от работы тем инженерам, которые хотели бы заниматься не только проектированием систем, но и разработкой ПО.

Помимо разработки средств автоматизации и иных мер по снижению нагрузки на SR-инженеров, проекты по разработке

ПО могут принести и другую пользу службе SRE, привлекая и помогая сохранять инженеров с широким спектром навыков. Желание иметь в команде таких специалистов особенно уместно для SRE, где сочетание знаний из разных областей и разных подходов позволяет избежать «слепых пятен». Поэтому в компании Google всегда стремится формировать команды SRE-инженеров как специалистами в традиционной разработке ПО, так и теми, кто имеет навыки сопровождения систем.

Пример Auxon: история проекта и предметная область

В этом примере рассматривается Auxon — мощный инструмент, разработанный в Google SRE и предназначенный для автоматизации планирования производительности [129](#) «промышленных» сервисов Google. Чтобы лучше понять, как был создан Auxon и каковы решаемые им задачи, мы рассмотрим предметную область, связанную с планированием производительности, а также сложности, с которыми сталкиваются разработчики при использовании традиционного подхода как в Google, так и в других компаниях. Значение терминов «сервис» и «кластер» в компании Google см. в главе 2.

Традиционное планирование производительности

Существует множество тактик планирования вычислительных ресурсов (см. [Hixson, 2015a]), но большая часть подходов сводится к циклу, который выглядит примерно так.

1. *Спрогнозировать потребности.* Сколько ресурсов потребуется? Когда и где понадобятся эти ресурсы?

- Лучшие данные, доступные сегодня, используются для планирования ситуаций, возможных в будущем.
- Обычно охватывается период от нескольких кварталов до нескольких лет.

2. *Разработать план ввода и распределения ресурсов.* Учитывая полученный прогноз, как мы можем наилучшим образом удовлетворить спрос с помощью дополнительных ресурсов? Сколько ресурсов потребуется и на каких площадках?
3. *Проанализировать и утвердить план.* Адекватен ли этот прогноз? Соответствует ли этот план требованиям к бюджету, продукту и реализации?
4. *Провести развертывание и конфигурирование ресурсов.* Как только ресурсы окажутся в нашем распоряжении (потенциально это будет происходить поэтапно, на протяжении некоторого периода времени), какие сервисы смогут воспользоваться этими ресурсами? Как можно обеспечить наибольшую пользу для сервисов от типичных низкоуровневых ресурсов (вроде ЦП, диска и т.д.)?

Нужно отметить, что такое планирование — это бесконечный цикл: предположения меняются, доставка и установка ПО задерживаются, а бюджеты урезаются, что приводит к постоянным пересмотрам *Плана*. Влияние каждого пересмотра распространяется на все планы всех последующих кварталов. Например, недовыполнение плана в текущем квартале должно быть скомпенсировано в последующих. При традиционном планировании производительности в качестве основного «двигателя» выступает уровень спроса, и уровень

снабжения изменяется вручную так, чтобы соответствовать изменениям спроса.

Врожденная неустойчивость

План выделения ресурсов, формируемый при традиционном планировании выделения ресурсов, может быть легко нарушен любым небольшим изменением. Например:

- снижается эффективность сервиса, и ему требуется больше ресурсов для того, чтобы обслужить то же количество запросов;
- растет уровень популярности у клиентов, что приводит к повышению спроса на ресурсы;
- срываются сроки поставки нового вычислительного кластера;
- решение об изменении целевого показателя производительности приводит к изменению количества установленных экземпляров сервиса и требуемых для них ресурсов.

Небольшие изменения требуют проверки всего плана выделения ресурсов для того, чтобы убедиться, остается ли план валидным; более крупные изменения (вроде задержки поставки оборудования или изменений в стратегии продукта) потенциально потребуют строить план заново. Срыв поставки для одного кластера может повлиять на отказоустойчивость или требования к задержке отклика многих сервисов: выделение ресурсов в других кластерах придется увеличивать, чтобы скомпенсировать этот срыв. Эти и другие изменения могут оказать влияние на весь план.

Следует также иметь в виду, что план производительности для любого заданного квартала (или другого промежутка времени) основывается на ожидаемом результате выполнения планов предыдущих кварталов, а это означает, что изменения в любом квартале потребуют корректировок в последующих кварталах.

Трудоемкость и неточность

Для многих команд процесс сбора данных, необходимых для прогнозирования спроса, оказывается слишком медленным и подверженным ошибкам. А когда наступает время для расчета производительности, соответствующей запросам, не все ресурсы одинаково хороши. Например, если по требованиям к задержке отклика сервис должен быть размещен на том же континенте, что и его пользователи, получение дополнительных ресурсов в Северной Америке не поможет скомпенсировать их нехватку в Азии. Каждый прогноз имеет ограничения — граничные условия, которым он должен соответствовать; ограничения неразрывно связаны с целями, то есть теми характеристиками сервиса, которые стремится получить пользователь, и это рассматривается в следующем разделе.

Преобразование запросов ресурсов с учетом ограничений в план фактического их выделения из числа доступных также происходит медленно: задача оптимального размещения в ограниченном пространстве (в математике ее обычно называют «задачей об укладке ранца». — Примеч. пер.), а также поиск решений, соответствующих ограниченному бюджету, очень сложные и трудоемкие, если выполнять их вручную.

Из-за этого у вас уже может сложиться мрачная картина, но ситуацию усугубляет еще и то, что требуемые для этого процесса инструменты, как правило, ненадежны или

громоздки. Электронные таблицы очень плохо масштабируются и почти не предоставляют средств для проверки на ошибки. Данные устаревают, и отслеживание изменений становится затруднительным. Команды зачастую должны делать упрощающие предположения и снижать сложность своих требований только для того, чтобы сделать задачу поддержания достаточной производительности подъемной.

Когда владельцы сервисов сталкиваются с необходимостью выделить нужные ресурсы с целью обеспечения требуемой производительности для серий запросов и при этом соблюсти все возможные ограничения, возникает дополнительная неточность. Оптимальная укладка — это NP -сложная задача, которую людям трудно решить вручную. Помимо этого, запрос ресурсов для сервиса зачастую представляет собой негибкий набор требований: X ядер в кластере Y . Причины, по которым требуется именно X ядер или именно кластер Y , а также любые возможные отклонения от этого запроса, теряются по мере достижения запросом человека, который пытается втиснуть все эти запросы в заданный объем ресурсов.

В результате вы потратите кучу сил на то, чтобы найти алгоритм выделения ресурсов, который в лучшем случае будет приблизительным. Этот процесс слишком чувствителен к изменениям, а диапазон значений оптимального решения неизвестен.

Наше решение: планирование производительности, основанное на целях

Укажите требования, а не реализацию.

Многие команды Google перешли на использование подхода, который мы называем планированием

производительности, основанном на целях. Главная идея этого подхода заключается в том, чтобы формализовать и запрограммировать зависимости и требуемые параметры (цели) пользователей сервиса, а затем на основе этого автоматически генерировать план выделения ресурсов, в котором уже будет детально указано, какие ресурсы и в каком кластере может использовать каждый сервис. Если спрос, предложение или требования сервиса изменяются, мы легко можем генерировать новый план соответственно изменившимся параметрам, который будет наилучшим образом представлять распределение ресурсов.

Как только мы будем обладать реальными данными о требованиях и гибкости сервиса, план производительности станет гораздо более гибким и мы сможем находить оптимальное решение, соответствующее наибольшему числу параметров. Перепоручив поиск оптимального решения компьютерам, мы значительно снизили уровень рутинной ручной работы, и владельцы сервисов смогли сосредоточиться на более приоритетных задачах вроде соответствия SLO, зависимостях производства и требованиях к инфраструктуре сервиса вместо того, чтобы выпрашивать низкоуровневые ресурсы.

Кроме того, алгоритмизация и использование вычислительных методов для поиска оптимальной реализации распределения ресурсов в соответствии с целями обеспечивает более высокую точность, что в итоге тоже дает экономию в масштабах организации. Задача об оптимальной укладке по-прежнему остается не решенной окончательно, поскольку некоторые ее разновидности все еще считаются *NP*-сложными; однако существующие алгоритмы вполне эффективны для известных оптимальных решений.

Планирование производительности, основанное на целях

Цель — это то, чем владелец сервиса аргументирует запрашиваемые для него параметры функционирования. Переход от запросов конкретных ресурсов к истинным целям этих запросов в ходе составления плана требует нескольких уровней абстракции. Рассмотрим следующую их цепочку.

1. «*Мне нужно 50 ядер в кластерах X, Y и Z для сервиса Foo*». Так выглядит явный запрос ресурсов. Но... зачем нам нужно именно столько ресурсов именно в этих кластерах?
2. «*Мне нужен 50-ядерный ресурс в любых трех кластерах в географическом регионе YYY для сервиса Foo*». Этот запрос чуть более гибок, и его проще осуществить, несмотря на то что он еще не объясняет цель таких требований. Но... почему нам нужно именно столько ресурсов, и почему три кластера?
3. «*Мне нужно соответствовать спросу на сервис Foo в каждом географическом регионе и иметь избыточность N + 2*». Теперь запрос еще более гибок, и мы можем понять на более «человеческом» уровне последствия того, что сервис Foo не получит свои ресурсы. Но... зачем нам нужна избыточность $N + 2$ для сервиса Foo?
4. «*Я хочу, чтобы сервис Foo работал на пяти девятках надежности*». Это требование более абстрактно, и последствия несоответствия требованиям становятся очевидными: пострадает надежность. Здесь мы имеем еще большую гибкость: возможно, избыточность $N + 2$ окажется недостаточной или неоптимальной, и найдется более подходящий план развертывания сервиса.

На каком уровне следует формулировать цели для планирования производительности, основанного на целях? В идеале мы должны поддерживать все уровни для тех сервисов, которые выигрывают от формулирования требований в виде целей вместо конкретных реализаций.

По опыту Google, для большинства сервисов выгоден выход на уровень 3: мы получаем достаточную гибкость, а результаты решений формулируются на более высоком уровне и их проще понять. Отдельные сложные сервисы могут тяготеть к уровню 4.

Предпосылки для целей

Что нужно знать о сервисе, чтобы определить цели? Давайте познакомимся с зависимостями, показателями производительности и приоритизацией.

Зависимости

Сервисы Google зависят от множества других инфраструктурных и пользовательских сервисов, и эти зависимости значительно влияют на расположение сервиса. Например, представим пользовательский сервис Foo, который зависит от Bar — инфраструктурного сервиса хранения данных. Сервис Foo предъявляет требование к расположению сервиса Bar — задержка отклика не должна превышать 30 миллисекунд. Это требование имеет важные последствия для размещения сервисов Foo и Bar, и при планировании производительности, основанном на целях, мы должны принимать во внимание эти ограничения.

Помимо этого, зависимости могут быть вложенными: основываясь на предыдущем примере предположим, что сервис Bar имеет собственные зависимости от сервисов Baz,

более низкоуровневого сервиса распределенного хранения данных, и Qux, сервиса управления приложениями. Теперь расположение сервиса Foo зависит от расположения сервисов Bar, Baz и Qux. Набор зависимостей может быть общим для нескольких целей, возможно имея различные допущения.

Показатели производительности

Спрос на один сервис может влиять на спрос на один или несколько других сервисов. Понимание цепочки зависимостей помогает более масштабно взглянуть на задачу распределения ресурсов, но нам все еще нужно больше информации об ожидаемом уровне их использования. Как много вычислительных ресурсов сервиса Foo нужно для обслуживания N пользовательских запросов? Сколько мегабайт в секунду будет отправлено сервису Bar для каждого N запросов к сервису Foo?

Показатели производительности служат своего рода «kleem» для зависимостей. Они помогают выполнить переход от одного высокоуровневого типа ресурсов или более к одному низкоуровневому типу или более. Получение соответствующих показателей для сервиса может предполагать также тестирование нагрузки и наблюдение за использованием ресурсов.

Приоритизация

Ограничения в ресурсах неизбежно приведут к компромиссам и непростым решениям: какими требованиями можно пожертвовать, если ресурсов не хватает?

Возможно, избыточность $N + 2$ для сервиса Foo более важна, нежели избыточность $N + 1$ для сервиса Bar. Или, возможно,

запуск функциональности X менее важен, чем избыточность $N + 0$ для сервиса Baz.

Планирование, основанное на целях, делает эти решения прозрачными, открытыми и целостными. Ограничения по ресурсам влекут за собой такие же компромиссы, но слишком часто приоритизация бывает ситуативной и непрозрачной для владельцев сервисов. А планирование, основанное на целях, позволяет проводить приоритизацию с любым требуемым уровнем детализации.

Знакомимся с Auxon

Auxon — это решение компании Google для планирования производительности, основанного на целях, и распределения ресурсов. Auxon — самый заметный пример программного продукта, разработанного в Google SRE: эта система создавалась небольшой группой программистов и техническим менеджером SRE в течение двух лет. Это идеальный пример того, как разработка ПО может органично вписаться в работу службы SRE.

Система Auxon активно применяется для планирования использования многомиллионных вычислительных ресурсов компании Google. Она стала критически важным компонентом планирования производительности для ряда подразделений компании Google.

Auxon предоставляет способ сбора описаний требований сервиса к ресурсам, основанных на целях, а также его зависимостей. Эти пользовательские цели выражены как требования к обеспечению сервиса ресурсами. Требования могут быть выражены как запросы, например: «Мой сервис должен иметь избыточность $N + 2$ для каждого континента» или «Фронтенд-серверы должны находиться не далее чем в 50

миллисекунд от бэкенд-серверов». Auxon собирает эту информацию либо с помощью языка пользовательских конфигураций, либо с помощью программного интерфейса (API), преобразуя сформулированные человеком цели в понятные машине ограничения. Требования могут иметь приоритеты, это полезно в ситуации, когда ресурсов недостаточно для того, чтобы выполнить все требования, и приходится прибегать к компромиссам. Эти требования (которые, собственно, и составляют цель) в итоге представляются внутри системы как гигантская задача частично-целочисленного или линейного программирования. Auxon решает эту задачу и использует полученное решение для формирования плана распределения ресурсов.

Рисунок 18.1 с последующими разъяснениями показывает основные компоненты Auxon.

Данные о производительности описывают масштабирование сервиса: сколько единиц зависимости Z используется для каждой единицы спроса X в кластере Y ? Эти данные о масштабировании могут быть получены несколькими способами в зависимости от стадии жизненного цикла конкретного сервиса. Для одних сервисов есть результаты нагрузочного тестирования, для других же предположения о масштабировании делаются на основе ранее наблюдавшейся производительности.

Данные о прогнозируемом спросе для каждого сервиса описывают тенденцию использования для спрогнозированных сигналов запроса. Некоторые сервисы узнают об уровне использования их в будущем через прогнозы спроса — прогнозы количества запросов в секунду, отдельно по континентам. Прогнозы спроса существуют не для всех сервисов: некоторые сервисы (например, сервисы хранения

данных вроде Colossus) получают сведения о спросе на них от зависимых сервисов.

Доступные ресурсы — это данные о доступности базовых ресурсов: например, количество машин, которые можно будет использовать в заданный момент времени в будущем. В терминологии линейного программирования этот показатель служит верхней границей роста сервисов, который также влияет на их размещение. В конечном счете мы хотим использовать наши ресурсы наилучшим образом, насколько нам это позволит основанное на целях описание всей группы сервисов.

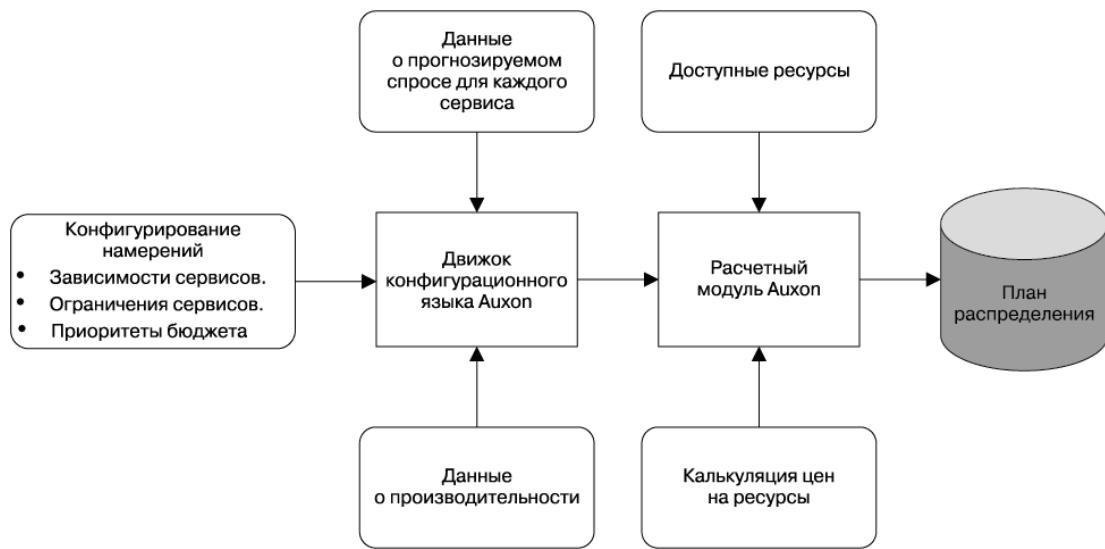


Рис. 18.1. Основные компоненты системы Auxon

Калькуляция цен на ресурсы предоставляет данные о стоимости базовых ресурсов. Например, стоимость машин в глобальной системе может значительно различаться в зависимости от стоимости площадей/энергии на различных площадках. В терминологии линейного программирования цены влияют на общую рассчитанную стоимость, выступающую в качестве целевого значения, которое мы хотим минимизировать.

Конфигурация цели — это способ, с помощью которого информация о намерениях попадает в Auxon. Она определяет, из чего состоит сервис, а также отношения между сервисами. Конфигурация выступает в качестве слоя, обеспечивающего связь между всеми остальными компонентами. Она разработана так, чтобы быть удобной для понимания и изменения людьми.

Интерпретатор («движок») языка конфигураций Auxon действует на основе информации, полученной из конфигурации. Этот компонент формирует в понятном для машины виде запрос (в формате «протокольных буферов»), который может быть понят расчетным блоком Auxon. Он выполняет небольшую проверку доступности конфигурации и ведет себя как шлюз между конфигурируемым человеком описанием целей и понимаемым машиной запросом по оптимизации.

Расчетный модуль Auxon — это мозг системы. Он формирует гигантскую задачу частично-целочисленного или линейного программирования на основе запроса по оптимизации, полученного от интерпретатора конфигураций. При его разработке была заложена высокая масштабируемость, что позволяет ему работать в кластерах Google на сотнях или даже тысячах машин. В дополнение к инструментарию частично-целочисленного линейного программирования, в расчетном модуле Auxon также имеются компоненты, которые работают с задачами планирования, управления пулом рабочих процессов и спуска по деревьям решений.

План распределения ресурсов — это результат работы расчетного модуля Auxon. Он указывает, какие ресурсы могут быть получены конкретными сервисами в заданных локациях. Этот план представляет собой рассчитанную детальную реализацию исполнения требований, описанных на основе

целей. План также включает в себя информацию о тех требованиях, которые не удалось выполнить, например, из-за нехватки ресурсов или из-за того, что существует другой сервис, претендующий на те же ресурсы.

От требований до реализации: достижения и полученные уроки

Изначально Auxon был задуман SR-инженерами и техническими менеджерами, которым независимо друг от друга команды ставили задачу планирования производительности для крупных фрагментов инфраструктуры Google. Выполняя планирование вручную с помощью электронных таблиц, они поняли неэффективность такого подхода и пути его улучшения с помощью автоматизации, а также какая функциональность может при этом понадобиться.

В течение всего времени разработки Auxon команда SRE, стоящая за продуктом, продолжала принимать участие в жизни промышленно эксплуатируемых систем. Команда участвовала в дежурных сменах на сопровождении нескольких сервисов, а также в обсуждениях проектов этих сервисов и техническом руководстве. Благодаря этому непрекращающемуся взаимодействию команда могла оставаться в курсе того, что происходит в промышленном окружении: они выступали и как потребитель, и как разработчик собственного продукта. Если продукт оказывался неудачным, это непосредственно влияло на саму команду. Запрашиваемая функциональность была обоснована непосредственно опытом из первых рук. Такой опыт не только позволил ощутить свой вклад в успех продукта, но и способствовал укреплению доверия к нему внутри службы SRE.

Упрощенная модель. Не концентрируйтесь на идеальном и чистом решении, особенно если масштабы проблемы еще не

известны. Просто запускайте свой продукт поскорее, работайте с ним и дорабатывайте его.

В любом достаточно сложном проекте разработки ПО обычно приходится иметь дело с неопределенностью: как должен быть спроектирован компонент или как должна быть решена задача. Разработчики Auxon столкнулись с такой неопределенностью на ранних этапах проекта, поскольку для членов команды мир линейного программирования представлял собой неисследованную территорию. Ограничения линейного программирования, которое должно было стать ядром продукта, не были известны досконально. Чтобы преодолеть опасения команды, которая должна была работать с этой недостаточно понятой зависимостью, мы решили создать для начала упрощенный расчетный модуль (мы назвали его *Stupid Solver* — «тупой вычислитель»), который применял простую эвристику для выстраивания сервисов согласно пользовательским требованиям. Хотя *Stupid Solver* никогда не смог бы дать действительно оптимальное решение, он помог команде понять, что создать Auxon вполне реально, даже несмотря на то, что поначалу у нас не все будет получаться.

При развертывании упрощенной модели, призванной ускорить разработку, важно было заложить возможность вносить в нее улучшения в будущем и возвращаться к ней. В случае *Stupid Solver* весь интерфейс решающей программы был абстрагирован внутри Auxon для того, чтобы его содержимое могло быть изменено в будущем. В дальнейшем, когда мы уже были уверены в унифицированной модели линейного программирования, оставалось лишь переключиться на использование чего-то «поумнее».

Требования продукта Auxon также содержали ряд неопределенностей. Создание ПО с запутанными

требованиями может оказаться крайне сложным, но эта неопределенность не смогла нас остановить. Она послужила нам стимулом добиваться того, чтобы ПО было одновременно достаточно обобщенным и модульным. Например, одной из целей проекта Auxon являлась интеграция с автоматическими системами внутри Google для того, чтобы реализовывать план распределения ресурсов непосредственно в промышленной среде (распределение ресурсов, а также включение/выключение/изменение размера сервисов по мере необходимости). Однако в то время мир автоматических систем находился в постоянном движении, использовалось огромное множество подходов. Вместо того чтобы попытаться разработать специализированные решения, позволяющие Auxon работать с каждым конкретным инструментом, мы сделали план распределения ресурсов универсальным. Это позволило нашим системам автоматизации работать с ним собственными средствами. Такой «агностический» подход оказался ключевым в деле привлечения и адаптации новых пользователей, поскольку он не требовал от них менять используемые инструменты.

Мы также воспользовались модульным дизайном, чтобы справиться с запутанными требованиями при сборке модели производительности машины внутри Auxon. Данные, описывающие ресурсы (например, ЦП) будущей машинной платформы, были пугающими, но наши пользователи хотели иметь способ моделирования различных сценариев по использованию мощности машины. Мы абстрагировались от данных машины, скрыв их за одним интерфейсом, что позволило пользователям менять разные модели конфигураций будущей машины. Позже на основе постоянно уточняющихся требований мы развили идею такой модульности еще дальше и создали простую библиотеку

моделирования производительности машин, которая работала в рамках нашего интерфейса.

Если бы необходимо было выбрать из нашего примера всего одну идею, она заключалась бы в том, что при разработке ПО внутри службы SRE старый девиз «запускайте и дорабатывайте» особенно актуален. Не ждите идеального дизайна; вместо этого при проектировании и разработке представляйте общую картину. Когда вы столкнетесь с неопределенностью, разрабатывайте ПО так, чтобы оно было достаточно гибким на случай, если процесс или стратегия изменится на более высоком уровне, чтобы не понести огромные затраты на переписывание ПО. Но в то же время не забывайте и о более приземленном: обеспечивайте наличие у общих решений специализированных реализаций для конкретных реальных задач, демонстрируя таким образом практичность дизайна.

Повышаем осведомленность и способствуем внедрению

Как и в случае с другими проектами, разработка ПО службой SRE должна учитывать знания о пользователях и требованиях. Внедрению ПО содействуют его полезность, эффективность и способность как служить целям повышения надежности продуктов Google, так и улучшать жизнь SR-инженеров. Процесс «социализации» продукта и принятия его внутри организации является ключевым для успеха проекта.

Не стоит недооценивать усилия, затрачиваемые для повышения осведомленности о вашем программном продукте и заинтересованности в нем, — одной презентации или электронного письма будет недостаточно. «Социализация» внутренних программных инструментов для крупной аудитории требует:

- понятного и неизменного подхода;
- рекламы среди пользователей;
- сотрудничества со стороны инженеров и менеджеров, которым вы будете демонстрировать полезность вашего продукта.

При создании продукта важно учитывать точку зрения пользователя. Инженер может не иметь времени или желания разбираться в исходном коде для того, чтобы понять, как использовать инструмент. Несмотря на то что внутренние клиенты зачастую более терпимы к шероховатостям и ранним альфа-версиям, нежели внешние, вам все же нужно предоставлять документацию. SR-инженеры — занятые люди, и, если ваше решение слишком сложное или запутанное, они напишут собственное.

Задавайте уровень ожиданий

Когда за разработку продукта берется инженер, давно знакомый с предметной областью, воображение охотно рисует утопическую картину результатов его работы. Однако важно различать амбициозные «максимальные» цели и минимальные критерии успеха (критерии минимально жизнеспособного продукта — *minimum viable product*). Проекты могут потерять доверие и провалиться, если было обещано дать слишком много за короткий промежуток времени; в то же время может быть трудно убедить команды попробовать что-то новое, если этот продукт не обещает быть достаточно полезным. Демонстрация постепенного прогресса путем выпуска небольших обновлений повышает уверенность пользователей в том, что ваша команда сможет создать полезное ПО.

В случае Auxon мы достигли баланса, создав долгосрочный план с возможностью внесения в него оперативных исправлений. Мы пообещали командам следующее.

- Любые усилия по внедрению проекта или совершенствованию его конфигураций будут немедленно давать отдачу в виде избавления от необходимости оптимизировать распределение запрашиваемых ресурсов вручную.
- По мере разработки дополнительной функциональности для Auxon мы будем использовать одни и те же конфигурационные файлы и предоставлять новые, гораздо более действенные способы снижения затрат и другие преимущества. План проекта позволил владельцам сервисов быстро определить, будет ли разработана необходимая им функциональность в ранних версиях. В то же время итерационный подход к разработке Auxon позволил менять приоритеты и вводить новые этапы развития продукта.

Определите наиболее подходящих клиентов

Команда, разрабатывающая Auxon, понимала, что однозначное решение не может удовлетворить всех; многие более крупные команды уже имели собственные решения по планированию производительности, которые достаточно хорошо работали. Несмотря на то что их собственные инструменты не были идеальны, эти команды не испытывали значительных проблем с планированием производительности и не чувствовали необходимости пробовать новый инструмент, особенно если он находится в альфа-версии.

Ранние версии Auxon были нацелены на команды, которые еще не наладили свой процесс планирования

производительности. Поскольку время на конфигурирование пришлось бы потратить в любом случае, выбери они существующее решение или наш подход, эти команды были заинтересованы в том, чтобы воспользоваться новейшим инструментом. Своих первых успехов система Auxon добилась благодаря тому, что этим командам была продемонстрирована полезность проекта, и потребители в итоге стали его защитниками.

Оценка и подсчет выгод от применения продукта оказались полезными и в дальнейшем; когда мы брались за очередное направление в нашем бизнесе, команда проводила тщательный анализ, детально сравнивая результаты до и после внедрения системы. Экономия времени и снижение человеческой рутины еще больше подогрели интерес других команд к Auxon.

Поддержка пользователей

Даже несмотря на то, что основной аудиторией для разработанного службой SRE ПО являются ТРМ и инженеры, имеющие высокую техническую подготовку, любое достаточно инновационное ПО требует определенной программы обучения для новых пользователей. Не бойтесь предоставить специальную поддержку для новых пользователей, чтобы помочь им освоиться. Иногда автоматизация влечет за собой эмоциональные вопросы, например опасение, что чья-то должность будет заменена автоматически выполняемым скриптом. Работая с первыми пользователями персонально, вы сможете помочь им справиться с таким страхом и продемонстрировать, что вместо рутины, связанной с выполнением утомительной работы вручную, команда получит конфигурацию, процессы и в конечном итоге будет обладать всеми результатами их технической работы. Пример

довольных первых пользователей послужит убеждению последующих.

Более того, поскольку команды SR-инженеров компании Google распределены по всему земному шару, первые пользователи, выступающие в поддержку вашего проекта, особенно полезны, поскольку они могут стать локальными экспертами для других команд, которым было бы интересно попробовать ваш проект.

Правильный выбор уровня разработки

Идея, которую мы назвали «агностицизмом», — написание обобщенного ПО, которое может принимать множество источников данных в качестве входной информации, — является основным принципом дизайна Auxon. Агностицизм означает, что от клиентов не требуется использовать строго определенные инструменты для того, чтобы применять фреймворк. Такой подход позволил Auxon оставаться полезным, даже когда его начали использовать команды с иным инструментарием. Мы обратились к потенциальным пользователям: «Приходите как есть; мы будем работать с тем, что вы предложите». Избегая чрезмерной подстройки под одного-двух крупных пользователей, мы достигли того, что наш проект стал пользоваться большим успехом в организации, а также снизили «порог вхождения» для новых сервисов.

Но мы также сознательно избежали ловушки определения успеха как достижения 100%-ного принятия нашего продукта в организации. Во многих случаях создание функциональности, которая бы подходила для абсолютно всех сервисов компании, не стоит затраченных усилий.

Команда в развитии

Отбирая SRE-инженеров для работы над программными продуктами, мы обнаружили, что наиболее эффективны команды, ядро которых совмещает в себе универсалов, способных быстро включаться в работу над новой темой, и инженеров с широкими познаниями и опытом. Такое разнообразие знаний помогает закрыть белые пятна, а также не дает считать аксиомой то, что каждая команда будет пользоваться продуктом так же, как и ваша.

Важно, чтобы в вашей команде установились рабочие отношения с нужными специалистами, а также чтобы вашим инженерам было комфортно работать над новой областью задач. Для большинства команд SRE во многих компаниях работа в этой новой области задач требует перепоручать часть задач сторонним исполнителям или работать с консультантами, но команды SRE в более крупных организациях имеют возможность объединиться с собственными экспертами компании. На начальных этапах проектирования Auxon мы представили наш проект командам, которые специализируются в области исследования операций и количественного анализа, чтобы воспользоваться их опытом в этой области и повысить уровень компетенции команды разработчиков Auxon, необходимой для задачи планирования производительности.

По мере разработки проекта и расширения функциональности Auxon к команде присоединились специалисты, имеющие опыт работы со статистикой и математической оптимизацией, что было равносильно привлечению стороннего консультанта для малой компании. Эти новые члены команды после завершения работы над базовой функциональностью смогли наметить направления

возможных улучшений, и главным нашим приоритетом стало повышение мастерства.

Удачный момент для привлечения специалистов, конечно же, разнится от проекта к проекту. Упрощенно говоря, проект должен стать успешным с первой версии, и это поможет вашей команде повысить свои навыки с помощью сторонних специалистов.

Культивирование разработки ПО в службе SRE

Что позволяет программе перестать быть одноразовым инструментом и превратиться в полноценный программный проект? Этому могут значительно поспособствовать инженеры, имеющие опыт из первых рук в требуемых областях, заинтересованные в работе над проектом, а также целевая база пользователей, имеющих технические навыки (такие пользователи могут представлять полезные отчеты о недочетах и ошибках на ранних этапах разработки). Проект должен предоставлять заметные преимущества — например, снижение рутины для сотрудников SRE, улучшение существующей инфраструктуры или упрощение сложного процесса.

Проекту важно вписаться в набор целей организации, чтобы ведущие инженеры смогли оценить его потенциальное влияние и впоследствии рекомендовать ваш проект как опираясь на свои подчиненные команды, так и посредством команд, с которыми они взаимодействуют. Кросс-организационная социализация и обзоры помогают предотвратить ситуацию разрозненных или перекрывающихся усилий, и для продукта, который легко может способствовать выполнению целей подразделения, становится легче выделить персонал и поддерживать его.

Какой проект считается плохим кандидатом? На такой проект указывает множество признаков — он связан с изменением слишком многих частей, его дизайн требует подхода «все или ничего», который мешает выполнять разработку итерационно. Поскольку команды Google SRE в данный момент организованы вокруг поддерживаемых ими сервисов, проекты, которые они разрабатывают, могут оказаться слишком узкоспециализированными, закрывая лишь небольшую часть потребностей организации. Поскольку команды мотивированы предоставлять возможность комфортной работы в первую очередь своим пользователям со своим сервисом, проекты, как правило, не могут быть обобщены для более широкого применения, так как стандартизация между SRE-командами проводится лишь во вторую очередь. С другой стороны, слишком общие фреймворки могут также оказаться проблемными; если инструмент является слишком гибким и слишком универсальным, он может не подойти в полной мере ни для одного конкретного применения и поэтому окажется практически бесполезен. Проекты с большой областью действия и абстрактными целями зачастую требуют приложения значительных усилий для их разработки, и при этом им недостает конкретных практик использования, необходимых для того, чтобы проект принес пользу конечному пользователю за разумный промежуток времени.

Примером продукта широкого применения может служить балансировщик нагрузки третьего уровня, разработанный SRE-инженерами компании Google: он был настолько успешным на протяжении многих лет, что стал пользовательским продуктом Google Cloud Load Balancer [Eisenbud, 2016].

Успешное создание культуры разработки ПО службой SRE: набор персонала и время разработки

SR-инженеры зачастую являются универсалами, поскольку желание получить широкие знания вместо глубоких способствует пониманию общей картины (найдется немного примеров систем, которые масштабнее сложной кухни современной технической инфраструктуры). Эти инженеры зачастую имеют серьезные навыки кодирования и разработки ПО, но могут не иметь традиционного опыта разработки ПО, который заключается в том, чтобы состоять в команде разработчиков продукта и в необходимости думать о пользовательских запросах функциональности. Один инженер, который работал в ранних проектах по разработке ПО, подытожил традиционный для SRE подход к разработке следующей фразой: «У меня есть описание архитектуры; зачем нам нужны требования?» Взаимодействуя с инженерами, менеджеры, имеющие опыт разработки пользовательского ПО, могут помочь создать культуру разработки ПО, в которой соединится лучшее из опыта разработки продукта и работы с уже эксплуатируемыми системами.

Любой проект должен разрабатываться без помех и перерывов. Это очень важно, поскольку практически невозможно писать код — и тем более сосредотачиваться на крупных и весомых проектах, — когда вы переключаетесь от задачи к задаче в течение одного часа. Поэтому возможность работать над проектом не прерываясь — это хороший стимул для инженера начать над ним работать.

Большая часть программных продуктов, разработанных внутри службы SRE, начинают свой путь как побочные проекты, чья полезность приводит к их росту и оформлению в более завершенном виде. В этот момент продукт может пойти по одному из следующих путей.

- Оставаться второстепенной низкоприоритетной работой, которой будут заниматься в свободное время.
- Пройти процесс структурирования и получить формальный статус проекта (см. подраздел «Движемся к успеху» далее).
- Получить поддержку от руководства SRE и стать полноценным проектом по разработке ПО с выделенным для него персоналом.

Однако при любом сценарии — и на это стоит обратить внимание — важно, чтобы SR-инженеры, участвующие в разработке, продолжали свою деятельность как SR-инженеры, а не становились приписанными к SRE разработчиками на полную ставку. Погружение в мир эксплуатируемых систем дает SR-инженерам, в том числе и занятым в разработке, бесценный опыт и делает их точку зрения уникальной, поскольку они выступают в роли как создателя, так и потребителя продукта.

Движемся к успеху

Если вам нравится идея организовать разработку ПО в службе SRE, вы, возможно, задумались о том, как внедрить модель разработки ПО в подразделении, организованном вокруг поддержки продукта.

Во-первых, вам нужно понять, что эта цель является не только организационной, но еще и технической задачей. SR-инженеры привыкли работать вместе со своими коллегами, быстро анализируя проблемы и реагируя на них. Поэтому вы движетесь против естественного инстинкта SR-инженера, который заключается в быстром написании кода, соответствующего его сиюминутным потребностям. Если ваша

команда SRE мала, это не станет проблемой. Однако по мере роста вашей организации этот подход не будет масштабироваться, став вместо многофункционального решения узким или специализированным, а созданными программными решениями нельзя будет поделиться, что неизбежно приведет к повторному выполнению работы снова и снова.

Далее подумайте о том, чего вы хотите достичнуть, разрабатывая ПО силами SRE. Вы просто хотите поспособствовать появлению хороших приемов разработки ПО внутри своей команды или же вы заинтересованы в разработке ПО, в итоге которой вы получите результаты, которые пригодны для использования другими командами и которые, возможно, станут стандартом в организации? В более крупных сформировавшихся организациях второй путь займет какое-то время, возможно даже несколько лет. Такие преобразования приходится вести по всем фронтам, но они дают и большие преимущества. Из нашего опыта мы можем посоветовать следующее.

- *Четко сформулируйте задачу и доведите ее до сведения остальных.* Важно определить и распространить вашу стратегию, планы и — что наиболее важно — преимущества, которые служба SRE получит благодаря таким изменениям. SR-инженеры довольно скептичны (фактически скептицизм — это черта, за которую мы их нанимаем); первой их реакцией будет ответ вроде «выглядит как лишняя нагрузка» или «это никогда не сработает». Начните с того, что расскажите, как эта стратегия поможет работе SRE, например:

- устойчивые и поддерживаемые программные решения ускорят обучение новых SR-инженеров;
- снижение количества способов выполнения одной и той же задачи позволит всем получить пользу от навыков, выработанных в одной команде, что даст возможность передавать знания между командами и объединять усилия.

Когда SR-инженер начнет задавать вопросы о том, почему ваша стратегия сработает, вместо вопросов о том, стоит ли ему заниматься вообще, можете считать, что первый этап пройден успешно.

- *Оцените возможности вашей организации.* SR-инженеры имеют множество навыков, но, как правило, им недостает навыка работы в команде, которая уже создала и поставила продукт множеству пользователей. Для того чтобы разрабатывать полезное ПО, вы, по сути, создаете команду продукта. Эта команда включает в себя требуемые роли и навыки, которые могли не требоваться ранее в вашей службе SRE. Будет ли кто-то играть роль менеджера продукта, выступая в защиту интересов потребителя? Имеет ли ваш ведущий инженер или менеджер проектов навыки или опыт для ведения процесса разработки по методологии Agile?

Начните заполнять эти пробелы, опираясь на те навыки, которые уже имеются в вашей компании. Попросите вашу команду разработчиков продуктов помочь наладить процесс Agile путем обучения или тренировок. Запросите консультацию у менеджеров продуктов для того, чтобы определить требования к продукту и расставить приоритеты в работе над функциональностью. При достаточно большом объеме программных разработок вполне допустимо нанять

отдельных специалистов для выполнения этих функций. Нанять людей на эти должности станет проще, как только вы добьетесь первых положительных результатов.

- *Запускайте и дорабатывайте.* Как только вы запустите программу по разработке ПО службой SRE, за вашими усилиями будет следить множество глаз. Очень важно завоевать доверие, предоставив хоть сколько-нибудь полезный продукт за приемлемый промежуток времени. Перед продуктом первого выпуска должны стоять относительно простые и достижимые цели — цели, которые не содержат противоречивых или уже существующих решений. Мы также обнаружили, что удобно объединять такой подход с шестимесячным циклом выпуска обновлений продукта, в которых становится доступной дополнительная полезная функциональность. Такой цикл позволяет командам сосредоточиться на определении подходящего набора функций для очередной итерации, а затем на реализации этих функций, одновременно обучаясь быть «продуктовой» командой разработчиков. После запуска результатов первой итерации некоторые команды переходят на модель push-on-green для еще более быстрой выдачи обновлений и получения обратной связи.
- *Не занижайте ваши стандарты.* Как только вы начнете разрабатывать ПО, вам может захотеться «срезать углы». Вы должны сопротивляться этому искушению, придерживаясь стандартов, которые установлены для ваших команд разработчиков. Например:
 - спросите себя: если бы этот продукт был создан отдельной командой разработчиков, воспользовались ли бы вы им;

- если ваше решение принято широким кругом пользователей, оно может стать критически важным для SR-инженеров, поскольку позволит более успешно выполнять их работу. Поэтому надежность для вас крайне важна. Пользуетесь ли вы правилами экспертизы кода? Проводится ли у вас непрерывное или интеграционное тестирование? Попросите другую команду SR-инженеров выполнить инспекцию готовности продукта, как если бы они это делали, принимая на сопровождение любой другой сервис.

Для того чтобы ваш продукт завоевал доверие, потребуется много времени, причем это доверие может быть утрачено из-за одного-единственного промаха.

Итоги главы

Проекты по разработке ПО службой SRE процветают по мере роста организации, и во многих случаях полученный опыт успешного выполнения подобных проектов проложил дорогу для последующих начинаний. Уникальный практический опыт работы со средой промышленной эксплуатации, который SR-инженеры применяют при разработке проектов, может способствовать появлению инновационных подходов к решению старых проблем, как это видно на примере разработки системы Auxon, предназначеннной для решения сложной задачи планирования производительности. Программные проекты, выполняемые специалистами SRE, также приносят пользу компании, так как такой подход помогает выстроить устойчивую модель поддержки сервисов разных масштабов. Поскольку SR-инженеры часто разрабатывают ПО для упрощения неэффективных процессов или для автоматизации выполнения часто встречающихся

задач, такие проекты означают, что команда SRE не будет разрастаться пропорционально росту поддерживаемых ими сервисов. Таким образом, наличие SR-инженеров, которые посвящают часть своего времени разработке ПО, приносит пользу компании, службе SRE и самим инженерам.

[129](#) Здесь под «производительностью» (англ. capacity) понимается обобщенное количество ресурсов, которыми располагает вычислительная система, и, соответственно, ее «емкость» в отношении выполняемой работы. — *Примеч. пер.*

19. Балансировка нагрузки на уровне фронтенда

Автор — Петр Левандовски

Под редакцией Сары Чевис

Каждую секунду мы обслуживаем миллионы запросов, и, как вы уже могли догадаться, для этих целей используем больше одного компьютера. Но даже если бы у нас был суперкомпьютер, который каким-то образом мог бы обрабатывать все эти запросы (представьте, какая связность сети потребовалась бы для этого!), мы все равно не могли бы полагаться на единственную «точку отказа». Когда вы работаете с крупномасштабными системами, класть все яйца в одну корзину — это прямой путь к катастрофе.

В этой главе рассматривается вопрос высокоуровневой балансировки нагрузки — как мы балансируем пользовательский трафик между дата-центрами. В следующей главе рассматривается вопрос балансировки нагрузки внутри дата-центра.

Мощность — это не ответ

Чисто теоретически предположим, что мы располагаем невероятно мощной машиной и сетью, которая никогда не дает сбоев. Будет ли достаточно такой конфигурации для удовлетворения потребностей компании Google? Нет. Даже при такой конфигурации нас сдерживали бы физические ограничения, связанные с нашей сетевой инфраструктурой. Например, скорость света — это ограничивающий фактор для скорости обмена данными посредством оптоволоконного

кабеля, устанавливающего верхнюю границу того, как быстро мы можем выдавать данные в зависимости от расстояния, которое необходимо преодолеть. Даже в идеальном мире полагаться на инфраструктуру, имеющую единственную «точку отказа», — плохая идея.

В реальности компания Google имеет тысячи машин и еще больше пользователей, многие из которых отправляют по несколько запросов одновременно. *Балансировка нагрузки*, создаваемой обращениями к сервисам (пользовательским трафиком) состоит в том, чтобы определить, какая из огромного множества машин в наших дата-центрах будет обслуживать каждый конкретный запрос. В идеале трафик будет распределяться среди множества сетевых узлов, дата-центров и машин оптимальным образом. Но что в данном контексте означает слово «оптимальным»? Единого ответа на этот вопрос не существует, поскольку оптимальность решения сильно зависит от следующих факторов.

- Уровень иерархии, на котором мы решаем задачу (глобальный или локальный).
- Технический уровень, на котором мы решаем задачу (аппаратное обеспечение или программное).
- Природа обрабатываемого трафика.

Начнем с рассмотрения двух распространенных сценариев работы с трафиком: простой поисковой запрос и запрос на загрузку видеоролика. Пользователи хотят получить результат выполнения своего запроса быстро, поэтому наиболее важная характеристика поискового запроса — задержка отклика. С другой стороны, пользователи ожидают, что загрузка видеороликов займет некоторое время, но также хотят, чтобы

такой запрос выполнялся с первого раза, поэтому наиболее важной характеристикой такого запроса будет пропускная способность. Различающиеся потребности для этих двух запросов имеют значение для выбора, как мы определяем оптимальное распределение для каждого запроса на глобальном уровне.

- Поисковой запрос отправляется в ближайший — согласно измеренному значению времени обращения запроса или «круговой задержки» к сети (round-trip time, RTT) — доступный дата-центр, поскольку мы хотим минимизировать задержку отклика для этого запроса.
- Поток загрузки видеоролика направляется по другому пути — скорее всего, на узел, который в данный момент мало используется — для максимизации пропускной способности ценой повышения задержек.

При этом на локальном уровне, внутри выбранного дата-центра, мы зачастую подразумеваем, что все машины в здании равноудалены от пользователя и одинаково подключены к одной и той же сети. Таким образом, оптимальное распределение нагрузки направлено на оптимальное использование ресурсов и защиту серверов от перегрузки.

Конечно же, в этом примере показана упрощенная картина. В реальности на оптимальность распределения нагрузки влияет большее количество факторов: некоторые запросы могут быть направлены в дата-центр, находящийся чуть дальше, для того, чтобы кэши оставались «теплыми», или, например, для предотвращения перегрузки сети неинтерактивный трафик может быть направлен в совершенно другой регион. В Google эта задача решается путем балансировки нагрузки на нескольких уровнях, два из которых

описаны в следующих разделах. Для большей определенности мы рассмотрим запросы HTTP, передаваемые через TCP. Балансировка нагрузки для сетевых сервисов, работающих без сохранения состояния (вроде DNS поверх UDP)[130](#), несколько отличается, но большая часть механизмов, описанных здесь, может быть применима и к ним.

Балансировка нагрузки с использованием DNS

Прежде чем клиент сможет отправить запрос HTTP, ему обычно нужно определить IP-адрес с помощью DNS. Это дает нам идеальную возможность продемонстрировать наш первый уровень балансировки нагрузки — балансировку нагрузки на уровне DNS. Самым простым решением будет возвращать в ответах DNS сразу несколько записей типа A или AAAA, содержащих различные IP-адреса, и позволить клиенту выбрать один из предложенных адресов произвольным образом. Несмотря на то что реализация такого подхода кажется тривиальной, она порождает несколько проблем.

Первая проблема состоит в том, что мы почти не контролируем поведение клиента: записи выбираются случайным образом, и за каждой из них стоит примерно одинаковый объем трафика. Как мы можем справиться с этой проблемой? Теоретически мы могли бы указать веса и приоритеты возвращенных записей с помощью записей типа SRV, но они пока не применимы для протокола HTTP.

Еще одна потенциальная проблема связана с тем, что клиент, как правило, не может определить ближайший к нему адрес. Проблема частично решается использованием для обращения к ответственным (authoritative) серверам имен «широковещательного» адреса в расчете на то, что запросы DNS будут попадать на ближайший адрес. В своем ответе

сервер может возвращать маршрут к ближайшему дата-центру в виде цепочки адресов. Следующим усовершенствованием будет создание карты всех сетей и их приблизительных физических местоположений, а на основе этой карты будут обрабатываться DNS-запросы. Однако для этого потребуется гораздо более сложная реализация DNS-сервера, а также служебный процесс, который бы поддерживал актуальность карты.

Конечно, ни одно из этих решений не оказывается тривиальным из-за фундаментальной особенности работы DNS: конечные пользователи редко общаются непосредственно с ответственными серверами имен. Вместо этого где-то между ними обычно располагается рекурсивный DNS-сервер. Он проксирует (и зачастую кэширует) запросы между пользователем и сервером. Можно выделить три важнейших аспекта влияния промежуточного (middleman) DNS-сервера на управление трафиком:

- рекурсивное разрешение имен в IP-адреса;
- возврат ответов по нефиксированным путям;
- дополнительное усложнение вследствие кэширования.

Проблемы при рекурсивном разрешении IP-адресов связаны с тем, что IP-адрес, видимый ответственному серверу имен, принадлежит не конечному пользователю, а промежуточному рекурсивному серверу, участвующему в разрешении имени. Это очень серьезное ограничение, поскольку из-за него оптимизация ответа возможна лишь для ближайшего отрезка между ответственным и промежуточным серверами. В качестве возможного решения мы можем использовать расширение EDNS0, предложенное в [Contavalli,

2015], которое добавляет информацию о подсети клиента в запрос DNS, отправляемый рекурсивным сервером. Таким образом, ответственный сервер имен возвращает ответ, оптимальный с точки зрения конечного пользователя, а не промежуточных серверов-преобразователей. Хотя такой подход все еще не является официальным стандартом, его очевидные преимущества привели к тому, что крупнейшие DNS-серверы (например, OpenDNS и Google¹³¹) уже начали его поддерживать.

Трудность заключается не только в поиске оптимального IP-адреса, который будет возвращен на сервер имен для заданного запроса пользователя, но и в том, что в зоне ответственности сервера имен могут находиться тысячи или миллионы пользователей на территориях от одиночного офиса до целого континента. Например, крупный национальный интернет-провайдер может запустить серверы имен для всех своих сетей в одном дата-центре, уже имея соединения с сетями всех агломераций. DNS-серверы этого провайдера вернут IP-адрес, наилучшим образом подходящий для их дата-центра, игнорируя иные сетевые пути, даже если они лучше подходят для всех пользователей!

Наконец, рекурсивные серверы имен обычно кэшируют ответы и повторно возвращают их на аналогичные запросы в пределах времени, указанного в поле TTL (time-to-live, «время жизни») в записи DNS. В результате становится трудно предсказать масштаб влияния каждого ответа ответственного сервера: любой из них может быть направлен как одному пользователю, так и тысячам. Мы решаем эту проблему двумя способами.

- Мы анализируем изменения трафика и постоянно обновляем список известных промежуточных DNS-серверов, указывая

примерный размер пользовательской базы каждого из них, что позволяет нам отслеживать их потенциальный вклад в общую нагрузку.

- Мы оцениваем географическое расположение пользователей, находящихся за каждым отслеживаемым промежуточным сервером, чтобы с большей вероятностью направлять их в data-центр с наилучшим местоположением.

Оценка географического расположения может быть особенно сложной, если пользовательская база распределена между крупными регионами. В таких случаях при определении наилучшего местоположения мы идем на компромиссы и стараемся сделать результат оптимальным для большинства пользователей.

Но что на самом деле означает фраза «наилучшее местоположение» в контексте балансировки нагрузки с использованием DNS? Наиболее очевидный ответ — это наиболее близкое к пользователю местоположение. Однако (как если бы определение местоположения пользователя не было трудной задачей само по себе) существует дополнительное требование. Балансировщик нагрузки DNS должен убедиться, что выбранный им data-центр имеет достаточную производительность, чтобы обслужить запросы пользователя. Он также должен знать, что выбранный data-центр и его сетевое подключение находятся в хорошем состоянии, поскольку не стоит направлять запросы туда, где есть проблемы. К счастью, мы можем обеспечить взаимодействие ответственного сервера DNS с нашими глобальными системами мониторинга трафика, производительности и состояния нашей инфраструктуры.

Третий аспект влияния промежуточного DNS-сервера связан с кэшированием. Учитывая, что ответственные серверы имен не могут очищать кэши промежуточных серверов, записи DNS должны иметь относительно небольшое значение TTL. Это, по сути, устанавливает нижнюю границу скорости распространения пользователям изменений в DNS (увы, не все DNS-серверы соответствуют значению TTL, установленному ответственными серверами). К несчастью, мы можем лишь иметь в виду эту проблему при принятии решений по балансировке нагрузки.

Несмотря на все эти проблемы, DNS все еще является самым простым и самым эффективным способом сбалансируировать нагрузку еще до того, как будет установлено соединение с пользователем. С другой стороны, должно быть очевидно, что использования лишь DNS будет недостаточно. Имейте также в виду, что все ответы DNS должны соответствовать ограничению в 512 байт^{[132](#)}, установленному RFC 1035 [Moskowitz, 1987]. Тем самым лимитируется количество адресов, которое можно вместить в один ответ DNS, и это число, скорее всего, будет значительно меньше, чем количество наших серверов.

Чтобы обеспечить полноценную балансировку нагрузки на уровне фронтенда, начальный уровень балансировки нагрузки с использованием DNS должен быть дополнен уровнем, использующим виртуальный IP-адрес.

Балансировка нагрузки с использованием виртуального IP-адреса

Виртуальные IP-адреса (virtual IP address, VIP) не присваиваются определенным сетевым интерфейсам. Вместо этого их обычно делят несколько устройств. Однако с точки

зрения пользователя VIP остаются обычными IP-адресами. В теории такой прием позволяет нам скрыть детали реализации (например, количество машин, находящихся за каждым VIP) и облегчает обслуживание сети, поскольку мы можем запланировать обновления или добавить новые машины в пул незаметно для пользователя.

На практике наиболее важной частью реализации VIP является устройство, называемое сетевым балансировщиком нагрузки. Балансировщик получает пакеты и перенаправляет их одной из машин, которые лежат за VIP. Эти бэкенды выполняют последующую обработку запроса.

Существует несколько подходов, которыми может воспользоваться балансировщик при определении того, какой бэкенд должен получить запрос. Первый (и, возможно, наиболее интуитивный) подход заключается в том, что балансировщик всегда должен отдавать предпочтение наименее загруженному бэкенду. Теоретически это должно приводить к наиболее быстрому выполнению запроса пользователя, поскольку он направляется на наименее загруженную машину. К сожалению, эта логика не работает для протоколов с хранением состояния, поскольку они должны использовать один и тот же бэкенд на протяжении всего процесса выполнения запроса. Это требование означает, что балансировщик должен следить за всеми соединениями, чтобы убедиться в том, что все остальные пакеты будут отправлены на правильный бэкенд. В качестве альтернативы мы можем использовать некоторые части пакета для создания идентификатора соединения (возможно, используя хеш-функцию и какие-то данные из пакета) и задействовать этот идентификатор для выбора бэкенда. Например, идентификатор может быть выражен следующим образом:

```
id(packet) mod N,
```

где `id` — это функция, которая принимает параметр `packet` и создает идентификатор соединения, а `N` — это количество участвующих в балансировке бэкендов.

Это позволяет избежать необходимости сохранять состояние, и все пакеты, относящиеся к одному соединению, всегда будут направляться на один и тот же бэкенд. Мы решили проблему? Не совсем. Что случится, если один бэкенд откажет и его нужно будет убрать из списка бэкендов? Внезапно `N` превращается в $N - 1$, и функция принимает вид `id(packet) mod N - 1`. Практически каждый пакет теперь указывает на другой бэкенд! Если бэкенды не делятся друг с другом информацией о состоянии, такое изменение приведет к сбросу всех существующих соединений. Этот сценарий нельзя считать хорошим для пользователя, и неважно, что это будет происходить редко.

К счастью, существует альтернативное решение, которое не требует хранения в памяти состояния каждого соединения и не приводит к сбросу всех соединений при сбое одного бэкенда: консистентное хеширование (consistent hashing). Этот подход появился в 1997 году, он [Karger, 1997] описывает способ соотнесения, который остается относительно стабильным даже при добавлении или удалении бэкендов из списка. Этот подход минимизирует ущерб для существующих соединений при изменении пула бэкендов. В результате мы можем использовать простые средства контроля соединений в обычной ситуации и откатываться к консистентному хешированию, если система подвергается давлению (например, во время DoS-атаки).

Вернемся к основному вопросу: как именно сетевой балансирующий нагрузки должен направлять пакеты к выбранному VIP-бэкенду? Одно из решений заключается в использовании механизма трансляции сетевых адресов

(Network Address Translation, NAT). Однако это потребует хранения информации о каждом соединении в специальной таблице, что не позволит сделать механизм восстановления после сбоя действительно свободным от хранения состояния.

Еще одним решением является модификация информации на канальном уровне (второй уровень сетевой модели OSI). Изменив MAC-адрес получателя перенаправляемого пакета, балансировщик может оставить всю информацию на верхних уровнях без изменений, и бэкенд получит оригинальные IP-адреса источника и места назначения. Далее бэкенд может отправить ответ непосредственно отправителю — этот прием известен как прямой ответ сервера (Direct Server Response, DSR). Если запросы малы, а ответы велики (это справедливо для многих запросов HTTP), DSR позволяет сэкономить много времени, поскольку через балансировщик нагрузки должна пройти лишь малая доля трафика. И даже лучше, DSR не требует от нас хранить состояние на устройстве балансировщика нагрузки. К сожалению, использование второго уровня OSI для внутренней балансировки нагрузки имеет серьезные недостатки при развертывании в больших масштабах: все машины (например, все балансировщики нагрузки и все их бэкенды) должны иметь возможность связаться друг с другом на канальном уровне. Это не является проблемой, если сеть поддерживает такие соединения, и если число машин не увеличивается слишком быстро, поскольку все они должны оставаться в одном широковещательном домене. Как вы понимаете, компания Google переросла это решение довольно давно, и нам пришлось искать альтернативный подход.

Наше текущее решение по балансировке нагрузки для VIP [Eisenbud, 2016] использует инкапсуляцию пакетов. Сетевой балансировщик нагрузки помещает (инкапсулирует)

перенаправляемый пакет в другой пакет IP в соответствии с протоколом туннелирования сетевых пакетов (Generic Routing Encapsulation, GRE) [Hanks, 1994] и использует в качестве адреса получателя адрес бэкенда. Бэкенд, получающий пакет, снимает внешний слой IP+GRE и обрабатывает внутренний пакет IP, как если бы тот был доставлен непосредственно на его сетевой интерфейс. Сетевому балансировщику нагрузки и бэкенду больше нет необходимости находиться внутри одного широковещательного домена; до тех пор пока между ними имеется проложенный через «туннель» маршрут, они даже могут находиться на разных континентах.

Инкапсуляция пакетов — это мощный механизм, который предоставляет большую гибкость для наших сетей, учитывая их проект и способ развития. К сожалению, за инкапсуляцию приходится платить увеличением размера пакета. Из-за такого «довеска» (если быть точным, 24 байта в случае IPv4+GRE) может быть превышен максимальный доступный размер пакета (Maximum Transmission Unit, MTU), и ему потребуется фрагментация.

Когда пакет достигнет дата-центра, фрагментации можно будет избежать, используя внутри дата-центра большее значение MTU; однако этот подход требует, чтобы сеть поддерживала протоколы с большим размером сообщений. Как и во многих других случаях работы с крупномасштабными системами, задача балансировки нагрузки на первый взгляд проста, но ее сложность кроется в деталях, как при балансировке нагрузки на уровне фронтенда, так и при обработке пакетов внутри дата-центра.

130 Сам по себе HTTP считается протоколом без сохранения состояния, но используемый им TCP — протокол с сохранением состояния. Служба DNS и используемый ею протокол UDP — без сохранения состояния. — Примеч. пер.

131 См.: <https://groups.google.com/forum/#!topic/public-dns-announce/67oxFjSLeUM>.

[132](#) В противном случае пользователи должны будут устанавливать TCP-соединение только для того, чтобы получить список IP-адресов.

20. Балансировка нагрузки в дата-центре

Автор — Александро Фореро Куэрво

Под редакцией Сары Чевис

В этой главе рассматривается вопрос балансировки нагрузки внутри дата-центра. В частности, речь пойдет об алгоритмах для распределения работы внутри заданного дата-центра для потока запросов. Мы рассмотрим политики уровня приложений для направления запросов к конкретным серверам, которые могут их обработать. Принципы работы с сетью на нижнем уровне (например, использование коммутаторов, маршрутизация пакетов) и выбор дата-центров в этой главе не рассматриваются.

Предположим, что существует поток запросов, поступающих в дата-центр — они могут исходить из самого дата-центра, из удаленных дата-центров, или и то и другое одновременно — в таком объеме, который не превышает возможностей дата-центра по их обработке (или превышает, но лишь на короткий промежуток времени). Предположим также, что в дата-центре работают *сервисы*, к которым поступают эти запросы. Эти сервисы реализованы в виде однотипных (гомогенных) взаимозаменяемых серверных процессов, работающих, как правило, на разных машинах. Самые маленькие сервисы обычно имеют как минимум три таких процесса (использование меньшего количества процессов означает потерю как минимум 50 % вашей производительности при потере одной машины), а самые крупные — более 10 000 (в зависимости от размера дата-центра).

Типичное количество процессов сервиса — от ста до тысячи. Мы называем эти процессы *задачами бэкенда* (или просто

бэкендами). Другие задачи, называемые *задачами клиентов*, хранят соединения с задачами бэкенда. Для каждого входящего запроса задача клиента должна решить, какая из задач бэкенда должна обрабатывать этот запрос. Задачи клиентов и бэкенда общаются между собой с помощью протокола, реализованного на основе комбинации TCP и UDP.

Следует заметить, что в данных центрах компании Google представлен обширный набор разнообразных сервисов, которые реализуют различные комбинации политик, рассматриваемых в этой главе. Рассматриваемый нами пример не соответствует в точности ни одному конкретному сервису. Он представляет собой обобщенный случай, позволяющий нам обсудить разнообразные технологии и методы, которые мы находим полезными для разных сервисов. Некоторые из них могут быть более или менее применимы для конкретных случаев, но все они разрабатывались и реализовывались множеством инженеров Google на протяжении многих лет.

Эти технологии и методы применимы для многих элементов нашего стека. Например, большая часть внешних HTTP-запросов достигают GFE (Google Frontend), нашей системы обратного проксирования HTTP. GFE использует описываемые алгоритмы (наряду с описанными в главе 19) для направления запросов и их метаданных конкретным процессам приложений, которые могут их обработать. Используемая при этом конфигурация соотносит различные шаблоны URL с отдельными приложениями, находящимися в ведении разных команд. Чтобы сгенерировать содержимое ответов (которые будут возвращены GFE и затем в браузеры), эти приложения зачастую используют эти же алгоритмы для взаимодействия с инфраструктурными или дополнительными сервисами, от которых они зависят. Иногда стек зависимостей оказывается достаточно глубоким: один входящий HTTP-

запрос может запустить длинную цепочку промежуточных зависимых запросов к различным системам, потенциально разветвляющуюся в нескольких местах.

Идеальный случай

В идеальном случае нагрузка для заданного сервиса распределяется равномерно по всем задачам бэкенда, и в любой момент времени наименее и наиболее загруженные задачи бэкенда потребляют ресурсы процессора примерно в одинаковом объеме.

Мы можем отправлять трафик в дата-центр только до тех пор, пока наиболее загруженная задача не достигнет предела нагрузочной способности; это показано на рис. 20.1 для двух ситуаций с одинаковым временным интервалом. В это время алгоритм балансировки нагрузки между дата-центрами должен избегать отправки дополнительного трафика в этот дата-центр, поскольку возможна перегрузка некоторых задач.

Как показано на левой гистограмме на рис. 20.2, значительная часть вычислительных ресурсов тратится впустую: у каждой задачи, кроме наиболее загруженной, остается запас производительности.

Распределение нагрузки между задачами

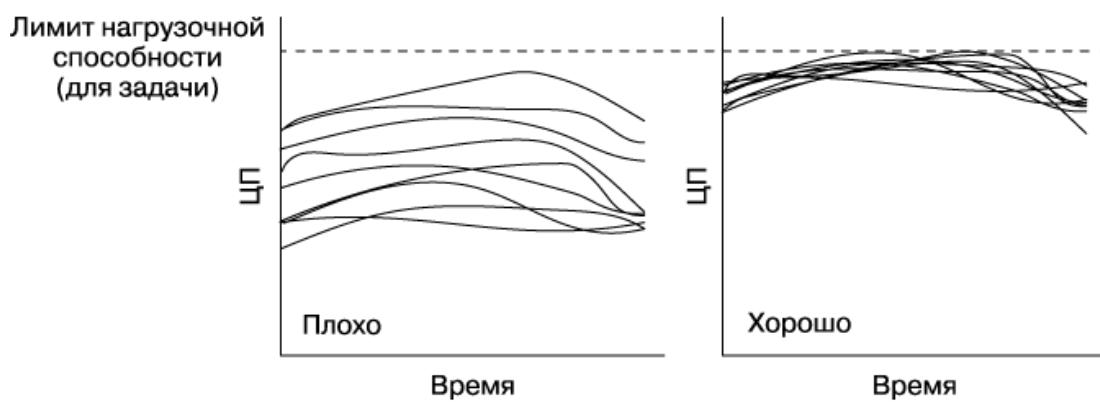


Рис. 20.1. Два сценария распределения нагрузки для задач с течением времени

Использование ЦП задачами в заданный момент времени

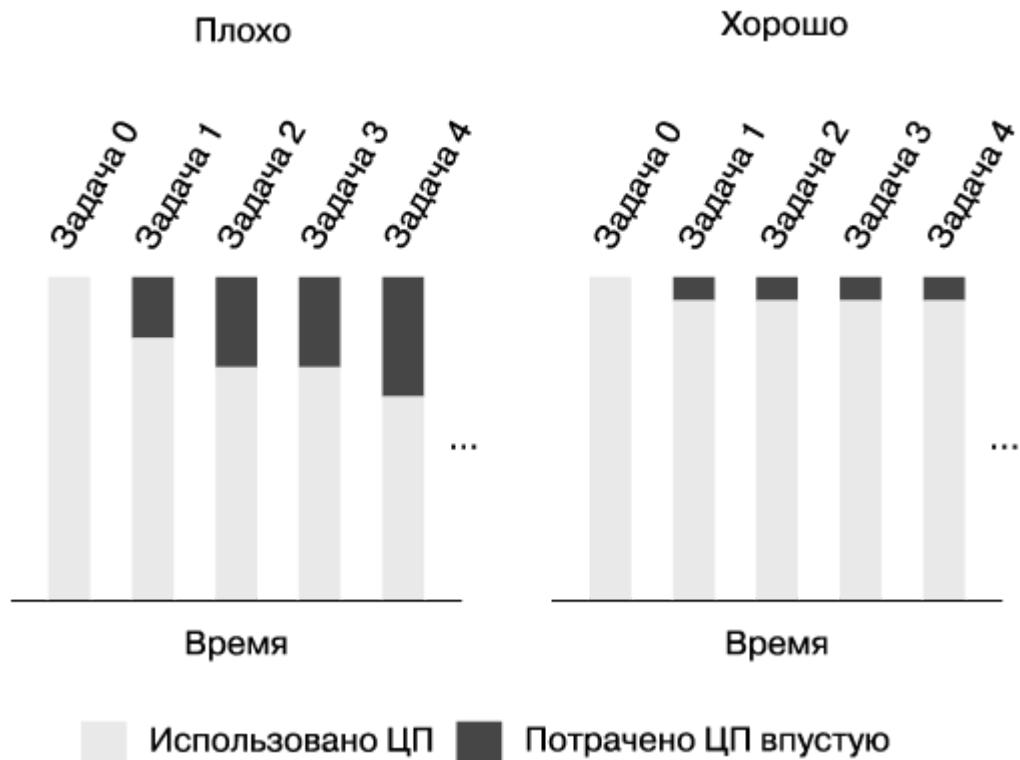


Рис. 20.2. Гистограммы использования процессоров в двух сценариях

Говоря более формально, пусть $CPU(i)$ — уровень использования ЦП задачей i в любой заданный момент времени. Предположим, что самой загруженной задачей будет задача 0. Тогда в случае большой дисперсии мы теряем сумму разностей между уровнем использования ЦП самой загруженной задачи и каждой из оставшихся задач: это значит, что будет потеряна сумма значений $(CPU(0) - CPU(i))$ для всех i -х задач. В данном случае «потеряна» означает «зарезервирована, но не использована».

Этот пример иллюстрирует, как неудачные внутренние методы балансировки нагрузки искусственным образом ограничивают доступность ресурсов: вы можете зарезервировать 1000 процессоров для своего сервиса в

определенном дата-центре, но реально сможете использовать, например, не более 700 из них.

Выявляем плохие задачи: управление потоками и «хромые утки»

Прежде чем решить, какая задача бэкенда должна принять запрос клиента, нам нужно выявить неработоспособные задачи в нашем пуле бэкендов и найти способ избежать их появления [133](#).

Простой подход к контролю работоспособности задач: управление потоками

Предположим, что наши задачи клиентов отслеживают количество активных («открытых») запросов, которые они отправили через соответствующее соединение каждой задаче бэкенда. Когда это число активных запросов достигнет указанного лимита, бэкенд будет считаться неработоспособным, и новые запросы ему перестанут отправляться. Для большинства бэкендов разумно ставить ограничение в 100 запросов; как правило, запросы завершаются достаточно быстро, и вероятность достижения заданного лимита в обычных условиях достаточно низка. Эта (очень простая!) форма управления потоками действует так же, как простейшая балансировка нагрузки: если заданная задача бэкенда оказывается перегруженной и начинают накапливаться необслуженные запросы, задачи клиентов станут избегать использования этого бэкенда и нагрузка будет естественным образом распределена между другими задачами бэкенда.

К сожалению, такой подход защищает задачи бэкенда лишь от самых крайних случаев перегрузки, и бэкенды могут оказаться перегруженными, даже не достигнув этого лимита. Противоположное тоже верно: в некоторых случаях клиенты могут достичь этого лимита, а у их бэкендов все еще будет в распоряжении достаточно ресурсов. Например, отдельные бэкенды могут иметь запросы с очень большой длительностью выполнения, быстро ответить на которые невозможно. Бывают случаи, когда заданный по умолчанию лимит отрабатывал неправильно, делая все задачи бэкенда недостижимыми, при том, что все запросы были заблокированы в клиенте до завершения по тайм-ауту или из-за ошибки. Повышение лимита активных запросов поможет избежать возникновения такой ситуации, но не решит основной проблемы — мы не можем определить, стала ли задача неработоспособной или она просто медленно отвечает.

Усовершенствованный подход к контролю работоспособности задач: состояние «хромой утки»

С точки зрения клиента заданная задача бэкенда может находиться в одном из следующих состояний.

- *Полная работоспособность.* Задача бэкенда была корректно проинициализирована и теперь обрабатывает запросы.
- *Отказ в соединении.* Задача бэкенда не отвечает. Это может случиться, если задача еще только запускается или уже завершается либо если бэкенд находится в ненормальном состоянии (однако иногда бывает так, что бэкенд перестает слушать свой порт, но сам не отключается).

- «Хромая утка» (*частичная работоспособность*). Задача бэкенда слушает порт и может обслуживать запросы, но явно просит клиентов перестать их отправлять.

Когда задача входит в состояние «хромой утки», она информирует об этом всех активных клиентов. Но как быть с неактивными клиентами? Согласно реализации RPC компании Google, неактивные клиенты (то есть клиенты, не имеющие активных соединений TCP) продолжают периодически слать запросы проверки состояния посредством протокола UDP. В результате о состоянии задачи будут быстро проинформированы все клиенты — как правило, за один или два интервала RTT — независимо от их текущего состояния.

Основное преимущество того, что задача может существовать в квазиоперационном состоянии «хромой утки», состоит в том, что это упрощает ее корректное отключение. Это позволяет избежать возврата ошибок вместо ожидаемого ответа всем неудачным запросам, которым не посчастливилось быть активными в момент начала отключения задачи бэкенда. Отключение задачи бэкенда, имеющей активные запросы, без генерации ошибок облегчает выполнение обновлений кода, поддержание доступности и восстановление после сбоев, которые могут потребовать перезапуска всех связанных задач. Такое отключение должно пройти следующие этапы.

1. Планировщик отправляет сигнал SIGTERM задаче бэкенда.
2. Задача бэкенда входит в состояние «хромой утки» и сообщает своим клиентам о необходимости отправлять новые запросы другим задачам бэкенда. Это выполняется с помощью вызова API в реализации RPC, которая явно вызывается в обработчике сигнала SIGTERM.

3. Любые запросы, чье выполнение началось до перехода в состояние «хромой утки» (или после этого перехода, но до того, как клиент это обнаружил), выполняются как обычно.
4. По мере того как ответы возвращаются клиентам, количество активных запросов бэкенда постепенно уменьшается до нуля.
5. По прошествии заданного времени задача бэкенда либо сама завершает работу, либо ее планировщик завершает ее принудительно. Интервал должен быть достаточно велик, чтобы все типичные запросы имели достаточно времени для своего завершения. Это значение зависит от сервиса, но мы рекомендуем устанавливать его в промежутке от 10 до 150 секунд в зависимости от сложности.

Эта стратегия также позволяет клиенту устанавливать соединения с задачами бэкендов во время выполнения потенциально длительных инициализирующих процедур, когда они еще не готовы начать обслуживать запросы. В противном случае задачи бэкенда могут начать принимать соединения, только когда они уже готовы начать обслуживание, а это неоправданно задержало бы установление соединений. Как только задача бэкенда будет готова начать обслуживание, она явно просигнализирует об этом своим клиентам.

Ограничение пула соединений с помощью подмножеств

В дополнение к управлению состоянием, еще одним подходом к балансировке нагрузки является использование

подмножеств: ограничение пула потенциальных задач бэкенда, с которыми может взаимодействовать задача клиента.

Каждый клиент в нашей системе RPC поддерживает пул долгоживущих соединений с бэкендами, которые он использует для отправки новых запросов. Эти соединения обычно устанавливаются, как только запускается клиент, и чаще всего остаются открытыми. Запросы проходят через них вплоть до «смерти» клиента. Альтернативной моделью будет установление и уничтожение соединения для каждого запроса, но эта модель имеет высокую стоимость в плане расхода ресурсов и задержки обработки данных. Если соединение простоявает слишком долгое время, наша реализация RPC включает оптимизацию: переводит соединение в более экономный «неактивный» режим, в котором, например, частота проверок работоспособности снижена, и базовое TCP-соединение сбрасывается, заменяясь на UDP.

Каждое соединение требует расхода некоторого объема памяти и времени процессора (из-за периодических проверок работоспособности) на обоих концах. Несмотря на то что такая нагрузка в теории невелика, она быстро становится заметной, когда это случается с большим количеством машин. Подмножества позволяют избежать ситуации, в которой один клиент соединяется со слишком многими задачами бэкенда или одна задача бэкенда получает соединения от слишком многих задач клиентов. В обоих случаях вы потенциально потратите слишком много ресурсов без существенной пользы.

Выбираем правильное подмножество

Выбор правильного подмножества сводится к выбору количества задач бэкенда, с которыми будет соединяться каждый клиент, — размеру подмножества — и алгоритму выбора. Мы обычно используем подмножества размером от 20

до 100 задач бэкенда, но «правильный» размер подмножества для системы сильно зависит от типичного поведения вашего сервиса. Например, более крупное подмножество вам может понадобиться, если:

- количество клиентов значительно меньше, чем число бэкендов. В этом случае вы хотите, чтобы количество бэкендов, приходящихся на клиента, было достаточно большим и у вас не осталось задач бэкенда, которые никогда не получают запросов;
- нагрузка внутри заданий клиента часто бывает неравномерной (например, одна задача отправляет больше запросов, чем другая). Это типично в случаях, когда клиент время от времени отправляет целые пакеты запросов. При этом сами клиенты принимают от других клиентов запросы, которые время от времени бывают «веерными» (к примеру, «считать всю информацию обо всех пользователях, связанных с заданным»). Поскольку весь пакет запросов будет направлен в одно подмножество, назначенное клиенту, вам понадобится подмножество большего размера, чтобы обеспечить равномерное распределение нагрузки между большим набором доступных задач бэкенда.

Как только вы определитесь с размером подмножества, понадобится алгоритм для определения подмножества задач бэкенда, который будет использовать каждая задача клиента. Это может показаться простой задачей, но она очень быстро становится сложной, когда вы начнете работать с крупномасштабными системами, для которых важна эффективность выделения ресурсов, а перезапуски будут обычным делом.

Алгоритм выбора для клиентов должен единообразно назначать бэкенды для оптимизации распределения ресурсов. Например, если при выделении подмножеств один бэкенд перегружается на 10 %, все бэкенды должны получить на 10 % больше ресурсов, чем им необходимо. Алгоритм также должен корректно обрабатывать перезапуски и сбои, продолжая нагружать бэкенды настолько единообразно, насколько это возможно, и при этом минимизируя отток клиентов. В этом случае «отток» связан с выбором бэкенда на замену. Например, когда задача бэкенда становится недоступной, ее клиентам потребуется выбрать бэкенд для временной замены. Когда такой бэкенд выбран, клиенты должны создать новые соединения TCP (и, скорее всего, повторно провести согласование на уровне приложений), что создает дополнительную нагрузку. Аналогично, когда задача клиента перезапускается, она должна повторно открыть соединения со всеми своими бэкендами.

Кроме того, алгоритм должен обрабатывать изменение количества клиентов и/или бэкендов с минимальной потерей соединений и не зная этого количества наперед. Эта возможность особенно важна (и сложна в реализации), когда множество задач клиентов или бэкендов перезапускается целиком одновременно (например, для обновления версии). Мы хотим, чтобы после запуска обновления бэкендов клиенты продолжали работать прозрачно и с минимально возможной потерей соединений.

Алгоритм выбора подмножества: случайное подмножество

Примитивная реализация алгоритма выбора подмножества может состоять в случайной перетасовке клиентом списка бэкендов и заполнении своего подмножества доступными и работоспособными бэкендами из этого списка. Однократная

перетасовка и последующий выбор бэкендов из начала списка позволяет надежно справляться с последствиями перезапусков и сбоев (сопровождающихся относительно небольшим перетоком соединений), поскольку они явно исключаются из рассмотрения¹³⁴. Однако мы обнаружили, что эта стратегия плохо работает в наиболее актуальных сценариях, поскольку очень неравномерно распределяет нагрузку.

Работая над балансировкой нагрузки, мы первоначально реализовали случайные подмножества и рассчитали ожидаемую нагрузку для разных случаев. В качестве примера рассмотрим следующую ситуацию:

- 300 клиентов;
- 300 бэкендов;
- размер подмножества равен 30 % (каждый клиент соединяется с 90 бэкендами).

Как показано на рис. 20.3, наименее загруженный бэкенд имеет всего 63 % от средней загрузки (57 соединений при среднем значении 90 соединений), а наиболее загруженный — 121 % (109 соединений). В большинстве случаев размер подмножества, равный 30 %, уже превышает то значение, которое мы хотели бы использовать на практике. Рассчитанное распределение нагрузки изменяется каждый раз, когда мы запускаем симуляцию, но общая картина остается той же.

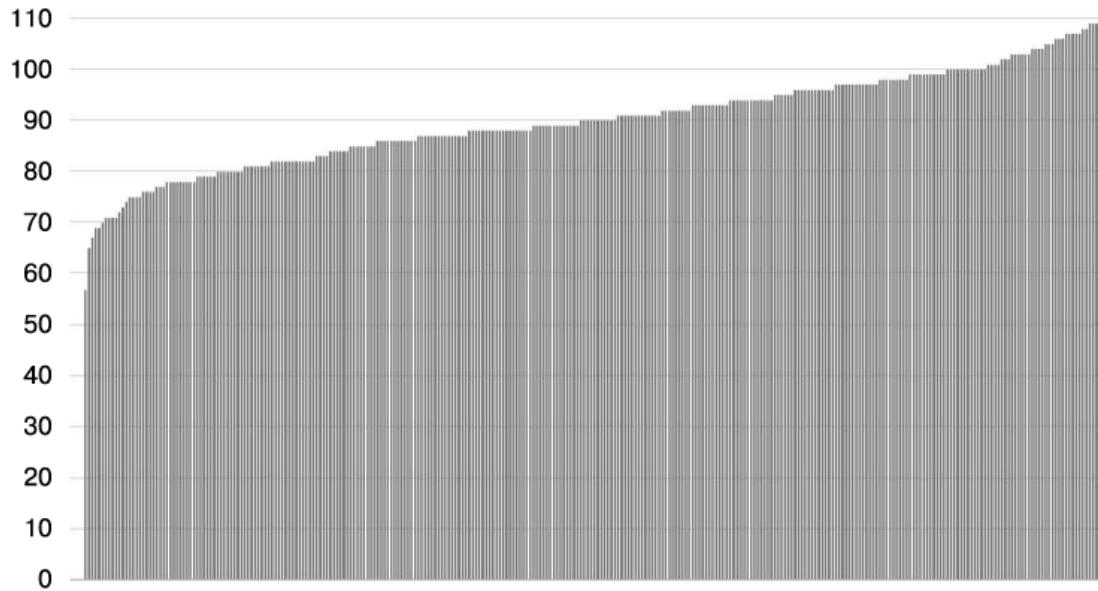


Рис. 20.3. Распределение соединений между 300 клиентами и 300 бэкендами при размере подмножества, равном 30 %

К сожалению, использование меньшего размера подмножества приводит к еще большему дисбалансу. Например, на рис. 20.4 показаны результаты снижения размера подмножества до 10 % (клиент соединяется с 30 бэкендами). В этом случае наименее загруженный бэкенд получает 50 % от средней загрузки (15 соединений), а наиболее загруженный — 150 % (45 соединений).

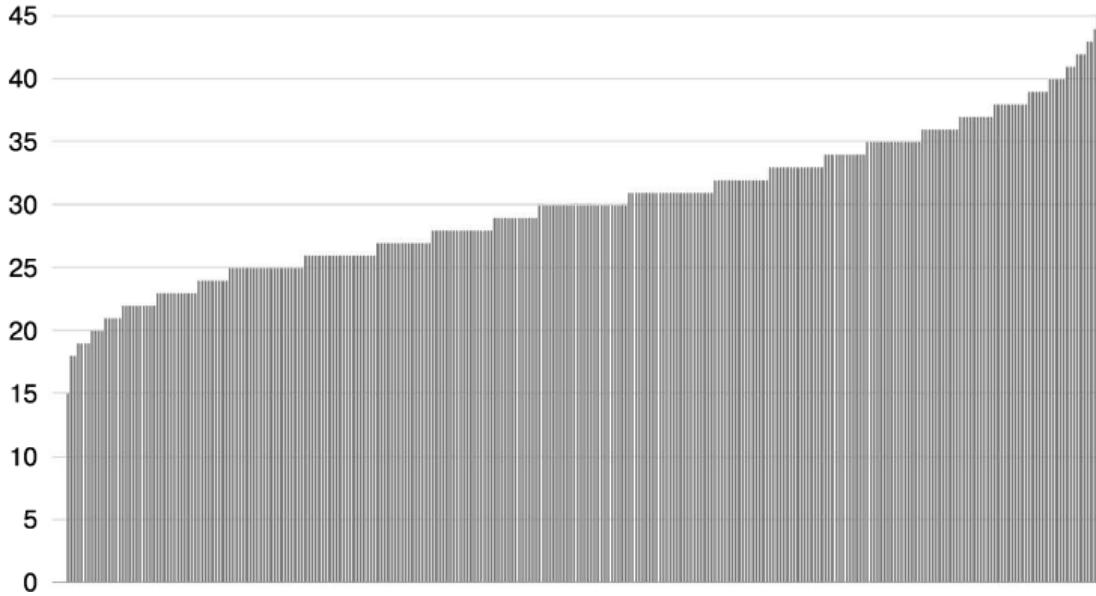


Рис. 20.4. Распределение соединений между 300 клиентами и 300 бэкендами при размере подмножества, равном 10 %

Мы пришли к выводу: чтобы в случайных подмножествах нагрузка распределялась относительно равномерно между всеми доступными задачами, нам понадобятся подмножества размером 75 %. Использовать подмножества такого размера попросту непрактично; колебания количества клиентов, связывающихся с задачей, слишком велики для того, чтобы считать такое формирование подмножеств хорошим решением при работе с крупными системами.

Алгоритм определения подмножества: предопределенное подмножество

Решение Google для преодоления ограничений случайного формирования подмножеств состоит в использовании *детерминированного алгоритма*. Следующий код реализует этот алгоритм (детально он описан далее):

```
def Subset(backends, client_id, subset_size):
    subset_count = len(backends) / subset_size
```

```

# Group clients into rounds; each round uses
the same shuffled list:
round = client_id / subset_count
random.seed(round)
random.shuffle(backends)

# The subset id corresponding to the current
client:
subset_id = client_id % subset_count

start = subset_id * subset_size
return backends(start:start + subset_size)

```

Мы делим *клиентские* задачи на «ряды» (rounds), где i -й «ряд» содержит `subset_count` задач клиентов с подряд идущими номерами, начиная с `subset_count` \times i , а `subset_count` — это количество подмножеств (равное общему количеству задач бэкенда, разделенному на желаемый размер подмножества). Внутри каждого «ряда» каждый бэкенд присваивается ровно одному клиенту (кроме, возможно, последнего «круга», поскольку клиентов уже может не хватать, поэтому некоторые бэкенды не будут присвоены).

Например, если у нас есть 12 задач бэкендов (0, 11) и желаемый размер подмножества равен 3, у нас будет 4 «ряда» из 4 клиентов каждый (`subset_count = 12/3`). Если бы у нас было 10 клиентов, приведенный выше алгоритм мог бы сформировать такие «ряды»:

- «ряд» 0 — (0, 6, 3, 5, 1, 7, 11, 9, 2, 4, 8, 10);
- «ряд» 1 — (8, 11, 4, 0, 5, 6, 10, 3, 2, 7, 9, 1);

- «ряд» 2 — (8, 3, 7, 2, 1, 4, 9, 10, 6, 5, 0, 11).

Следует отметить, что в каждом «ряду» каждый бэкенд назначается ровно одному клиенту (кроме последнего, если клиенты заканчиваются). В этом примере каждый бэкенд назначается двум или трем клиентам.

Список должен быть перетасован; в ином случае клиентам будет присвоена группа задач бэкенда с последовательными номерами, которые могут все стать временно недоступными (например, если обновление задания бэкенда выполняется последовательно в порядке номеров задач). Для разных «рядов» используются разные инициализирующие значения генератора случайных чисел. Если этого не сделать, то после сбоя бэкенда нагрузка, которую он обрабатывал, распределяется лишь между оставшимися бэкендами его подмножества. Если дают сбой и другие бэкенды, то эффект суммируется, и ситуация быстро ухудшается: если N бэкендов подмножества отключены, их нагрузка распределяется между оставшимися (`subset_size - N`) бэкендами. Гораздо лучшим решением будет распределение этой нагрузки между всеми оставшимися бэкендами, для этого перетасовка выполняется заново для каждого «ряда».

Если мы заново перетасуем бэкенды для каждого «ряда», клиенты из одного «ряда» начнут с одинакового перетасованного списка, однако клиенты разных «рядов» будут иметь разные списки. С этого момента алгоритм строит определения подмножеств, исходя из перетасованного списка бэкендов и желаемого размера подмножества. Например:

- `Subset(0)` — от `shuffled_backends(0)` до `shuffled_backends(2)`;

- `Subset(1)` — от `shuffled_backends(3)` до `shuffled_backends(5)`;
- `Subset(2)` — от `shuffled_backends(6)` до `shuffled_backends(8)`;
- `Subset(3)` — от `shuffled_backends(9)` до `shuffled_backends(11)`,

где `shuffled_backend` — это перетасованный список, создаваемый для каждого клиента. Для того чтобы назначить подмножество задаче клиента, мы просто возьмем подмножество, которое соответствует его позиции внутри «ряда» (например $(i \% 4)$ для `client(i)` при четырех подмножествах:

- `client(0), client(4), client(8)` будут использовать `subset(0)`;
- `client(1), client(5), client(9)` будут использовать `subset(1)`;
- `client(2), client(6), client(10)` будут использовать `subset(2)`;
- `client(3), client(7), client(11)` будут использовать `subset(3)`.

Поскольку клиенты разных «рядов» будут применять разные значения для `shuffled_backends` (и, соответственно, для `subset`), а клиенты внутри «рядов» используют разные подмножества, нагрузка соединения будет распределена

равномерно. В случаях, когда общее количество бэкендов не делится нацело на желаемый размер подмножества, мы позволим новым подмножествам быть чуть больше, чем другим, но в большинстве случаев количество клиентов, назначенных бэкенду, будет отличаться не более чем на единицу.

Как показано на рис. 20.5, такое распределение для предыдущего примера, где 300 клиентов соединяются с 10–300 бэкендами, дает очень хорошие результаты: каждый бэкенд получает одинаковое количество соединений.

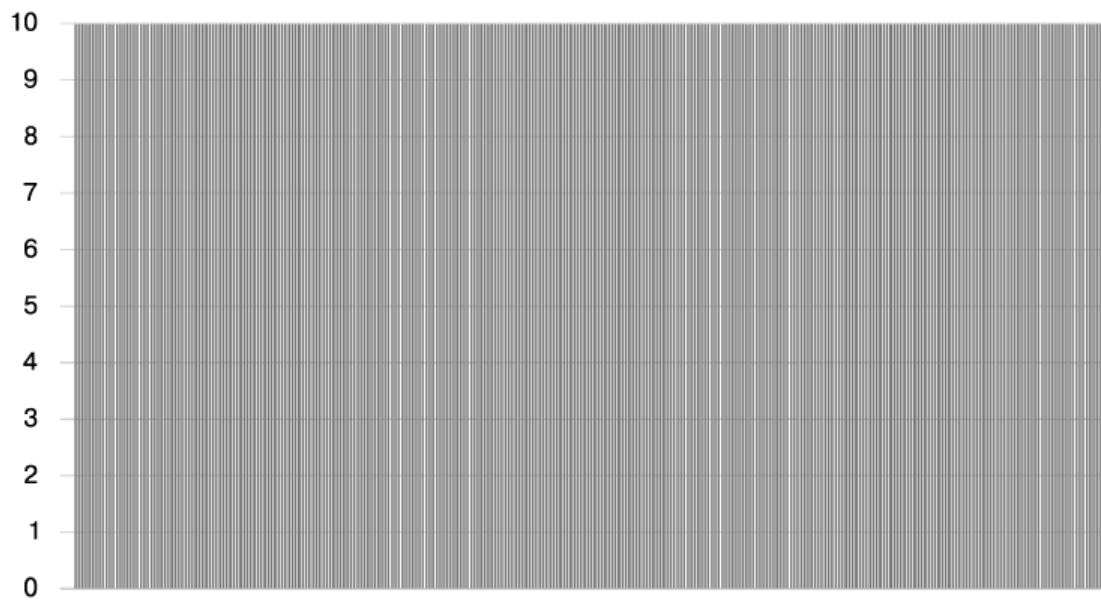


Рис. 20.5. Распределение соединений с 300 клиентами и предопределенными подмножествами, состоящими из 10–300 бэкендов

Политики балансировки нагрузки

Теперь, когда мы рассмотрели основы того, как отдельная задача клиента обслуживает набор соединений с известной работоспособностью, рассмотрим *политики балансировки нагрузки*. Они представляют собой механизмы и правила, используемые задачами клиентов для выбора задачи бэкенда,

которой будет передан запрос. Большой вклад в сложность политик балансировки вносят распределенный характер процесса принятия решений, когда клиенту нужно в реальном времени решить, какой бэкенд следует использовать для каждого запроса, имея лишь частичную и/или устаревшую информацию о состоянии бэкенда.

Политики балансировки нагрузки могут быть очень простыми. Они могут не принимать во внимание состояние бэкендов (например, алгоритм *Round Robin* — циклический алгоритм) или же могут действовать на основе информации о бэкендах (например, *Least-Loaded Round Robin* — циклический алгоритм поиска наименее загруженного элемента или *Weighted Round Robin* — взвешенный циклический алгоритм).

Простой *Round Robin*

Простой подход к балансировке нагрузки заключается в том, что каждый клиент отправляет очередные запросы последовательно каждой задаче бэкенда своего подмножества, кроме тех, с которыми невозможно соединиться, или находящимися в состоянии «хромой утки». На протяжении многих лет мы использовали этот подход, и он до сих пор работает во многих сервисах.

К сожалению, несмотря на то что алгоритм *Round Robin* выигрывает у конкурентов за счет своей простоты, и при этом работает значительно лучше метода произвольного выбора бэкендов, результаты этой политики могут оказаться очень плохими.

Реальные показатели зависят от многих факторов, например, от изменяющейся «стоимости» запроса и разнообразия машин, и мы обнаружили, что использование алгоритма *Round Robin* может привести к почти двукратному

разбросу уровня потребления ресурсов процессора между наименее и наиболее загруженными задачами. Такой разброс крайне расточителен и может происходить по многим причинам, включая:

- малый размер подмножества;
- изменение стоимости запросов;
- разнообразие машин;
- непредсказуемые факторы, влияющие на производительность.

Малый размер подмножества

Одна из самых простых причин, по которым алгоритм Round Robin плохо распределяет нагрузку, заключается в том, что все его клиенты могут генерировать разное количество запросов. Появление разного количества запросов среди клиентов особенно вероятно, когда совершенно разные процессы работают на одних бэкендах. В этом случае, особенно если вы используете относительно небольшой размер подмножеств, бэкенды подмножеств для клиентов, которые генерируют наибольший объем трафика, естественным образом будут загружены больше.

Изменение стоимости запросов

Многие сервисы обрабатывают запросы, которые требуют разного объема ресурсов для своего выполнения. На практике мы обнаружили, что семантика многих сервисов Google такова, что самые «дорогие» запросы потребляют в 1000 раз больше ресурсов (или даже больше), чем самые «дешевые».

Балансировка нагрузки с помощью алгоритма Round Robin еще больше затрудняется, когда стоимость запроса нельзя предсказать заранее. Например, запрос вроде «получить все электронные письма, полученные пользователем XYZ за последний день» может быть как очень дешевым (если пользователь в течение дня получил мало писем), так и крайне дорогим.

Балансировка нагрузки в системе, имеющей большие расхождения в потенциальной стоимости запроса, очень проблематична. Возможно, придется скорректировать интерфейсы сервисов, чтобы функционально ограничить трудоемкость запросов. Например, в случае рассмотренного выше запроса на получение электронных писем вы можете предусмотреть в интерфейсе разбиение на страницы и изменить семантику запроса на «вернуть 100 последних писем (или меньше), полученных пользователем XYZ за последний день». К сожалению, ввести такие изменения семантики зачастую очень сложно. Это не только потребует изменения всего клиентского кода, но и повлечет за собой потенциальные проблемы со стабильностью. Например, пользователь может получать новые электронные письма или удалять их в тот момент, когда клиент получает письма постранично. В этом случае клиент, который просто проходит по результатам и объединяет ответы (вместо того, чтобы выполнять разбиение на страницы, основываясь на фиксированном представлении данных), получит, скорее всего, нестабильное представление, дублируя одни сообщения и/или опуская другие.

Для того чтобы интерфейс (и его реализация) оставались простыми, сервисы зачастую позволяют самым «дорогим» запросам потреблять в 100, 1000 или даже 10 000 раз больше ресурсов, чем самым «дешевым». Однако изменение требований к ресурсам для выполнения запросов означает, что

некоторые задачи бэкенда будут менее удачливыми, чем другие — им придется обработать больше дорогостоящих запросов. Степень влияния этого на балансировку нагрузки зависит от того, насколько дорогостоящими являются самые «дорогие» запросы. Например, для одного из наших бэкендов Java запросы потребляют в среднем примерно 15 миллисекунд процессорного времени, но на выполнение некоторых запросов может потребоваться до 10 секунд. Каждая задача этого бэкенда резервирует несколько ядер процессора, что снижает задержку отклика за счет параллельного выполнения некоторых вычислений. Но, несмотря эти зарезервированные ядра, при получении одного из таких «тяжелых» запросов, его загрузка на несколько секунд значительно возрастает. Плохо ведущая себя задача может израсходовать всю память или даже полностью перестать отвечать (например, из-за засорения памяти), но даже в обычном случае (то есть когда бэкенд имеет достаточное количество ресурсов, и его загрузка нормализуется, как только завершится выполнение длительного запроса) задержка выполнения других запросов может пострадать из-за необходимости конкурировать за использование ресурсов с дорогостоящим запросом.

Разнообразие машин

Еще одной сложностью использования простого алгоритма Round Robin является то, что не все машины в одном дата-центре одинаковы. В любом дата-центре могут найтись машины с разной производительностью, и поэтому выполнение одного и того же запроса на разных машинах может занимать разное время.

Решение проблемы разнообразия машин — без требования строгой гомогенности — многие годы было одной из насущных задач компании Google. Теоретически решение, позволяющее

работать с гетерогенными вычислительными ресурсами во всем парке техники, выглядит просто: мы масштабируем резервирование ЦП в зависимости от типа процессора и машины. Однако на практике реализация этого решения требовала больших усилий, поскольку планировщик должен был пересчитывать соответствие ресурсов, основываясь на средней производительности машин для выбранных сервисов. Например, два ЦП на машине X («медленной» машине) эквивалентны 0,8 ЦП на машине Y («быстрой» машине). Имея эту информацию, планировщик должен скорректировать резервирование ЦП для процесса, учитывая коэффициент соответствия и тип машины, на которой запланировано выполнение этого процесса. Пытаясь справиться с этой проблемой, мы ввели виртуальный элемент для измерения уровня CPU, который назвали GCU (Google Compute Unit — вычислительный элемент Google). GCU стал стандартом для моделирования возможностей CPU и использовался для описания соотношения между архитектурой каждого процессора в наших data-центрах с соответствующим CGU, основываясь на их производительности.

Непредсказуемые факторы, влияющие на производительность

Возможно, самым крупным фактором, усложняющим использование простого алгоритма Round Robin, было то обстоятельство, что машины — или, говоря более точно, производительность задач бэкендов на разных машинах — могли значительно отличаться из-за некоторых *непредсказуемых* факторов.

Рассмотрим два таких фактора.

- *Соседи-антагонисты.* Другие процессы (зачастую никак не связанные с вашими и запущенные другими командами)

могут оказывать значительное влияние на производительность ваших процессов. Мы наблюдали, как это меняло производительность на целых 20 %. Причиной колебаний производительности обычно бывает конкуренция за использование разделяемых ресурсов, таких как место в кэширующей памяти или полоса пропускания сети, причем не самым очевидным образом. Например, если задержка отклика для исходящих от задачи бэкенда запросов увеличивается (из-за конкуренции за ресурсы сети с соседом-антагонистом), количество активных запросов также будет расти, что может вызвать более интенсивное выполнение сборки мусора.

- *Перезапуски задач.* Когда задача перезапускается, ей часто требуется много ресурсов на протяжении нескольких минут. Мы заметили, что такое состояние чаще влияет на платформы вроде Java, которые динамически оптимизируют код. В ответ на это мы добавили в логику немного серверного кода — мы задерживаем серверы в состоянии «хромой утки» и предварительно «прогреваем» их (заставляем срабатывать эти оптимизации) какое-то время после запуска до тех пор, пока их производительность не достигнет номинальной. Эффект от перезапуска задач может стать заметной проблемой, если нужно обновлять много серверов (например, устанавливать новые сборки, что потребует перезапуска этих задач) каждый день.

Если ваша политика балансировки нагрузки не может адаптироваться к непредвиденным ограничениям производительности, вы однозначно получите неоптимальное распределение нагрузки при масштабировании системы.

Least-Loaded Round Robin

Альтернативой простому алгоритму Round Robin является подход, при котором каждая задача клиента должна отслеживать количество активных запросов к каждой задаче бэкенда своего подмножества и использовать алгоритм Round Robin среди множества задач с минимальным количеством активных запросов.

Например, пусть клиент использует подмножество задач бэкенда от t_0 до t_9 , и количество активных запросов для них составляет:

t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
2	1	0	0	1	0	2	0	0	1

При поступлении нового запроса клиент должен отфильтровать список потенциальных задач бэкенда так, чтобы в нем остались лишь задачи с наименьшим количеством соединений (t_2 , t_3 , t_5 , t_7 и t_8), и затем выбрать из него бэкенд. Предположим, что был выбран бэкенд t_2 . Таблица состояний соединений клиента теперь будет выглядеть следующим образом:

t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
2	1	1	0	1	0	2	0	0	1

Предположим, что ни один из текущих запросов не был завершен, поэтому для следующего запроса пул кандидатов выглядит как t_3, t_5, t_7 и t_8 .

Переместимся в тот момент времени, когда у нас появится четыре новых запроса. Мы все еще предполагаем, что ни один запрос за это время не завершится, поэтому после приема этих запросов таблица соединений будет выглядеть следующим образом:



t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
2	1	1	1	1	1	2	1	1	1

В этот момент в множестве кандидатов будут рассматриваться все задачи, кроме t0 и t6. Однако если запрос к задаче t4 завершится, ее текущим состоянием станет «0 активных запросов» и новый запрос будет направлен ей.

В этой реализации также используется алгоритм Round Robin, но он применяется только к множеству задач с минимальным количеством активных запросов. Без такой фильтрации политика может не суметь распределить запросы достаточно хорошо, чтобы избежать простоя части доступных бэкендов. Идея, лежащая в основе этой политики, заключается в том, что у сильно загруженных задач задержка отклика скорее всего окажется больше, чем у свободных, а эта стратегия естественным образом снимает нагрузку с этих загруженных задач.

С учетом вышесказанного, мы узнали (сложным способом!) об одной очень опасной ловушке, которую таит подход Least-Loaded Round Robin: если задача совсем неработоспособна, она может начать возвращать 100 % ошибок. В зависимости от природы этих ошибок они могут иметь очень низкую задержку отклика; зачастую сообщить «Я неисправен!» можно гораздо быстрее, чем выполнить запрос. В результате клиенты могут начать отправлять очень много трафика неработоспособным задачам, ошибочно считая их доступными, однако задача просто быстро генерирует сбои! В таких случаях мы говорим, что задача теперь «поглощает трафик».

К счастью, этой ловушки можно избежать, слегка модифицировав политику и начав подсчитывать недавние ошибки так же, как если бы они были активными запросами. Таким образом, если задача бэкенда становится

неработоспособной, политика балансировки нагрузки начинает отводить от нее нагрузку так же, как если бы она отводила нагрузку от перегруженной задачи.

Алгоритм Least-Loaded Round Robin имеет два важных ограничения.

- Количество активных запросов может оказаться не очень хорошим индикатором возможностей заданного бэкенда. Многие запросы проводят значительную часть своей жизни, ожидая ответа сети (например, ожидая ответов на запросы, которые они инициировали для других бэкендов), и лишь очень малое время они действительно выполняются. Например, одна задача бэкенда может обработать в два раза больше запросов, чем другая (поскольку она запущена на машине с вдвое более быстрым процессором по сравнению с остальными), но задержка отклика для ее запросов будет почти равна задержке отклика для другой аналогичной задачи (поскольку запросы проводят большую часть своей жизни, ожидая ответа сети). В этом случае, поскольку заблокированный в ожидании ввод/вывод обычно потребляет нулевое время процессора, очень мало оперативной памяти и не затрагивает полосу пропускания сети, мы все еще хотим отправлять вдвое более быстрому бэкенду вдвое больше запросов. Однако политика Least-Loaded Round Robin будет считать обе задачи бэкенда одинаково загруженными.
- Количество активных запросов в каждом клиенте не учитывает запросы других клиентов к этим бэкендам. Соответственно, каждая из задач клиентов имеет очень ограниченное представление о состоянии своей задачи бэкенда: она видит только собственные запросы.

На практике мы обнаружили, что крупные сервисы, использующие политику Least-Loaded Round Robin, будут видеть, что их наиболее загруженные задачи бэкенда задействуют в два раза больше ресурсов процессора, чем наименее загруженные. Следовательно, она работает почти так же плохо, как и простая политика Round Robin.

Weighted Round Robin

Weighted Round Robin (взвешенный циклический алгоритм) — это важная политика балансировки нагрузки, которая улучшает предыдущие версии, добавляя в процесс принятия решения информацию, предоставленную бэкендом.

Идея, лежащая в основе политики Weighted Round Robin, довольно проста: каждая задача клиента ведет подсчет «возможностей» для каждого бэкенда своего подмножества. Запросы распределяются так же, как и при политике Round Robin, но задачи клиентов пропорционально взвешиваются распределения запросов по бэкендам. Для каждого ответа (включая ответы на проверку работоспособности) бэкенды включают текущее наблюдаемое соотношение запросов и ошибок в секунду в дополнение к показателю используемости (обычно использованию процессора). Клиенты периодически корректируют подсчитанные «возможности» для того, чтобы выбирать задачи бэкенда, основываясь на их текущем количестве обработанных успешных запросов и стоимости выполнения; запросы, завершившиеся сбоем, приводят к штрафам, которые повлияют на будущие решения.

Алгоритм Weighted Round Robin достаточно хорошо показал себя на практике и значительно снизил разницу между наиболее и наименее загруженными задачами. На рис. 20.6 представлены соотношения использования процессора для произвольного подмножества задач бэкенда до и после

перехода клиентов с алгоритма Least-Loaded Round Robin на алгоритм Weighted Round Robin. Разница между наименее и наиболее загруженными задачами значительно уменьшилась.

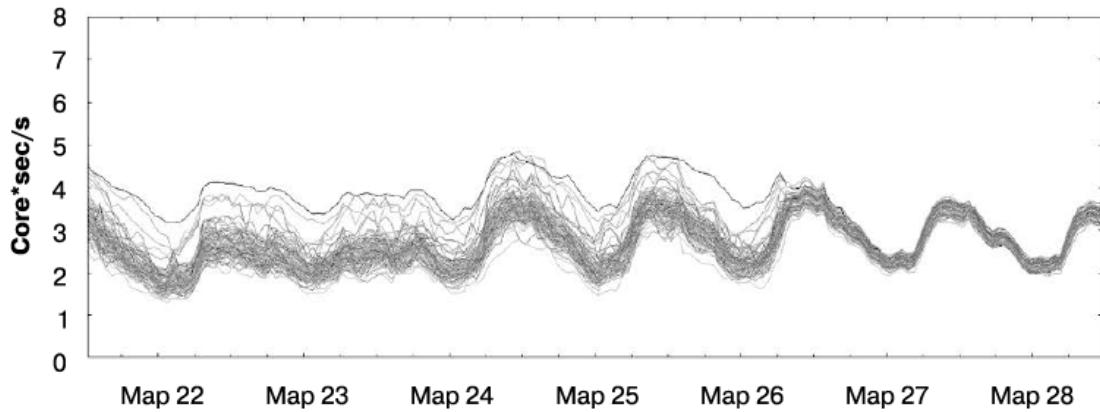


Рис. 20.6. Распределение уровня использования процессора до и после перехода на алгоритм Weighted Round Robin

[133](#) Здесь под неработоспособными (unhealthy) понимаются задачи (процессы), которые не работают из-за нехватки ресурсов, но не из-за ошибки, сбоя и т.п. — Примеч. пер.

[134](#) Предположительно имеется в виду исключение из рассмотрения тех задач, которые подверглись сбою или перезапуску. — Примеч. пер.

21. Справляемся с перегрузками

Автор — Александр Фореро Куэрво

Под редакцией Сары Чевис

Целью любой политики балансировки нагрузки является отсутствие перегрузок. Но независимо от того, насколько эффективна политика балансировки нагрузки, в конечном счете какая-то часть вашей системы окажется перегруженной. Мягкая обработка состояний перегрузки очень важна для того, чтобы обслуживающая система надежно работала.

Одним из вариантов обработки перегрузки является выдача деградировавших ответов. Эти ответы не такие точные, как обычные, и могут содержать меньше данных, но их легче сгенерировать. Например:

- вместо поиска по всей текстовой базе данных для предоставления лучших результатов поискового запроса вы выполняете поиск по небольшому количеству вариантов;
- вы полагаетесь только на локальную копию результатов, которая может быть не полностью обновлена, но дешевле использовать ее, а не основное хранилище.

Однако в условиях серьезных перегрузок сервис даже может не иметь возможности вычислять и выдавать деградировавшие ответы. В такой ситуации у него не будет других вариантов, кроме возвращения ошибок. Одним из способов решения этой проблемы является балансировка трафика между data-центрами таким образом, чтобы ни один из них не принимал больше трафика, чем может обработать. Например, если в data-центре запущены 100 задач бэкенда и каждая может

обработать до 500 запросов в секунду, алгоритм балансировки нагрузки не позволит отправить этому дата-центру больше 50 000 запросов в секунду. Однако даже этого ограничения может оказаться недостаточно для того, чтобы избежать перегрузки, если вы работаете с крупномасштабными сервисами. В конечном счете лучше всего создавать клиенты и бэкенды, которые могут справляться с ограничениями по ресурсам мягко: выполнять перенаправление везде, где это возможно, отправлять деградировавшие результаты, когда необходимо, и обрабатывать ошибки ресурсов прозрачно, когда все остальные этого не могут.

Ловушки понятия «запросов в секунду»

Различные запросы имеют совершенно разные требования к ресурсам. Стоимость запроса может изменяться в зависимости от произвольных факторов, таких как код клиента, который их отправляет (для сервисов, которые могут иметь разные клиенты), или даже время суток (например, пользователи, работающие из дома, против пользователей, работающих из офиса, или интерактивный трафик конечных пользователей против пакетного трафика).

Этот урок дался нам нелегко: моделирование производительности с помощью понятия «запросы в секунду» или использование статической функциональности запросов, которые считаются прокси для потребляемых ими ресурсов (например, «сколько ключей считывают запросы»), зачастую являются неудачными сценариями. Даже если в какой-то момент времени они сработают как следует, соотношение может измениться. Иногда такие изменения проходят постепенно, а иногда — мгновенно (например, в новой версии ПО некоторая функциональность запросов требует

значительно меньше ресурсов). Постоянно изменяющаяся величина — плохой показатель для проектирования и реализации балансировки нагрузки.

Более удачный вариант — измерять производительность непосредственно в доступных ресурсах. Например, для одного сервиса в заданном дата-центре у вас зарезервированы 500 ядер процессоров и 1 Тбайт памяти. Естественно, гораздо лучше использовать эти значения непосредственно для моделирования производительности дата-центра. Мы зачастую говорим о стоимости запроса, когда хотим сказать о нормализованной мере того, сколько времени процессора он потребил (для разных архитектур процессоров учитывается разница в производительности).

Мы обнаружили, что в большинстве случаев (конечно, не во всех) удобно ориентироваться на показатель потребления ресурсов процессора.

- На платформах, для которых предусмотрена сборка мусора, нехватка памяти естественным образом преобразуется в повышенное потребление ресурсов процессора.
- На других платформах возможно выполнение поставок других ресурсов таким образом, что они вряд ли закончатся до того, как будут исчерпаны ресурсы процессора.

В случаях, когда чрезмерные поставки непроцессорных ресурсов крайне дороги, мы рассматриваем потребление ресурсов для каждой системы отдельно.

Лимиты потребителей

Очень важно выработать перечень действий на случай *глобальной* перегрузки. В идеальном мире, где команды осторожно координируют свой запуск с владельцами зависимостей своих бэкендов, глобальная перегрузка никогда не наступает и производительности сервисов бэкендов всегда достаточно для обслуживания потребителей. К несчастью, мы живем не в идеальном мире. В реальности глобальные перегрузки происходят довольно часто (это особенно верно для внутренних сервисов, для которых большое количество команд запускают множество клиентов).

Когда глобальная перегрузка все-таки *происходит*, очень важно, чтобы сервис возвращал ошибки только неправильно ведущим себя потребителям, не затрагивая остальных. Для того чтобы этого добиться, владельцы сервисов поставляют для них производительность, основываясь на их используемости, и определяют квоты для потребителей в соответствии с этими данными.

Например, если сервис бэкенда имеет 10 000 процессоров, размещенных по всему миру (в разных дата-центрах), их лимиты для потребителей могут выглядеть так:

- сервису Gmail можно использовать до 4000 секунд процессорного времени за секунду реального времени;
- сервису Calendar — до 4000 секунд процессорного времени за секунду реального времени;
- сервису Android — до 3000 секунд процессорного времени за секунду реального времени;
- сервису Google+ — до 2000 секунд процессорного времени за секунду реального времени;

- любому другому потребителю — до 500 секунд процессорного времени за секунду реального времени.

Обратите внимание на то, что сумма этих чисел больше чем 10 000. Владельцы сервиса полагаются на то, что ситуация, когда все их потребители запросят все полагающиеся им ресурсы, маловероятна.

Мы собираем глобальную информацию об использовании ресурсов в реальном времени всеми задачами бэкенда и применяем эти данные для того, чтобы раздвинуть границы индивидуальных задач бэкенда. Подробное рассмотрение системы, которая реализует эту логику, в данной книге не предусмотрено, но стоит сказать, что мы пишем большие объемы кода для ее реализации в задачах бэкенда. Интересным фрагментом головоломки является вычисление в реальном времени количества ресурсов, особенно процессорных, потребляемых каждым отдельным запросом. Такие расчеты особенно запутаны для серверов, которые не придерживаются модели «поток на запрос», где пул потоков выполняет разные части всех поступающих запросов, применяя неблокирующие API.

Регулирование на стороне клиента

Когда клиент превышает свою квоту, задача бэкенда должна быстро отклонять запросы, ожидая, что возврат ошибки «для клиента закончилась квота» потребляет значительно меньшее количество ресурсов, чем реальная обработка запроса и отправка корректного ответа. Однако эта логика действует не для всех сервисов. Например, отклонять запросы, для обслуживания которых требуется выполнить простой поиск в памяти (где нагрузка обработки протокола запроса/ответа

значительно выше нагрузки для выработки ответа), так же дорого, как и их выполнять. И даже в случае, когда отказ от запроса экономит значительные ресурсы, эти запросы *все еще* потребляют некоторое их количество. Если отклоненных запросов много, эти числа довольно быстро увеличиваются. В таких случаях бэкенд может оказаться перегруженным, даже если большое количество ресурсов процессора затрачивается на то, чтобы отклонять запросы!

Эту проблему можно решить регулированием количества запросов на стороне клиента [135](#). Когда клиент обнаруживает, что значительная часть его недавних запросов отклоняется из-за ошибки «закончилась квота», он начинает самостоятельно регулировать и ограничивать количество генерируемого им исходящего трафика. Запросы, вышедшие за этот предел, даже не достигнут сети.

Мы реализовали такой подход с помощью приема, который называем *адаптивным регулированием*. В частности, каждая задача клиента на протяжении 2 минут сохраняет следующую информацию:

- `requests` — количество запросов, выполненное на уровне приложения (на клиенте на основе системы адаптивного регулирования);
- `accepts` — количество запросов, принятых бэкеном.

При нормальных условиях эти два значения равны. Как только бэкенд начинает отклонять трафик, значение `accepts` становится меньше значения `requests`. Клиенты могут продолжать отправлять запросы на бэкенд до тех пор, пока значение `requests` не превысит значение `accepts` в K раз. По достижении этой точки клиент начинает самостоятельно

регулировать себя, и новые запросы будут отклоняться локально (на клиенте) с вероятностью, рассчитанной по следующей формуле:

$$\max\left(0, \frac{\text{requests} - K \text{accepts}}{\text{requests} + 1}\right). \quad (21.1)$$

Когда сам клиент начнет отклонять запросы, значение `requests` по-прежнему будет превышать значение `accepts`. Это может показаться нелогичным, учитывая, что локально отклоненные запросы не отправляются на бэкенд, но такое поведение более предпочтительно. По мере увеличения этого соотношения (относительно уровня, при котором бэкенд принимает запросы) мы хотим повысить вероятность того, что новые запросы не будут срабатывать.

Мы обнаружили, что адаптивное подавление хорошо работает на практике, это привело к стабильным соотношениям запросов в целом. Даже при сильной перегрузке бэкенд начинает отклонять по одному запросу на каждый уже обрабатываемый им запрос. Одним из преимуществ такого подхода является то, что решение принимается задачей клиента на базе только локальной информации и с использованием относительно простой реализации: не существует зависимостей или штрафов, дополняющих задержку обработки данных.

Для сервисов, для которых стоимость обработки запроса примерно равна стоимости его отклонения, выделение примерно половины ресурсов бэкенда на отклонение запросов может быть неприемлемым. В этом случае решение выглядит просто: модифицируйте множитель K (задайте ему, например, значение 2) в формуле (21.1), выражающей вероятность отклонения запросов. Таким образом:

- уменьшение множителя сделает адаптивное подавление более агрессивным;
- увеличение множителя сделает адаптивное подавление менее агрессивным.

Например, вместо того, чтобы клиент саморегулировался, когда `requests = 2 · accepts`, он будет делать это при равенстве `requests = 1,1 · accepts`. Уменьшение множителя до 1,1 означает, что будет отклоняться только один запрос на десять обработанных.

Обычно мы используем множитель $2x$. Позволив количеству запросов, превышающему ожидаемое, достичь бэкенда, мы тратим впустую больше ресурсов бэкенда, но в то же время ускоряем распространение состояния от бэкенда клиентам. Например, если бэкенд решит прекратить отклонять трафик задач клиента, задержка обнаружения этого состояния задачами клиента уменьшится.

Еще одна проблема заключается в том, что подавление на стороне клиента может оказаться неэффективным для клиентов, которые очень нерегулярно отправляют запросы бэкендам. В этом случае объем информации, которую каждый клиент имеет о состоянии бэкенда, значительно снижается и приемы увеличения ее доступности, как правило, дорогие.

Критичность

Критичность — это еще одно понятие, которое мы считаем очень полезным в контексте глобальных квот и регулирования количества запросов. Запрос, сделанный на бэкенд, связан с одним из четырех возможных значений критичности в зависимости от того, насколько критичным мы его считаем.

- CRITICAL_PLUS. Зарезервирован для наиболее критичных запросов, сбои в которых значительно повлияют на пользователей.
- CRITICAL. Стандартное значение для запросов, отправленных задачами в производственной среде. Эти запросы оказывают влияние на пользователей, но оно значительно меньше, чем влияние запросов с критичностью CRITICAL_PLUS. Сервисы должны иметь достаточную производительность для всего трафика, помеченного как CRITICAL и CRITICAL_PLUS.
- SHEDDABLE_PLUS. Трафик, частичная недоступность которого ожидается. Это значение по умолчанию имеют пакетные задачи, которые могут повторять свои запросы спустя несколько минут или часов.
- SHEDDABLE. Трафик, для которого часто ожидается частичная, а иногда и полная недоступность.

Мы обнаружили, что для моделирования практически любого сервиса достаточно четырех значений. И много раз обсуждали возможность добавления новых значений, поскольку это позволило бы нам более качественно классифицировать запросы. Однако определение дополнительных значений потребует большего количества ресурсов для работы множества систем, следящих за критичностью. Мы сделали критичность основным понятием системы RPC и хорошо поработали, интегрировав ее в механизмы управления, чтобы ее можно было учитывать, когда происходит реакция на перегрузки.

- Когда у клиента заканчивается глобальная квота, задача бэкенда будет отклонять запросы с заданной критичностью, если она уже отклоняет все запросы с более низкой критичностью (фактически лимиты для клиентов, поддерживаемые нашей системой и описанные ранее, могут быть установлены для разных значений критичности).
- Когда задача перегружена сама по себе, она будет быстрее отклонять запросы с меньшей критичностью.
- Система адаптивного подавления также сохраняет отдельную статистику для каждого уровня критичности.

Критичность запроса не зависит от требований к задержке обработки, поэтому используется лежащий в ее основе показатель качества обслуживания (quality of service, QoS). Например, когда после ввода пользователем поискового запроса система отображает результаты поиска или рекомендации, лежащие в основе этого процесса, запросы можно значительно сегментировать (если система перегружена, приемлемо не отображать результаты), но они, как правило, имеют строгие требования к задержке обработки данных.

Мы также значительно расширили систему RPC для того, чтобы распространять значение критичности автоматически. Если бэкенд получает запрос A и в качестве одного из этапов его выполнения отправляет запросы B и C другим бэкендам, то запросы B и C по умолчанию будут использовать тот же уровень критичности, что и запрос A.

В прошлом многие системы компании Google выработали свои представления о критичности, которые не всегда подходили для других сервисов. Стандартизировав и

распространив критичность как часть системы RPC, мы теперь можем единообразно задавать значение критичности в определенных точках. Это означает, что мы можем быть уверены в том, что перегруженные зависимые системы будут принимать во внимание заданный высокий уровень критичности при отклонении трафика независимо от того, насколько глубоко в стеке RPC они находятся. Практическое воплощение заключается в том, что мы задаем критичность настолько точно, насколько это возможно в браузерах и мобильных клиентах — обычно во фронтендах HTTP, которые создают возвращаемый код HTML, — и переопределяем эти значения в особых случаях, когда это имеет смысл сделать в заданных местах стека.

Сигналы загруженности

Реализация защиты от перегрузок на уровне задач основана на понятии *загруженности*. Во многих случаях загруженность — это всего лишь показатель уровня загруженности процессоров (наблюдаемый в настоящее время уровень загруженности процессоров, разделенный на общее количество процессоров, зарезервированных для выполнения этой задачи), но в некоторых случаях мы также учитываем показатели наподобие порции зарезервированной памяти, которая используется в данный момент. По мере того как загруженность приближается к заданным границам, мы начинаем отклонять запросы, основываясь на их критичности (для более высоких уровней критичности задаются более высокие границы). Применяемые сигналы загруженности основаны на состоянии задачи поскольку цель этих сигналов заключается в том, чтобы защитить ее, и у нас есть реализации для разных сигналов.

Наиболее полезный сигнал основан на загрузке процесса, которая определяется системой, называемой *средней нагрузкой исполнителя*.

Для того чтобы найти среднюю нагрузку исполнителя, мы подсчитываем количество активных потоков в процессах. В этом случае под активными потоками подразумеваются потоки, которые либо работают, либо готовы к запуску и ожидают появления свободного процессора. Мы сглаживаем это значение с помощью экспоненциального уменьшения и начинаем отклонять запросы, когда количество активных потоков превышает количество процессоров, доступных для выполнения задачи. Это означает, что сильно разветвленный входящий запрос вызовет кратковременный всплеск загрузки, но сглаживание поглотит большую его часть. Однако если операции не краткосрочные (загрузка растет и долго остается высокой), задача начнет отклонять запросы.

Несмотря на то что средняя нагрузка исполнителя доказала свою полезность, наша система может воспользоваться любым сигналом загруженности, который может понадобиться конкретному бэкенду. Например, мы можем использовать недостаток памяти, который покажет, превысило ли потребление памяти нормальное значение, как еще один возможный сигнал загруженности. Также мы можем сконфигурировать систему таким образом, чтобы она задействовала несколько сигналов и отклоняла запросы, которые превзойдут объединенные (или индивидуальные) целевые значения загруженности.

Обработка ошибок, связанных с перегрузкой

Мы много размышляли не только о мягкой обработке загрузки, но и о том, как клиенты должны реагировать на получение

сообщения об ошибке, связанной с загрузкой. В случае ошибок, связанных с перегрузкой, мы выделяем две возможные ситуации.

- *Большое подмножество задач бэкенда в дата-центре перегружено.* Если система балансировки нагрузки между дата-центрами работает идеально, то есть может распространять состояние и реагировать на изменения трафика мгновенно, это условие не сработает.
- *Небольшое подмножество задач бэкенда в дата-центре перегружено.* Эта ситуация обычно вызывается неидеальностью балансировки нагрузки внутри дата-центра. Например, если недавно задача получила очень дорогой запрос. В этом случае вполне вероятно, что другие дата-центры обладают производительностью, необходимой для обработки запроса.

Если перегружено большое подмножество задач дата-центра, запросы повторять не нужно и на вызывающей стороне должны появиться ошибки (например, следует вернуть ошибку конечному пользователю). Гораздо шире распространена ситуация, когда лишь небольшая часть задач становится перегруженной, в этом случае лучше всего немедленно повторить запрос. Как правило, система балансировки нагрузки между дата-центрами пытается направить трафик клиентов к ближайшему доступному дата-центру. В некоторых случаях ближайший дата-центр находится далеко (например, ближайший доступный клиенту бэкенд находится на другом континенте), но мы обычно стараемся размещать клиентов поближе к их бэкенду. Таким образом, дополнительной задержкой, вызванной повторной отправкой запроса, — она

представляет собой несколько обращений к сети — можно пренебречь.

С точки зрения политики балансировки нагрузки повторные отправки запросов нельзя отличить от новых запросов. Соответственно, мы не используем явную логику, чтобы гарантировать, что повторная попытка отправится на другой бэкенд, — мы полагаемся на то, что это событие произойдет с высокой вероятностью, основываясь на том, что в подмножестве находится много бэкендов. Гарантия того, что все повторные попытки будут отправляться другим задачам бэкенда, потребует излишнего усложнения наших API.

Когда бэкенд перегружен лишь немного, запрос клиента зачастую лучше обслуживается, если бэкенд отклоняет повторные попытки и новые запросы одинаково и быстро. Эти запросы могут быть мгновенно выполнены повторно на другой задаче бэкенда, которая может иметь свободные ресурсы. Последствия того, что мы не разграничиваем повторные попытки и новые запросы на бэкенде, заключаются в том, что повторная отправка запросов становится формой органичной балансировки нагрузки: она перенаправляет нагрузку задачам, которые лучше подходят для обработки запросов.

Решаем выполнить повторную попытку

Когда клиент получает ошибку «Задача перегружена», он должен решить, выполнять ли запрос повторно. У нас есть несколько механизмов, которые позволяют избежать выполнения повторных запросов, если большая порция задач кластера перегружена.

Во-первых, для каждого запроса мы реализуем бюджет повторных попыток — максимум три. Если запрос уже дал сбой три раза, мы позволяем ошибке попасть к вызывающей

стороне. Обоснование такого поведения заключается в следующем: если запрос попал на перегруженные задачи три раза подряд, то маловероятно, что следующая попытка будет плодотворной, поскольку весь дата-центр, скорее всего, перегружен.

Во-вторых, мы реализуем *бюджет повторных попыток для каждого клиента*. Каждый клиент отслеживает соотношение запросов и повторных попыток. Запрос будет выполняться до тех пор, пока оно не превышает 10 %. Обоснование таково: если перегружен лишь небольшой объем задач, потребность в выполнении повторных запросов будет довольно небольшой.

В качестве конкретного примера (худшего случая) предположим, что дата-центр принимает небольшое количество запросов и отклоняет остальные. Пусть x — общий уровень запросов, отправленных дата-центру в соответствии с логикой клиента. Из-за повторных запросов общее их количество значительно увеличится — почти до $3x$. Хотя мы и ограничили рост, вызванный повторными попытками, тройное увеличение количества запросов выглядит значительно, особенно если стоимость отклонения запроса существенна по сравнению со стоимостью его обработки. Однако создание слоев бюджета повторных запросов клиента (соотношение повторных запросов равно 10 %) в общем случае снижает рост до $1,1x$ — это значительное улучшение.

Третий подход заключается в том, что клиенты включают в метаданные счетчик количества попыток отправки запроса. Например, при первой попытке счетчик равен 0 и увеличивается при каждой попытке до тех пор, пока не получит значение 2. В этот момент бюджет повторных запросов указывает прекратить выполнять запросы повторно. Бэкенды хранят гистограммы этих значений за недавнее время. Когда бэкенду нужно отклонить запрос, он

консультируется с этими гистограммами, чтобы определить вероятность того, что другие бэкенды также перегружены. Если гистограммы показывают значительное количество повторений (это говорит о том, что другие задачи, скорее всего, также перегружены), они вернут ошибку «Перегружен; не повторять» вместо стандартного ответа «Задача перегружена», который вызывает отправку повторных запросов.

На рис. 21.1 показано количество попыток для каждого запроса, полученного заданной задачей бэкенда в разных ситуациях в рамках скользящего окна (в соответствии с 1000 исходных запросов, не считая повторных). Для простоты бюджет повторных запросов клиента будет проигнорирован (то есть эти числа показывают, что единственным ограничителем количества повторных запросов является бюджет, равный трем попыткам), и подмножество может каким-то образом изменить эти числа.



Рис. 21.1. Гистограммы попыток при разных условиях

Более крупные системы, как правило, представляют собой большие стеки систем, которые, в свою очередь, могут зависеть друг от друга. При такой архитектуре запросы должны

выполняться повторно только на том слое, который находится выше слоя, который их отклоняет. Когда мы решаем, что заданный запрос не может быть обслужен и его нельзя выполнить повторно, мы используем ошибку «Перегружен; не повторять» и поэтому избегаем комбинаторного взрыва повторений.

Рассмотрим пример, показанный на рис. 21.2 (на практике наши стеки зачастую гораздо более сложны). Представим, что фронтенд базы данных перегружен и отклоняет запрос. В этом случае происходит следующее.

- Бэкенд Б выполнит запрос повторно в соответствии с заданным планом.
- Как только бэкенд Б определит, что запрос к фронтенду базы данных не может быть обслужен, например, поскольку запрос уже отклоняли три раза, он должен вернуть бэкенду А ошибку «Перегружен; не повторять» или деградировавшие ответы, предполагая, что может дать умеренно полезный ответ, даже если запрос к базе данных дал сбой.
- Бэкенд А имеет те же самые варианты для запроса, который получил от фронтенда, и будет действовать точно так же.

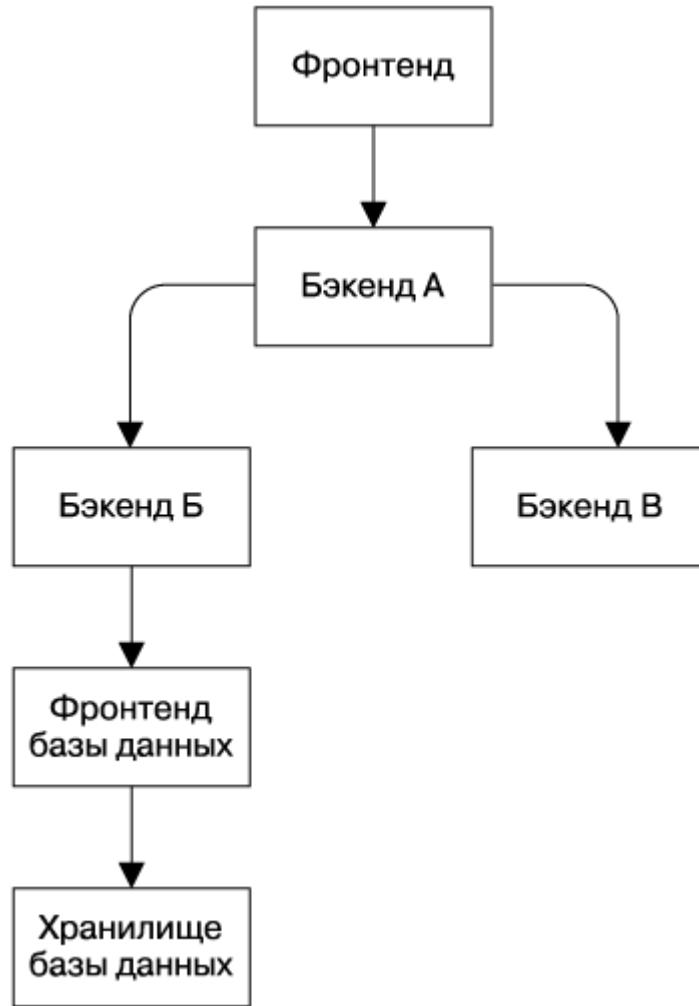


Рис. 21.2. Стек зависимостей

Основной смысл процесса заключается в том, что запрос, давший сбой на фронтенде базы данных, будет выполнен повторно только бэкендом Б — слоем, который находится непосредственно над фронтендром. Если повторные запросы будут выполнены на нескольких уровнях, возникнет комбинаторный взрыв.

Нагрузка от соединений

Нагрузка, связанная с соединениями, — это последний фактор, который стоит упомянуть. Мы иногда принимаем во внимание

только ту нагрузку бэкендов, которая вызвана непосредственно получаемыми ими запросами, что является одной из проблем подходов, опирающихся на модель нагрузки, основанную на показателе «запросов в секунду». Однако такой подход мешает увидеть стоимость поддержания большого пула соединений или стоимость быстрого оттока соединений, выраженную в потреблении ресурсов процессора и памяти. Такой информацией в небольших системах можно пренебречь, но это быстро становится проблемой для очень крупных систем RPC.

Как упоминалось ранее, протокол RPC требует, чтобы неактивные клиенты периодически проверяли состояние. По окончании заданного промежутка времени, в течение которого соединение не использовалось, клиент разрывает соединение TCP и переключается на UDP для проверки состояния.

К сожалению, это поведение вызывает проблемы, когда у вас имеется большое количество задач клиента, отправляющих небольшое количество запросов: проверка состояния соединений может потребовать большего количества ресурсов, чем обслуживание запросов. Подход, заключающийся в тщательной настройке параметров соединения (например, значительном снижении частоты проверок состояния) или даже создании и уничтожении соединений динамически, может значительно улучшить эту ситуацию.

Обработка очередей новых запросов на соединение — это вторая, но связанная с первой проблема. Мы видели, что очереди такого типа возникают, когда очень крупные пакетные задачи создают очень большое количество рабочих задач клиента одновременно. Необходимость согласовывать и поддерживать избыточное количество новых соединений одновременно может легко перегрузить группу бэкендов. Мы, исходя из собственного опыта, определили несколько стратегий, которые помогают справиться с такой загрузкой.

- Учитывать нагрузку в алгоритме балансировки нагрузки между дата-центрами (например, базовая балансировка нагрузки основана на степени использования кластера, а не на количестве запросов). В этом случае загрузка от запросов, по сути, уходит к другим дата-центрам, которые имеют резерв производительности.
- Указать, что пакетные задачи клиента должны применять отдельный набор *пакетных прокси-задач* бэкендов, которые только организованно перенаправляют запросы к лежащим в их основе бэкендуам и передают клиентам полученные ответы. Поэтому вместо схемы «пакетный клиент → бэкенд» у вас получается схема «пакетный клиент → пакетный прокси → бэкенд». Когда начинает работать самая крупная задача, пострадает только пакетная прокси-задача, защищая реальные бэкенды и клиенты с высоким приоритетом. По сути, пакетный прокси действует как предохранитель. Еще одним преимуществом использования прокси является снижение количества соединений с бэкендом, что улучшает балансировку нагрузки для этого бэкенда (например, прокси-задачи могут задействовать большие подмножества и, возможно, лучше представлять себе состояние задач бэкендов).

Итоги главы

В этой главе и в главе 20 мы рассмотрели, как различные методы (предопределенные подмножества, «взвешенный» циклический алгоритм Weighted Round Robin, регулирование фронтенда, квоты для клиентов и т.д.) могут помочь относительно равномерно распределить нагрузку между задачами в дата-центре. Однако работа этих механизмов зависит от состояния всей распределенной системы и от

распространения информации в ней. И хотя в общем случае они работают достаточно хорошо, при реальной эксплуатации могут иногда возникать ситуации, когда они будут действовать неоптимально.

В результате мы считаем критически важным обеспечить защиту от перегрузки каждой задачи. Проще говоря, задача бэкенда, имеющая техническую возможность обрабатывать трафик на некотором уровне производительности, должна продолжать обрабатывать его на этом уровне, и объем отправленного задаче «лишнего» трафика не должен оказывать существенного влияния на задержку отклика. Из этого следует, что задача бэкенда не должна давать отказы и сбои под нагрузкой. Это утверждение должно оставаться верным для определенного объема трафика — выше 2x или даже 10x от того значения, которое задача имеет техническую возможность обработать. Мы принимаем, что существует определенный порог, после которого система начнет давать сбой, но превысить его должно быть достаточно трудно.

Суть в том, чтобы серьезно отнестись к условиям, при которых возникает деградация. Если их проигнорировать, то многие системы будут работать плохо. По мере увеличения нагрузки задачи из-за нехватки памяти будут регулярно «падать» или тратить все ресурсы процессора на обработку переполнения памяти, время отклика возрастет, трафик будет теряться, а задачи — конкурировать за доступ к ресурсам. Оставленный без внимания отдельный сбой (например, в одной из задач бэкенда) может спровоцировать сбои в других компонентах системы, и это потенциально может привести к краху всей системы или значительной ее части. Последствия таких каскадных сбоев могут быть очень серьезными, поэтому любая крупномасштабная система должна иметь средства защиты от них (см. главу 22).

Зачастую ошибочно предполагают, что перегруженный бэкенд должен перестать принимать трафик и отключиться. Но это идет вразрез с концепцией надежной балансировки нагрузки. Мы хотим, чтобы бэкенд принимал максимально возможный объем трафика. Хорошо функционирующий бэкенд, поддерживаемый продуманной политикой балансировки нагрузки, должен принимать только те запросы, которые может обработать, и корректно отклонять остальные.

Несмотря на большой выбор инструментов для реализации хорошей системы балансировки нагрузки и защиты от перегрузок, панацеи не существует: для балансировки нагрузки зачастую требуется глубокое понимание системы и семантики ее запросов. Приемы, описанные в этой главе, развивались вместе с потребностями многих систем компаний Google и, скорее всего, продолжат развиваться, поскольку характер систем постоянно меняется.

[135](#) Например, взгляните на Doorman — корпоративную распределенную систему регулирования на стороне клиента.

22. Справляемся с каскадными сбоями

Автор — Майк Ульрих

*Если вы не преуспели сразу,
отступайте экспоненциально.*

Дэн Сэндлер, инженер-программист компании Google

*Почему люди всегда забывают о
том, что нужно учитывать
небольшую погрешность?*

Аде Ошинай, Developer Advocate¹³⁶ Google

Каскадный сбой — это сбой, который распространяется с течением времени в результате положительной обратной связи¹³⁷. Он может произойти, когда часть системы дает сбой, повышая вероятность того, что дадут сбой и другие ее части. Например, одна реплика для сервиса может дать сбой из-за перегрузки, повышая нагрузку на оставшиеся реплики и увеличивая вероятность того, что и они дадут сбой, вызвав эффект домино, который приведет к сбою всех реплик. В качестве примера в этой главе мы будем использовать поисковый сервис, который рассмотрели в конце главы 2. Его конфигурация может выглядеть так, как показано на рис. 22.1.

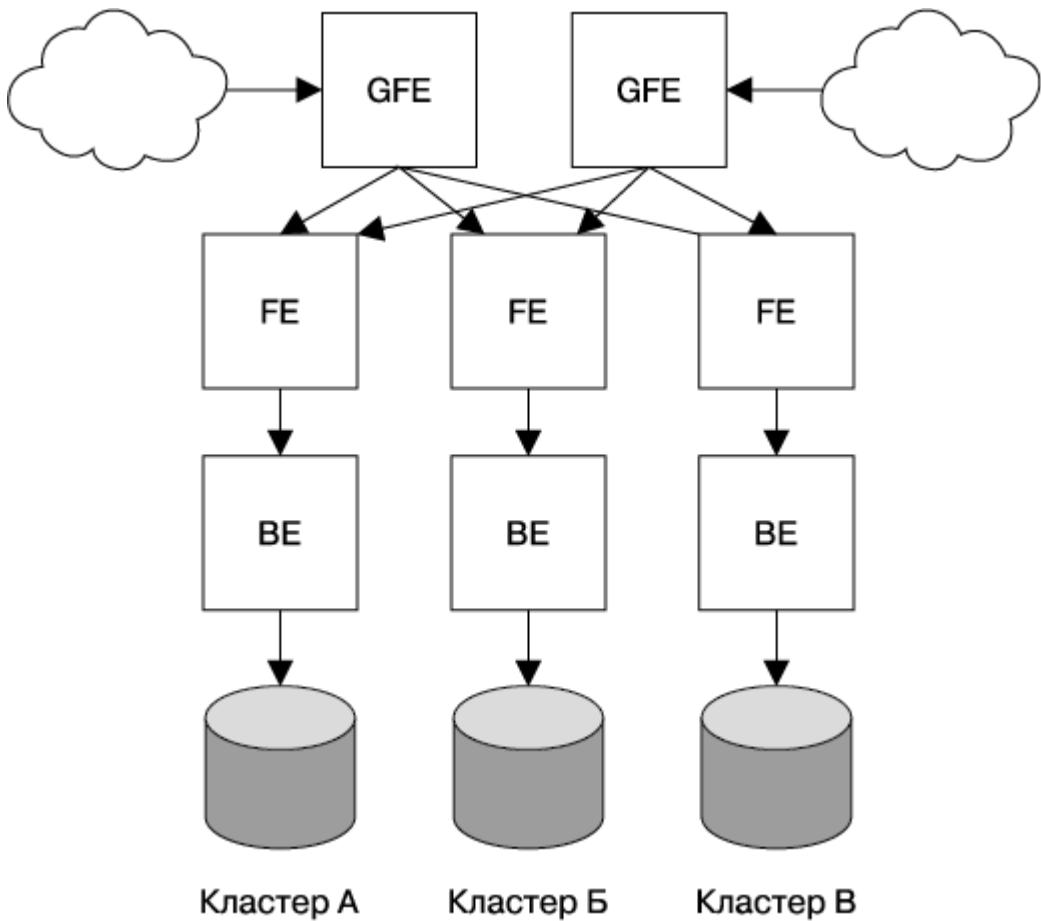


Рис. 22.1. Пример производственной конфигурации поискового сервиса

Причины каскадных сбоев и способы их избежать

Хорошо продуманный дизайн системы должен принимать во внимание некоторые типичные сценарии, которые учитывают большую часть каскадных сбоев

Перегруженность сервера

Наиболее часто причиной каскадных сбоев становится перегруженность. Бо́льшая часть каскадных сбоев, описанная здесь, произошла либо непосредственно из-за

перегруженности сервера, либо из-за последствий или вариаций такого сценария.

Предположим, что фронтенд в кластере А обрабатывает 1000 запросов в секунду (requests per second, QPS), как показано на рис. 22.2.

Если кластер Б дает сбой (рис. 22.3), уровень запросов к кластеру А повышается до 1200 QPS. Фронтенды кластера А не могут обработать 1200 QPS, поэтому у них заканчиваются ресурсы, что приводит к сбоям, пропущенным дедлайнам и другим вариантам неправильного поведения. В результате количество успешно обработанных запросов для кластера А падает ниже 1000 QPS.

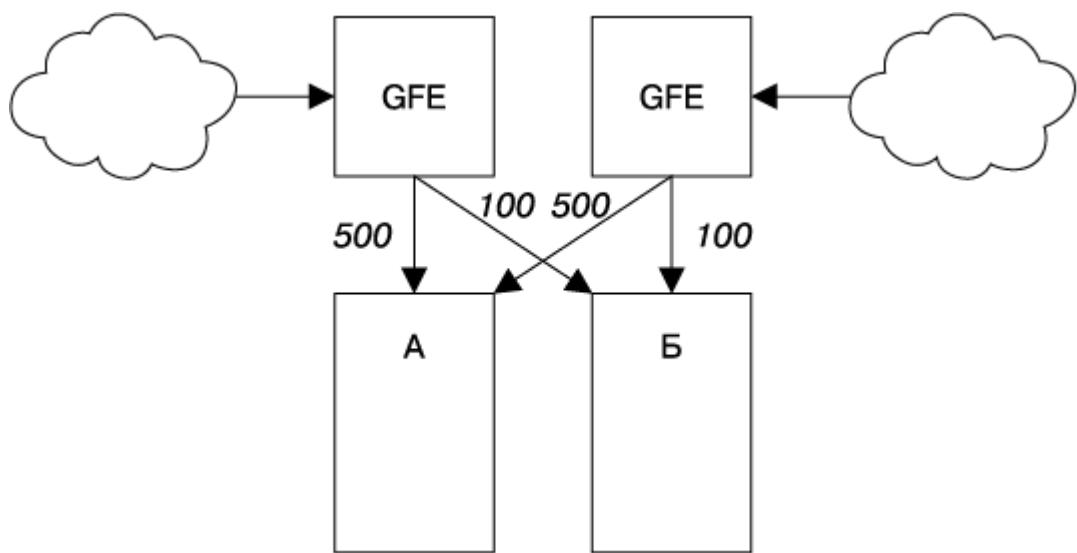


Рис. 22.2. Нормальное распределение загрузки между кластерами А и Б

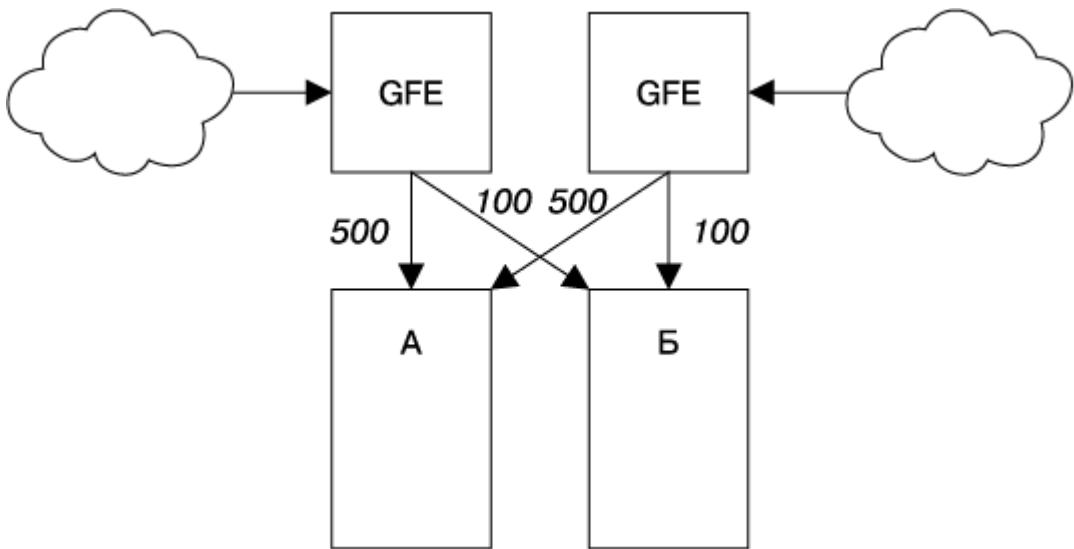


Рис. 22.3. Кластер Б дает сбой, отправляя весь трафик на кластер А

Такое снижение уровня успешно выполняемой работы может распространиться на другие области отказов, возможно, даже глобально. Например, локальная перегруженность одного кластера может привести к падению его серверов; из-за этого контроллер балансировки нагрузки отправляет запрос на другие кластеры, перегружая и их серверы, что вызывает перегрузку на уровне всего сервиса. Эти события могут быть обнаружены довольно быстро, иногда за несколько минут, поскольку системы балансировки нагрузки и планирования задач могут действовать с большой скоростью.

Истощение ресурсов

Истощение ресурсов может привести к увеличенной задержке, повышенному количеству ошибок или предоставлению недостоверных результатов. Фактически перечислены желательные эффекты нехватки ресурсов: за превышение допустимой загрузки сервера вы должны чем-то расплачиваться.

В зависимости от того, какие ресурсы истощаются на сервере, а также от его конфигурации нехватка ресурсов может сделать сервер менее эффективным или заставить его «упасть», в результате чего балансировщик нагрузки распространит проблемы с ресурсами на другие серверы. Когда это происходит, уровень успешно обработанных запросов может снизиться и, возможно, отправить кластер или весь сервис в состояние каскадного сбоя.

Истощаться могут ресурсы разных типов, это оказывает на серверы разное воздействие.

Процессор

Если для обработки загрузки запроса не хватает ресурсов процессора, то, как правило, все запросы выполняются медленнее. Этот сценарий может привести к проявлению разных вторичных эффектов, включая следующие.

- *Увеличение количества активных запросов.* Поскольку запросы обрабатываются дольше, одновременно начинает обрабатываться большее их количество — вплоть до максимально возможного, по достижении которого они могут начать попадать в очереди. Это влияет практически на все ресурсы, включая память, количество активных потоков (если используется модель «поток на запрос»), количество дескрипторов файлов и ресурсы бэкенда (что, в свою очередь, может вызвать другой эффект).
- *Чрезмерно длинные очереди.* Если для обработки всех запросов с постоянной скоростью недостаточно производительности, сервер начинает заполнять свои очереди. Это означает увеличение задержки использования памяти пакетами запросов (запросы находятся в очереди дольше). Стратегии

выхода из таких ситуаций рассматриваются в подразделе «Управление очередями» далее.

- *Зависание потоков.* Если поток не может выполняться, поскольку ожидает заблокированный ресурс, то проверка состояния может дать сбой, если невозможно вовремя обслужить конечную точку проверки состояния.
- *Перегруженность процессора или зависание запроса.* Внутренние сторожевые таймеры¹³⁸ на сервере могут обнаружить, что сервер не выполняет обслуживание вследствие сбоев из-за перегруженности процессора, или, если сторожевой таймер запускается удаленно, зависают его запросы в общем потоке запросов.
- *Пропущенные дедлайны RPC.* Когда сервер оказывается перегруженным, его ответы на RPC клиентов будут приходить позже, и из-за этого могут быть упущены дедлайны, установленные для этих клиентов. Работа, которую сервер выполнил для того, чтобы генерировать ответ, проделана впустую, и клиенты могут попробовать снова вызвать RPC, что приведет к еще большей перегрузке.
- *Уменьшенные преимущества кэширования процессора.* Чем больше процессоров действует, тем выше вероятность того, что задача будет выполнена на большем количестве ядер, а это уменьшает степень использования локального кэша и снижает эффективность процессора.

Память

Большее количество запросов потребляет большее количество памяти, которая требуется для выделения объектам запроса,

ответа и RPC. Истощение памяти может привести к следующим результатам.

- *Умирающие задачи.* Например, задача может быть исключена менеджером контейнера (VM или чем-то другим) за превышение доступных ресурсов или характерных для приложения сбоев, которые могут вызвать остановку задач.
- *Повышенная частота сборки мусора (garbage collection, GC) в Java, что приводит к усиленному использованию процессора.* В этом сценарии может получиться замкнутый круг: будет доступно меньше процессоров, что приведет к медленному выполнению запросов, что вызовет повышение степени использования памяти, что приведет к более частой сборке мусора и, как результат, к еще меньшей доступности процессоров. Эта ситуация известна как «спираль смерти GC».
- *Снижение частоты попаданий кэша.* Уменьшение доступного объема памяти может понизить частоту попадания кэша приложения, что приведет к большему количеству вызовов бэкендов со стороны RPC, из-за чего бэкенды могут оказаться перегруженными.

Потоки

Зависание потоков может вызывать ошибки или приводить к сбоям проверки состояния. Если сервер по мере надобности добавляет потоки, они могут начать использовать слишком большое количество памяти. В наиболее вопиющих случаях зависание потоков может привести к тому, что у вас закончатся идентификаторы процессов.

Дескрипторы файлов

Израсходование свободных дескрипторов файлов может привести к неспособности инициализировать сетевые соединения, что, в свою очередь, заставит проверки состояния генерировать сбои.

Зависимости между ресурсами

Обратите внимание на то, что многие сценарии израсходования ресурсов происходят один из другого — перегруженный сервис зачастую имеет вторичные симптомы, которые могут выглядеть как основная причина, что затрудняет отладку.

Например, представьте следующий сценарий.

1. Фронтенд, написанный на Java, имеет плохо настроенные параметры сборки мусора (GC).
2. Под высокой, но ожидаемой нагрузкой у фронтенда заканчивается память из-за GC.
3. Перегруженность процессора замедляет выполнение запросов.
4. Увеличение количества выполняющихся запросов приводит к тому, что для их обработки используется больший объем памяти.
5. Недостаток памяти из-за запросов вкупе с фиксированным выделением памяти для всего процесса фронтенда приводит к тому, что для кэширования остается меньше памяти.
6. Уменьшенный размер кэша означает, что в нем будет храниться меньше записей, а также снизится частота

попаданий.

7. Увеличение промахов кэша приведет к тому, что большее количество запросов будет попадать на бэкенд для обслуживания.
8. У бэкенда, в свою очередь, закончатся ресурсы процессора или потоков.
9. Наконец, перегруженность процессора приведет к сбоям при проверке состояния, запуская каскадный сбой.

В сложных ситуациях наподобие предыдущего сценария маловероятно, что причинно-следственная цепочка будет полностью проdiagностирована во время отключения. Бывает очень трудно определить, что сбой бэкенда вызван снижением частоты попаданий кэша на фронтенде, особенно если у фронтенда и бэкенда разные владельцы.

Недоступность сервисов

Израсходование ресурсов может привести к падению серверов. Например, серверы могут начать сбоить, если контейнеру выделено слишком много памяти. Как только несколько серверов «упадут» из-за перегруженности, нагрузка на оставшиеся может возрасти, что вызовет и их сбои. Обычно проблема растет как снежный ком, и вскоре все серверы войдут в состояние сбоя. Зачастую избежать такого сценария сложно, поскольку, как только серверы вернутся в строй, они будут бомбардированы чрезвычайно большим количеством запросов и практически моментально дадут сбой.

Например, если при 10 000 QPS сервис был работоспособен, но запустил каскадный сбой из-за того, что «упал» при 11 000

QPS, снижение нагрузки до 9000 QPS, скорее всего, не остановит процесс. Это происходит потому, что сервис будет обрабатывать повышенный спрос с пониженной производительностью, и только малая доля серверов будет достаточно работоспособна для того, чтобы обрабатывать запросы. Доля серверов, которые останутся работоспособными, определяется на основе нескольких факторов: как быстро система может начать запускать задачи, как быстро приложение может начать работать на полную мощность и как долго только что созданная задача сможет выдерживать нагрузку. В нашем примере, если 10 % серверов достаточно работоспособны, чтобы обрабатывать запросы, интенсивность запросов должна будет снизиться примерно до 1000 QPS для того, чтобы система смогла стабилизироваться и восстановиться.

Аналогично серверы могут представляться неработоспособными для уровня балансировки нагрузки, что приведет к снижению производительности балансировщика нагрузки: серверы могут войти в состояние «хромой утки» (см. подраздел «Усовершенствованный подход к контролю работоспособности задач: состояние “хромой утки”» на с. 293) или давать сбои при проверке состояния, не «падая» при этом. Эффект может оказаться аналогичным «падению»: большее количество серверов представляются неработоспособными, работоспособные серверы принимают запросы лишь очень короткий промежуток времени перед тем, как выйти из строя, и еще меньшее количество серверов участвует в обработке запросов. Политики балансировки нагрузки, которые исключают серверы, возвращающие ошибки, могут еще больше усугубить проблему — несколько бэкендов вернули ошибки и поэтому больше не вносят вклад в повышение производительность сервиса. Это повышает нагрузку на оставшиеся серверы, запуская эффект снежного кома.

Предотвращаем перегруженность сервера

В следующем списке представлены стратегии по предотвращению перегруженности серверов в порядке приоритетности.

- *Запустите нагрузочный тест для анализа производительности сервера и проверьте режим восстановления при перегруженности.* Это самое главное, что вы должны сделать, чтобы предотвратить перегрузку сервера. Если вы не выполняете тесты в реалистичной среде, очень трудно предсказать, какой ресурс будет исчерпан и как это повлияет на систему. Для получения более подробной информации прочтите раздел «Тестирование на предмет каскадных сбоев» далее в этой главе.
- *Отправляйте деградированные результаты.* Отправляйте пользователю низкокачественные, но более дешевые для вычисления результаты. Ваша стратегия будет зависеть от конкретного сервиса. Для получения более подробной информации прочтите подраздел «Сегментация нагрузки и мягкая деградация» далее.
- *Укажите серверу отклонять запросы при перегруженности.* Серверы должны защищать себя от перегрузки и «падений». При перегруженности на уровне фронтенда или бэкенда генерируйте сбой рано и дешево. Более подробную информацию об этом вы найдете в подразделе «Сегментация нагрузки и мягкая деградация» далее.
- *Укажите высокоуровневым системам отклонять запросы, а не перегружать сервера.* Обратите внимание на то, что, поскольку ограничение уровня запросов зачастую не учитывает состояние всего сервиса, это не сможет

остановить уже начавшийся сбой. Простые реализации ограничения уровня запросов приведут также к тому, что некоторые ресурсы не будут использоваться. Ограничить уровень запросов можно в нескольких местах:

- *на уровне обратных прокси* путем ограничения объема запросов по таким критериям, как IP-адрес, для того чтобы справиться с попытками DDoS-атак и злоупотребляющими клиентами;
- *на уровне балансировщиков нагрузки* путем отклонения поступающих запросов, когда сервис вошел в состояние глобальной перегруженности. В зависимости от характера и сложности сервиса такое ограничение уровня запросов может быть как общим («отклонять весь трафик, превышающий x запросов в секунду»), так и избирательным («отклонять запросы, которые пришли не от пользователей, взаимодействовавших с сервисом недавно» или «отклонять запросы от операций с низким приоритетом наподобие фоновой синхронизации, но продолжать обслуживать интерактивные сессии пользователя»);
- *на уровне отдельных задач* для того, чтобы предотвратить ситуацию, когда случайные флуктуации при балансировке нагрузки вызовут перегруженность сервера.
- *Планирование производительности.* Хорошее планирование производительности может снизить вероятность каскадного сбоя. Его нужно объединить с тестированием производительности для того, чтобы определить нагрузку, при которой сервер начнет давать сбой. Например, если

каждый кластер дает сбой при 5000 QPS, нагрузка равномерно распределена между кластерами^{[139](#)}, а пиковая нагрузка сервиса достигает 19 000 QPS, то для того, чтобы сервис работал на уровне $N + 2$, вам понадобится примерно шесть кластеров.

Планирование производительности снижает вероятность того, что произойдет каскадный сбой, но его может быть недостаточно. Если вы теряете значительные части своей инфраструктуры из-за запланированного или незапланированного события, никакого планирования производительности не будет достаточно для того, чтобы предотвратить каскадные сбои. Проблемы с балансировкой нагрузки, нарушение связности сети или неожиданное увеличение трафика могут вывести пики высокой нагрузки за пределы спланированных значений. Некоторые системы способны наращивать количество обслуживающих задач для сервиса по запросу, что может предотвратить перегруженность, однако планировать производительность все же необходимо.

Управление очередями

Бо́льшая часть серверов, работающих по принципу «поток на запрос», использует для обработки запросов очередь, расположенную перед пулом потоков. Входящие запросы попадают в очередь, и затем поток выбирает их из нее и выполняет реальную работу (любые действия, которые должен реализовать сервер). Обычно, если очередь заполнена, сервер будет отклонять новые запросы.

Если уровень запросов и задержка обработки заданной задачи являются константными, нет необходимости помещать

запросы в очередь: константное количество потоков должно быть занятым. Согласно этому идеализированному сценарию запросы могут быть помещены в очередь только в том случае, если количество входящих запросов превышает некоторое граничное значение и сервер не может обработать их все, что приводит к насыщению пула потоков и возникновению очереди.

Запросы, помещенные в очередь, потребляют память и увеличивают задержку обработки данных. Например, если размер очереди равен количеству потоков, умноженному на 10, время обработки запроса в потоке составляет 100 миллисекунд. Если очередь заполнена, то на обработку запроса потребуется 1,1 секунды, при этом бо́льшую часть времени он проведет в очереди.

Для системы, чей поток трафика с течением времени остается относительно равномерным, обычно следует иметь очереди, длина которых незначительно превышает размер пула потоков, например, на 50 % или меньше, в результате чего сервер заранее будет отклонять запросы, которые не сможет обработать. Например, сервис Gmail зачастую использует серверы без очередей, полагаясь на процесс восстановления после сбоев других серверных задач, когда потоки заполнены. В то же время системы с периодически возникающей пиковой нагрузкой, при которой шаблоны поступления трафика могут значительно изменяться, работают лучше, когда размер очереди основан на текущем количестве потоков, времени обработки каждого запроса, а также размере и частоте пиков запросов.

Сегментация нагрузки и мягкая деградация

Сегментация позволяет убрать некоторую часть нагрузки, отклоняя трафик, когда сервер начинает перегружаться. Цель

заключается в том, чтобы предотвратить ситуацию, когда у сервера закончится свободная оперативная память и он начнет проваливать проверки состояния, обслуживать запросы с высокой задержкой или показывать какие-нибудь другие симптомы, связанные с перегруженностью, при этом выполняя максимально возможное количество полезной работы. Одним из наиболее простейших способов сегментации нагрузки является позадачное подавление, основанное на показателях использования процессора, памяти или длины очереди. Ограничение длины очереди, о котором говорилось в подразделе «Управление очередями» ранее в данной главе, является одной из форм такой стратегии. Например, одним из эффективных подходов является возвращение кода HTTP 503 («сервис недоступен») любому входящему запросу, когда количество поступивших запросов превышает заданное.

Изменение метода формирования очереди со стандартного «первым вошел, первым вышел» (*first-in, first-out, FIFO*) на «последним вошел, первым вышел» (*last-in, first-out, LIFO*) или применение алгоритма контролируемой задержки (*controlled delay, CoDel*) [Nichols, 2012] или похожих подходов может снизить нагрузку путем избавления от запросов, которые, скорее всего, не стоит обрабатывать [Maurer, 2015]. Если пользовательский поисковый запрос в Интернете выполняется медленно из-за того, что RPC находилась в очереди 10 секунд, весьма вероятно, что пользователь устал ждать и обновил браузер, отправив новый запрос: нет смысла отвечать на первый запрос, поскольку ответ будет проигнорирован! Эта стратегия работает хорошо, если ее объединить с распространением дедлайнов RPC по всему стеку, как описано в подразделе «Задержка и дедлайны» далее.

Более сложные подходы заключаются в определении клиентов, более требовательных к качеству работы или

определению самых важных или приоритетных запросов. Такие стратегии, скорее всего, подойдут для создания разделяемых сервисов.

Мягкая деградация использует концепцию сегментации нагрузки и выводит ее на новый уровень, снижая объем выполняемой работы. В некоторых приложениях можно значительно уменьшить объем работы или время, требующееся для выполнения запроса, ухудшив качество ответа. Например, поисковое приложение может выполнять поиск лишь в небольшом подмножестве данных, хранящихся в расположенных в памяти кэшах, вместо того чтобы искать во всей базе данных. Или же задействовать менее точный, но более быстрый алгоритм ранжирования при перегруженности.

При оценке вариантов сегментации нагрузки или мягкой деградации для сервиса вам нужно принять во внимание следующее.

- Какие показатели вы можете применить для определения момента, в который можно начинать сегментацию нагрузки или мягкую деградацию (например, загруженность процессора, задержку, длину очереди, количество использованных потоков, а также решение о том, должен ли сервис входить в деградированный режим автоматически, или ему необходимо ручное вмешательство)?
- Какие действия нужно предпринять, если сервер вошел в деградированный режим?
- На каком слое нужно реализовывать сегментацию нагрузки и мягкую деградацию? Имеет ли смысл реализовывать эти стратегии на каждом слое стека или достаточно сделать это на одном проблемном участке?

При оценке вариантов и внедрении помните о следующем.

- Мягкая деградация не должна запускаться очень часто — только при снижении запланированной производительности или неожиданном изменении нагрузки. Пусть ваша система будет простой и понятной, особенно если ее станут использовать не так уж часто.
- Помните, что часть кода, которую вы никогда не применяете, зачастую является неработающей. При обслуживании запросов в обычном темпе режим мягкой деградации задействоваться не будет, что подразумевает, что вы получите гораздо меньше операционного опыта в этом режиме и вряд ли будете понимать его особенности, что *повышает* уровень риска. Вы можете убедиться, что мягкая деградация продолжает работать, регулярно допуская ситуацию, когда небольшое подмножество серверов работает почти на пределе, просто чтобы проверить этот сценарий.
- Отслеживайте ситуацию, когда слишком большое количество серверов входит в эти режимы, и оповещайте о ней.
- Сложная сегментация нагрузки и мягкая деградация могут вызвать проблемы сами по себе — избыточная сложность может заставить сервер войти в деградированный режим или в циклы обратной связи, когда это нежелательно. Разработайте способ быстро отключать сложную мягкую деградацию или настраивать ее параметры по необходимости.

Сохранение этой конфигурации в устойчивой системе, в которой каждый сервер наподобие Chubby следит за

изменениями, может повысить скорость развертывания, но в то же время вы будете не защищены от рисков сбоя синхронизации.

Повторные попытки

Предположим, что код фронтенда, который общается с бэкендом, просто выполняет повторные попытки.

Он делает это после того, как произойдет сбой, и ограничивает количество RPC бэкенда для одного запроса десятью. Рассмотрим код фронтенда, в котором используются gRPC и Go:

```
func exampleRpcCall(client pb.ExampleClient,
request pb.Request) *pb.Response {

    // Set RPC timeout to 5 seconds.
    opts := grpc.WithTimeout(5 * time.Second)

    // Try up to 20 times to make the RPC call.
    attempts := 20
    for attempts > 0 {
        conn, err := grpc.Dial(*serverAddr,
opts...)
        if err != nil {
            // Something went wrong in setting
            up the connection. Try again.
            attempts--
            continue
        }
        defer conn.Close()
    }
}
```

```

        // Create a client stub and make the
        // RPC call.
        client := pb.NewBackendClient(conn)
        response, err :=
client.MakeRequest(context.Background, request)
        if err != nil {
            // Something went wrong in making
            // the call. Try again.
            attempts--
            continue
        }

        return response
    }

    grpclog.Fatalf("ran out of attempts")
}

```

Эта система может функционировать следующим образом.

1. Предположим, что наш бэкенд имеет известную границу 10 000 QPS на задачу, после чего все дальнейшие запросы отклоняются и для них выполняется мягкая деградация.
2. Фронтенд вызывает MakeRequest с заданным уровнем 10 100 QPS и перегружает бэкенд на 100 QPS, которые тот отклоняет.
3. Для этих 100 QPS, вызвавших сбой, MakeRequest выполняет повторные попытки каждые 1000 миллисекунд, и они, возможно, успешно выполняются. Но повторные попытки сами по себе увеличивают количество запросов,

отправляемых на бэкенд, — теперь он получает 10 200 QPS, 200 из которых вызывают сбой из-за перегрузки.

4. Количество повторных попыток растет: 100 QPS повторных попыток в первую секунду приводят к 200 QPS повторных попыток, затем к 300 QPS и т.д. Все меньшее количество запросов успешно реализуется с первой попытки, поэтому полезную работу выполняет все меньшее количество запросов.
5. Если задача бэкенда не может обработать повышение загрузки, которое потребляет дескрипторы файлов, память и процессорное время бэкенда, она может дать сбой под нагрузкой, создаваемой запросами и повторными попытками. Из-за этого запросы, получаемые бэкендом, перераспределяются между оставшимися задачами, что, в свою очередь, приведет и к их перегрузке.

Чтобы проиллюстрировать этот сценарий, были сделаны некоторые упрощающие предположения¹⁴⁰, но общий смысл заключается в том, что повторные попытки могут дестабилизировать систему. Обратите внимание на то, что как временные всплески нагрузки, так и медленное ее повышение могут вызвать такой эффект.

Даже если уровень вызовов MakeRequest окажется ниже катастрофического (например, 9000 QPS), вы можете не избавиться от проблемы. Это будет зависеть от того, во что бэкенду обошлось возвращение ошибки. Здесь играют роль два фактора.

- Если бэкенд тратит значительное количество ресурсов на обработку запросов, которые в итоге завершатся с ошибкой

из-за перегруженности, то повторные попытки сами по себе могут поддерживать бэкенд в режиме перегруженности.

- Серверы бэкенда сами могут быть нестабильны. Повторные попытки могут усилить эффекты, о которых говорилось в разделе «Перегруженность сервера» ранее.

Если выполняется хотя бы одно из этих условий, то для того, чтобы справиться со сбоем, вы должны значительно снизить нагрузку (или вовсе избавиться от нее) на фронтендах до тех пор, пока повторные попытки не прекратятся и бэкенд не стабилизируется.

Такая схема стала причиной нескольких каскадных сбоев, в которых либо фронтенд, либо бэкенд общались посредством сообщений RPC, где фронтенд — код клиента JavaScript — отправлял вызовы XMLHttpRequest к конечной точке и повторял попытки при сбоях.

При автоматическом выполнении повторных попыток имейте в виду следующее.

- В этом случае применима бо́льшая часть стратегий защиты бэкенда, описанных ранее в разделе «Предотвращаем перегруженность сервера». В частности, тестирование системы может выявить проблемы, а мягкая деградация — ослабить воздействие, который повторные попытки оказывают на бэкенд.
- При планировании повторных попыток всегда используйте рандомизированный экспоненциальный откат (см. статью Exponential Backoff and Jitter в блоге AWS Architecture Blog [Brooker, 2015]). Если повторные попытки не распределены случайным образом в рамках окна повторов, небольшие проблемы, например связанные с сетью, могут вызвать

ураган повторных попыток, запланированных на одно и то же время, что может увеличить их количество [Floyd, 1994].

- Ограничите количество повторных попыток для запроса. Не пытайтесь выполнить запрос бесконечное количество раз.
- Рассмотрите возможность введения лимита повторных попыток для сервера. Например, позволяйте процессу выполнять 60 повторных попыток в минуту и, если это количество будет превышено, не продолжайте — просто возвращайте ошибку. Эта стратегия может продемонстрировать разницу между ошибкой при планировании производительности, которая вызывает отклонение некоторых запросов, и глобальным каскадным сбоем.
- Подумайте о сервере как о едином объекте и решите, хотите ли вы выполнять повторные попытки на заданном уровне. В частности, избегайте увеличения числа повторных попыток, выполняя их на всех уровнях: один запрос на верхнем уровне может создать количество повторных попыток, равное произведению количества попыток на каждом уровне вплоть до нижнего. Если база данных не может обработать запросы сервиса, поскольку он перегружен, и уровни бэкенда, фронтенда и JavaScript выполняют по три повторные попытки (четыре вызова), одно действие пользователя может создать 64 обращения (4^3) к базе данных. Нежелательно такое поведение, при котором база данных возвращает эти ошибки из-за перегруженности.
- Используйте понятные коды ответов и обдумывайте, как нужно обрабатывать разные ошибки. Например, разделите

ошибочные состояния, при которых можно повторять запросы, и состояния, при которых этого делать нельзя. Не пытайтесь выполнять повторные попытки для постоянных ошибок или плохо сформированных запросов в клиенте, поскольку это не принесет пользы. Верните определенный статус при перегруженности, чтобы клиент и другие уровни не пытались выполнить повторные попытки самостоятельно.

В экстренных случаях может казаться неочевидным, что сбой вызван некорректным поведением при выполнении повторных попыток. Графики уровня повторных попыток могут служить индикатором некорректного поведения при выполнении повторных попыток, но это можно принять за симптом, а не за причину. Что касается смягчения последствий — это особый случай проблемы недостаточной производительности, имеющий дополнительный подводный камень: вам нужно либо исправить поведение при повторных попытках (это обычно требует изменения кода), либо значительно снизить нагрузку, либо полностью избавиться от запросов.

Задержка и дедлайны

Когда фронтенд отправляет RPC серверу бэкенда, фронтенд потребляет ресурсы, ожидая ответа. Дедлайны RPC определяют, как долго можно ждать запрос до того, как фронтенд вернет управление, ограничивая время, в которое бэкенд может потреблять ресурсы фронтенда.

Выбор дедлайна

Как правило, надо установить разумный дедлайн. Установка очень длинного дедлайна или его игнорирование может заставить краткосрочные задачи, которые давно решены, продолжать потреблять ресурсы сервера до его перезапуска.

Установка длинных дедлайнов может привести к тому, что на высоких уровнях стека будет потребляться больше ресурсов, когда на низких уровнях возникают какие-то проблемы. Короткие дедлайны могут вызвать сбои у некоторых особо дорогих запросов. Выбор сбалансированного дедлайна — это в чем-то искусство.

Пропущенные дедлайны

Распространенной причиной многих каскадных сбоев является ситуация, когда серверы тратят ресурсы на обработку запросов, которые превышают свои дедлайны на клиенте. В результате ресурсы тратятся впустую, а вы не получаете следующих заданий RPC.

Предположим, RPC имеет десятисекундный дедлайн, установленный клиентом. Сервер очень перегружен, и, чтобы перейти из очереди в пул потоков, требуется 11 секунд. В этот момент клиент уже отказался от запроса. В большинстве случаев для сервера будет неразумно пытаться выполнить этот запрос, поскольку клиенту после истечения дедлайна уже неважно, будет ли запрос выполнен, так как клиент уже отказался от него.

Если обработка запроса выполняется в несколько этапов (например, имеется несколько обратных вызовов или вызовов RPC), сервер должен проверять оставшееся на выполнение запроса время на каждом этапе перед тем, как делать еще какую-то работу по обработке запроса. Например, если запрос разделен на этапы анализа, запроса бэкенда и обработки,

имеет смысл проверять, достаточно ли осталось времени для обработки запроса, перед началом каждого этапа.

Распространение дедлайнов

Вместо того чтобы просто указывать дедлайн при отправке RPC бэкендам, серверы должны использовать распространение дедлайнов и сигналов об отмене.

При распространении дедлайн устанавливается в верхней части стека, например на фронтенде. Дерево RPC исходя из первоначального запроса будет иметь тот же самый абсолютный дедлайн. Например, если сервер А выберет в качестве значения для дедлайна 30 секунд, обработает запрос за 7 секунд и отправит RPC серверу Б, эта RPC будет иметь значение дедлайна, равное 23 секундам. Если серверу Б на обработку запроса потребуется 4 секунды, то для RPC, которая будет отправлена на сервер В, значение дедлайна составит 19 секунд, и т.д. В идеале каждый сервер в дереве запросов реализует распространение дедлайна. Без применения этого приема может возникнуть следующая ситуация.

1. Сервер А отправляет RPC серверу Б с десятисекундным дедлайном.
2. Сервер Б тратит 8 секунд на обработку запроса и отправляет RPC на сервер В.
3. Если сервер Б использует распространение дедлайна, он должен задать его значение, равное 2 секундам, но предположим, что он применяет жестко закодированное значение, равное 20 секундам.
4. Сервер В выберет запрос из очереди спустя 5 секунд.

Если бы сервер Б использовал распространение дедлайна, сервер В мгновенно отказался бы от запроса, поскольку дедлайн на его выполнение был бы превышен. Однако в нашем сценарии сервер В обрабатывает запрос, думая, что у него есть еще 15 секунд, но он не выполняет полезную работу, поскольку запрос от сервера А к серверу Б уже превысил дедлайн.

Вы можете захотеть немного снизить исходящий дедлайн (например, на несколько сотен миллисекунд), чтобы учесть время передачи по сети и последующую обработку на клиенте. Также рассмотрите возможность установки верхней границы для исходящих дедлайнов. Вы можете захотеть ограничить время ожидания сервером исходящих RPC к некритичным бэкендам. Однако убедитесь, что понимаете структуру своего трафика, поскольку в противном случае вы можете непреднамеренно заставить каждый раз давать сбой некоторые типы запросов, например запросы с большим объемом полезной нагрузки или те, которые для ответа требуют большого количества вычислений.

Существуют исключения, когда серверы могут продолжить обрабатывать запрос, когда дедлайн уже истек. Например, если сервер получает запрос, который включает в себя выполнение затратных операций для достижения некоего состояния и периодическую фиксацию контрольных точек процесса, удачным приемом может быть проверка дедлайна только после контрольной точки вместо проверки после выполнения затратной операции.

Распространение сигналов об отмене позволяет избежать потенциальной утечки RPC, которая происходит в том случае, когда исходный RPC имеет длинный дедлайн, но RPC, находящиеся между более глубокими слоями стека, имеют короткий дедлайн и тайм-аут. Используя простое распространение дедлайна, исходный RPC продолжает

задействовать ресурсы сервера до тех пор, пока не завершится по тайм-ауту, несмотря на невозможность продвинуться в выполнении задачи.

Бимодальная задержка

Предположим, что фронтенд из предыдущего примера состоит из десяти серверов, каждый из которых имеет 100 рабочих потоков. Это означает, что фронтенд имеет производительность, равную 1000 потоков. Во время работы в обычном режиме фронтенд выполняет 1000 QPS и запросы завершаются за 100 миллисекунд. Это означает, что у фронтенда заняты 100 рабочих потоков из 1000 сконфигурированных ($1000 \text{ QPS} \cdot 0,1 \text{ с}$). Предположим, что некоторое событие заставляет 5 % запросов остаться незавершенными. Это может быть результатом недоступности некоторых диапазонов строк Bigtable, из-за чего запросы, связанные с этим ключевым пространством, становятся необслуживаемыми. В результате 5 % запросов пропускают дедлайн, а остальные 95 % затрачивают обычные 100 миллисекунд. Имея дедлайн 100 миллисекунд, 5 % запросов потребят 5000 потоков ($50 \text{ QPS} \cdot 100 \text{ с}$), но у фронтенда нет такого количества доступных потоков. Если других вторичных действий не наблюдается, фронтенд сможет обработать только 19,6 % запросов ($1000 \text{ доступных потоков} / (5000 + 95)$ потоков, необходимых для работы), в результате уровень ошибок составит 80,4 %.

Поэтому ошибку станут возвращать не 5 % запросов, которые не могут быть выполнены из-за недоступности ключевого пространства, а их большая часть. Следующие рекомендации могут помочь справиться с проблемами такого рода.

- Определить такую проблему может быть очень трудно. В частности, не сразу можно понять, что причиной сбоев является бимодальная задержка, когда вы смотрите на *среднее значение* задержки. Увидев, что задержка увеличивается, попробуйте проанализировать *распределение* задержек в дополнение к их средним значениям.
- Проблемы можно избежать, если запросы, которые не завершаются, возвращают ошибку досрочно, а не ожидают конца дедлайна. Например, если бэкенд недоступен, то лучше сразу вернуть для него ошибку, чем потреблять ресурсы до конца дедлайна. Если ваш уровень RPC поддерживает вариант fail-fast, используйте его.
- Наличие дедлайнов, на несколько порядков превышающих среднюю задержку обработки запроса, — это обычно плохо. В предыдущем примере пропускают дедлайн небольшое количество запросов, но значение дедлайна на три порядка больше, чем обычная средняя задержка, поэтому пул потоков истощился.
- При задействовании общих ресурсов, которые могут быть истощены некоторыми ключевыми пространствами, рассмотрим возможность либо ограничения запросов в этом ключевом пространстве, либо применения других видов отслеживания некорректного использования. Предположим, что бэкенд обрабатывает запросы для разных клиентов, имеющих разные производительность и характеристики запросов. Вы можете позволить одному клиенту занять лишь 25 % потоков, чтобы обеспечить одинаковое качество обслуживания при возможности высокой нагрузки одним неверно ведущим себя клиентом.

Холодный запуск и холодное кэширование

Процессы сразу после запуска зачастую отвечают на запросы медленнее, чем тогда, когда их состояние стабилизировалось. Это может быть вызвано одной из следующих причин.

- *Необходимость инициализации.* Настройка соединений при получении первого запроса, которому требуется заданный бэкенд.
- *Улучшение производительности во время выполнения программы в некоторых языках, в особенности Java.* Компиляция Just-In-Time, hotspot-оптимизация и отложенная загрузка классов.

Аналогично некоторые приложения менее эффективны, когда кэши не заполнены. Например, в некоторых сервисах Google большая часть запросов обслуживается из кэша, поэтому запросы, для которых не происходит попадания кэша, стоят гораздо больше. При работе в стабильном состоянии, когда кэш разогрет, промахов немного, но когда он совсем пустой, дорогими становятся 100 % запросов. Другие сервисы могут использовать кэш для сохранения состояния пользователя в оперативной памяти. Этого можно добиться с помощью «липкости»[141](#), реализованной аппаратно или программно, между обратными прокси и фронтендами сервиса.

Если сервис не имеет достаточного количества ресурсов для обработки запросов из холодного кэша, риск возникновения сбоев возрастет и вам следует что-то предпринять, чтобы их избежать.

Кэш может оказаться холодным в следующих ситуациях.

- *Включение нового кластера.* У добавленного недавно кластера будет холодный кэш.
- *Возврат кластера сервису после обслуживания.* Кэш может оказаться устаревшим.
- *Перезапуски.* Если задача с кэшем была недавно перезапущена, заполнение ее кэша может занять некоторое время. Возможно, имеет смысл переместить кэширование с сервера в отдельное приложение вроде memcache, что позволит организовать общий кэш для нескольких серверов, пусть и ценой добавления дополнительного RPC и небольшого роста задержки.

Если кэширование значительно влияет на сервис^{[142](#)}, вы можете воспользоваться одной из следующих стратегий.

- *Предоставление сервису избыточного количества ресурсов.* Важно иметь в виду разницу между кэшем задержки и кэшем производительности: когда вы пользуетесь кэшем задержки, сервис может поддерживать ожидаемую нагрузку с помощью пустого кэша, но сервис, который использует кэш производительности, так делать не может. Владельцы сервиса должны внимательно добавлять кэши для него и убеждаться в том, что все новые кэши являются либо кэшами задержки, либо достаточно хорошо спроектированы для того, чтобы безопасно функционировать как кэши производительности. Иногда кэши добавляют в сервис для улучшения производительности, но в итоге они становятся необходимостью.
- *Используйте общие техники предотвращения каскадных сбоев.* В частности, перегруженные серверы должны отклонять

запросы или входить в деградированные режимы. Также нужно выполнять тестирование для того, чтобы увидеть, как сервис будет вести себя после событий наподобие крупного перезапуска.

- *При добавлении к кластеру нагрузки повышайте ее уровень медленно.* Небольшое количество запросов на начальном этапе разогреет кэш. Как только он прогреется, можете добавить трафика. Вам стоит гарантировать, что все кластеры поддерживают номинальную нагрузку, а кэши остаются прогретыми.

Всегда спускайтесь вниз по стеку. Рассмотрим пример: в сервисе Shakespeare фронтенд взаимодействует с бэкендом, который, в свою очередь, взаимодействует с уровнем хранилища. Проблема, проявляющаяся на уровне хранилища, может вызвать проблемы на серверах, которые взаимодействуют с этим уровнем, но исправление ситуации на уровне хранилища в большинстве случаев исправит также уровни фронтенда и бэкенда.

Однако предположим, что бэкенды взаимодействуют друг с другом. Например, они могут отправлять друг другу прокси-запросы для того, чтобы изменить обработчик запросов пользователя, когда уровень хранилища не может обработать запрос. При таком взаимодействии в рамках одного уровня проблемы могут возникнуть по нескольким причинам.

- При взаимодействии может произойти распределенная взаимоблокировка. Бэкенды могут использовать один и тот же пул потоков для ожидания результата RPC, отправленных удаленным бэкендам, которые одновременно получают запросы от удаленных бэкендов. Предположим, что пул потоков бэкенда А исчерпан. Бэкенд Б отправляет запрос

бэкенду А и задействует очередной поток бэкенда Б на время, пока один из потоков из пула бэкенда А не освободится. Такое поведение может вызвать распространение переполнения пула потоков.

- Если интенсивность взаимодействия внутри уровня повышается в ответ на некий сбой или при большой нагрузке (например, балансировщик нагрузки более активен при высокой нагрузке), взаимодействие между уровнями может быстро измениться.

Предположим, что пользователь имеет основной бэкенд и заранее определенный, готовый к работе вторичный бэкенд в другом кластере, который может принять этого пользователя. Основной бэкенд проксирует запросы на вторичный бэкенд из-за возникновения ошибок на более низком уровне или когда нагрузка на него серьезно возрастет. Если вся система перегружена, такое проксирование, скорее всего, еще больше нагрузит ее из-за дополнительной стоимости анализа запроса и ожидания его выполнения на вторичном бэкенде.

- В зависимости от критичности взаимодействия между уровнями автомастерка системы может стать более сложной.

Как правило, стоит избегать взаимодействия внутри одного уровня. Вместо этого предоставьте клиенту заботиться о взаимодействии. Например, если фронтенд общается с бэкендом, но выбирает неправильный бэкенд, последний не должен проксировать на правильный бэкенд. Вместо этого

бэкенд должен предложить фронтенду выполнить запрос повторно уже для правильного бэкенда.

Срабатывание условий каскадного сбоя

Если сервис подвержен каскадным сбоям, то инициировать эффект домино могут несколько возможных нарушений в работе. В этом разделе рассматриваются факторы, приводящие к каскадным сбоям.

«Гибель» процесса

Некоторые задачи сервера могут «погибнуть», снижая уровень доступной производительности. Это может произойти из-за «запроса смерти» (RPC, чье содержимое заставит процесс сгенерировать сбой), проблем с кластером, контроля условий выполнения и по множеству других причин. Элементарное событие, например несколько «падений» или перенос выполнения задач на другие машины, может привести к сбою сервера.

Обновления процессов

Отправка новой версии приложения или обновление его конфигурации может вызвать каскадный сбой, если большое количество задач будет затронуто одновременно. Для того чтобы предотвратить этот сценарий, следует либо учитывать необходимую нагрузку на производительность при настройке инфраструктуры обновления сервиса, либо отправлять новую версию во время непиковой нагрузки. Динамическая корректировка количества работающих обновлений задач, основанная на объеме запросов и доступной производительности, может оказаться удачным подходом.

Новые релизы

Новый бинарный файл, коррективы конфигурации или изменение лежащей в основе стека инфраструктуры могут вызвать изменения в профиле запроса, использовании ресурсов и их ограничениях, бэкендах и других системных компонентах, которые могут спровоцировать каскадный сбой.

Во время каскадного сбоя стоит проверить недавние изменения и рассмотреть возможность откатить их, особенно если они повлияли на производительность или изменили профиль запросов.

Сервис должен журналировать изменения в какой-нибудь форме, что может помочь быстро определить выполненные недавно.

Органический рост

Во многих случаях каскадный сбой бывает вызван не конкретным изменением сервиса, а тем, что активизация использования сервиса не сопровождалась корректировкой производительности.

Запланированные изменения, исчерпание ресурсов или отключения

Если сервис располагается в нескольких кластерах, может оказаться невозможно задействовать часть производительности из-за обслуживания или сбоев кластера. Аналогично одна из критически важных зависимостей сервиса может быть перегружена, что приведет к снижению производительности работающего сервиса или увеличению задержки из-за необходимости отправлять запросы в более удаленный кластер.

Изменение профиля запросов

Сервис бэкенда может получать запросы из разных кластеров, поскольку сервис фронтенда перемещает свой трафик из-за изменения конфигурации балансировщика нагрузки и структуры трафика или переполненности кластера. Кроме того, средняя стоимость обработки индивидуальной нагрузки могла поменяться из-за кода фронтенда или изменения конфигурации. Аналогично данные, обрабатываемые сервисом, могли измениться естественным образом из-за увеличения объема или изменения формата: например, со временем количество и объем изображений для каждого пользователя сервиса хранения фотографий только увеличивается.

Ограничения ресурсов

Некоторые кластерные операционные системы могут предоставлять избыток ресурсов. Процессор — это измеримый ресурс, зачастую машины имеют свободные ресурсы процессора, что в какой-то степени защищает их от всплесков использования. Доступность этих ресурсов различается в разных сегментах, а также на различных машинах одного сегмента.

Зависимость от свободных ресурсов может быть опасной. Их доступность полностью зависит от поведения других задач в кластере, поэтому они могут закончиться в любое время. Например, если команда запускает MapReduce, которая потребляет большую часть ресурсов процессора и планирует задачи на нескольких машинах, общее количество свободных ресурсов процессора может внезапно уменьшиться и процессор окажется перегруженным для других задач. При выполнении

нагрузочных тестов убедитесь, что вы остаетесь в заявленных рамках использования ресурсов.

Тестирование на предмет каскадных сбоев

Определить точные причины сбоя сервиса может быть очень трудно. В этом разделе рассматриваются стратегии тестирования, которые помогут определить вероятность того, что сервис вызовет каскадный сбой.

Вы должны протестировать свой сервис, чтобы определить, как он ведет себя при высоких нагрузках, чтобы быть уверенными в том, что он не войдет в состояние каскадного сбоя при определенных обстоятельствах.

Тестируйте до возникновения сбоев и после

Понять, как поведет себя сервис под высокой нагрузкой, — это, возможно, самый важный первый шаг, помогающий избежать каскадных сбоев. Зная, как система ведет себя при перегрузках, вы сможете определить, какие инженерные задачи более важны в долгосрочной перспективе. По крайней мере, это знание может помочь вам самостоятельно запустить процесс отладки, когда приходится срочно вызывать дежурных инженеров.

Тестируйте компоненты под нагрузкой до тех пор, пока они не дадут сбой. Обычно по мере повышения нагрузки компонент успешно обрабатывает запросы до тех пор, пока не достигает точки, когда уже не может справиться с бо́льшим количеством запросов. В идеале в этот момент в ответ на дополнительную нагрузку он должен начать выдавать ошибки или деградировавшие результаты, но незначительно снизить уровень, при котором успешно обрабатывает запросы. В

момент, когда наступает перегрузка, компонент, сильно подверженный каскадным сбоям, упадет или начнет выдавать большое количество ошибок, тогда как более качественно спроектированный будет отклонять некоторые запросы и выживет.

Нагрузочное тестирование также показывает, где находится точка перелома, что очень важно для планирования производительности. Это позволяет выполнять регрессионное тестирование, чтобы в худших случаях найти компромисс, выбирая между полезностью и безопасностью.

Из-за эффектов кэширования постепенное увеличение нагрузки может дать результаты, отличающиеся от мгновенного повышения нагрузки до ожидаемого уровня. Поэтому рассмотрите возможность тестирования постепенных и импульсных сценариев подачи нагрузки.

Вы также должны протестировать компонент в момент, когда он вернется к номинальной нагрузке после того, как поработал под нагрузкой, значительно ее превышающей, и понять, как он ведет себя в данной ситуации. Такое тестирование может ответить на следующие вопросы.

- Если при высокой нагрузке компонент входит в деградированный режим, может ли он выйти из него без вмешательства человека?
- Если несколько серверов падают под высокой нагрузкой, насколько нужно снизить ее, чтобы стабилизировать систему?

Если вы тестируете сервис с сохранением состояния (*stateful*) или использующий кэширование, нагрузочный тест должен отслеживать состояние при разных взаимодействиях и

проверять правильность при высоких нагрузках, поскольку в этот момент зачастую проявляются труднонаходимые ошибки, связанные с распараллеливанием.

Имейте в виду, что отдельные компоненты могут иметь разные точки перелома, поэтому выполняйте нагружочное тестирование для них по отдельности. Вы не можете заранее знать, какой компонент даст сбой первым, и хотите знать, как ведет себя система, если сбой все-таки произошел. Если вы уверены в том, что система хорошо защищена от перегрузок, рассмотрите выполнение тестов для небольшого фрагмента производственных данных, чтобы определить места, где может произойти сбой в процессе работы с реальным трафиком. Эти ограничения могут быть не отражены в синтезированном трафике нагружочного тестирования, поэтому тесты с реальным трафиком дают более реальные результаты, правда, при этом есть риск повлиять на пользователей. Будьте осторожны при тестировании с реальным трафиком: убедитесь, что имеется резерв производительности на случай, если автоматическая защита не сработает и понадобится исправлять сбой вручную. Вы можете рассмотреть выполнение следующих производственных тестов:

- быстрое или постепенное уменьшение количества задач до значений, выходящих за пределы ожидаемых объемов трафика;
- быстрая потеря кластером производительности;
- эмуляция отказов разных бэкендов.

Тестируйте популярные клиенты

Разберитесь в том, как крупные клиенты используют сервис. Например, вы должны знать, что клиенты:

- могут ставить задачи в очередь, если сервис не работает;
- применяют рандомизированные экспоненциальные откаты при ошибках;
- уязвимы к внешним сигналам, которые могут создать большую загрузку (например, вызванное извне обновление ПО может очистить кэш онлайн-клиента).

В зависимости от сервиса вы можете иметь или не иметь возможности управлять всем кодом клиента, который общается с сервисом. Однако в любом случае не помешает понимать, как поведут себя крупные клиенты, которые взаимодействуют с сервисом.

Эти же принципы применимы и к крупным внутренним клиентам. Симулируйте системные сбои для крупных клиентов и посмотрите, как они будут реагировать. Спросите у внутренних клиентов, как они получают доступ к вашему сервису и какие механизмы используют для обработки сбоев бэкенда.

Тестируйте некритические бэкенды

Тестируйте некритичные бэкенды и убедитесь, что их недоступность не повлияет на критически важные компоненты вашего сервиса.

Предположим, что у фронтенда имеются критически важные и некритичные бэкенды. Зачастую заданный запрос включает в себя как критические компоненты (например, результаты запросов), так и некритические (например,

варианты правописания). Выполнение ваших запросов может значительно замедлиться и потреблять ресурсы, ожидая завершения некритических бэкендов.

В дополнение к тестированию поведения, когда некритические бэкенды недоступны, протестируйте поведение фронтенда, если некритический бэкенд никогда не отвечает (например, если он поглощает запросы). Бэкенды, указанные как некритические, все же могут создавать проблемы для фронтендов, если их запросы имеют длинные дедлайны. Фронтенд не должен начинать отклонять большое количество запросов, истощать свои ресурсы или обслуживать запросы с очень высокой задержкой, если некритический бэкенд начинает поглощать запросы.

Мгновенное реагирование на каскадные сбои

Как только вы поняли, что сервис вошел в состояние каскадного сбоя, можете реализовать несколько разных стратегий для исправления ситуации. И конечно, каскадный сбой — это отличная возможность использовать протокол управления инцидентами (см. главу 14).

Увеличьте количество ресурсов

Если система работает с деградированной производительностью и у вас имеются свободные ресурсы, то добавление задач может стать самым быстрым способом восстановления после сбоя. Однако если сервис вошел в некую «спираль смерти», добавления новых ресурсов может оказаться недостаточно.

Прекратите выполнять проверки на сбои/«гибель»

Некоторые системы планирования задач для кластеров, например Borg, проверяют работоспособность задач и перезапускают неработающие. Этот прием может вызвать режим сбоя, при котором сама проверка состояния может сделать сервис неработоспособным. Например, если половина задач в данный момент стартует, а другая половина скоро будет «убита», так как эти задачи перегружены и проваливают проверки состояния, временное отключение таких проверок может позволить системе стабилизироваться так, что начнут выполняться все задачи. Проверка состояния процесса («Отвечает ли этот бинарный файл?») и сервиса («Может ли этот бинарный файл отвечать на запросы данного вида прямо сейчас?») — это две концептуально разные операции. Проверка состояния процесса важна для планировщика задач кластера, а проверка состояния сервиса — для балансировщика нагрузки. Четкое различие двух типов проверок состояния позволяет избежать рассмотренного сценария.

Перезапускайте серверы

Если серверы зависли и не выполняют работу, может помочь их перезапуск.

Попробуйте перезапускать серверы в следующих ситуациях.

- Серверы Java вошли в «спираль смерти GC».
- Некоторые запросы не имеют дедлайнов, но потребляют ресурсы, что приводит к блокировке потоков.
- Серверы находятся в ситуации взаимной блокировки.

Убедитесь, что нашли источник каскадных сбоев, до того, как перезапустите серверы. Убедитесь, что это действие не

приведет к простому перераспределению нагрузки. Проверьте корректность этого изменения и выполните его медленно. Ваши действия могут лишь усугубить ситуацию, если каскадный сбой вызван, например, холодным кэшем.

Отбрасывайте трафик

Отбрасывание трафика — это способ, зарезервированный для ситуаций, когда у вас случился реальный каскадный сбой и вы никак не можете иначе исправить ситуацию. Например, если высокая нагрузка заставляет серверы «падать» сразу же, как только они стали работоспособными, вы можете вернуть сервис к работе так.

1. Исправьте изначальную причину сбоя, например добавив производительность.
2. Уменьшите нагрузку настолько, чтобы сбои прекратились. Постарайтесь сделать это агрессивно — если сбои дает весь сервис, позвольте ему обрабатывать, например, всего 1 % трафика.
3. Позвольте большинству серверов выздороветь.
4. Постепенно увеличивайте нагрузку.

Эта стратегия позволит прогреть кэши, установить соединения и т.д., прежде чем нагрузка достигнет обычного уровня.

Очевидно, что эта тактика позволяет пользователям заметить нанесенный во время сбоя урон. Можете ли вы (или должны ли) отбрасывать весь трафик, зависит от конфигурации сервиса. Если у вас есть какой-то механизм, который позволяет

отбрасывать не слишком важный трафик (например, механизм предварительной выборки), следует им воспользоваться.

Важно помнить, что эта стратегия позволяет восстановиться от каскадного отключения только тогда, когда устранена проблема, лежащая в его основе. Если она не ликвидирована — например, не хватает глобальной производительности, то каскадный сбой может повторяться вскоре после того, как нагрузка выйдет на нормальный уровень. Поэтому, прежде чем использовать эту стратегию, рассмотрите возможность устранения (или хотя бы компенсации) основной причины или условия сбоя. Например, если у сервиса закончилась память и он вошел в «спираль смерти», добавление дополнительной памяти или новых задач должно стать первым шагом к исправлению ситуации.

Входите в деградированные режимы

Выдавайте деградированные результаты, выполняя меньше работы или отбрасывая неважный трафик. Эта стратегия должна быть встроена в сервис, и ее можно реализовать только в том случае, если вы знаете, как трафик может быть ухудшен, и у вас есть возможность определять разные виды полезной нагрузки.

Избавляйтесь от пакетной нагрузки

Нагрузка на некоторые сервисы важна, но некритична. Рассмотрите возможность отключения этих источников нагрузки. Например, если обновления индекса, копирование данных или генерация статистики потребляют ресурсы, подумайте, нельзя ли отключить эти источники нагрузки во время сбоя.

Избавляйтесь от плохого трафика

Если некоторые запросы создают высокую нагрузку или приводят к сбоям (например, «запросы смерти»), попробуйте блокировать их или избавляться от них другими способами.

КАСКАДНЫЕ СБОИ И СЕРВИС SHAKESPEARE

В Японии выходит документальный фильм о творчестве Шекспира, где упоминается в том числе о сервисе Shakespeare. Из-за этой передачи трафик, идущий к азиатским data-центрам, превышает все возможности сервиса. Эта проблема, связанная с производительностью, еще больше осложнена крупным обновлением сервиса Shakespeare, которое в это же время происходит в data-центре.

К счастью, определенные меры безопасности помогут предотвратить потенциальный сбой. Процесс Production Readiness Review определил проблемы, которые команда уже решила. Например, разработчики внедрили в сервис механизм мягкой деградации. Как только производительность становится недостаточной, сервис больше не возвращает картинки или небольшие карты, иллюстрирующие остальной материал. И в зависимости от предназначения RPC, которая выходит по тайм-ауту, или не выполняется повторно (например, в случае вышеупомянутых картинок) или выполняется с рандомизированным экспоненциальным откатом.

Несмотря на эти меры безопасности, задачи дают сбой одна за другой, и Borg их перезапускает, что еще больше снижает количество рабочих задач. В результате некоторые графы на информационной панели становятся красными, и уходит вызов SR-инженерам. В ответ на это инженеры временно увеличивают производительность азиатских data-центров, повышая количество доступных задач для сервиса Shakespeare. Это позволяет восстановить работу сервиса в азиатском кластере.

После этого команда SR-инженеров делает постмортем, где детально описывает цепочку событий, варианты улучшения сервиса и количество действий, которые необходимо предпринять для того, чтобы этот сценарий не реализовался вновь. Например, в случае перегрузки сервиса балансировщик нагрузки GSLB перенаправит некоторое количество трафика в соседние data-центры, чтобы вам не пришлось опять волноваться из-за этой проблемы.

Итоги главы

Если системы перегружены, нужно что-то предпринять для того, чтобы это исправить. Как только сервис проходит пороговую точку, лучше допустить заметное пользователям снижение качества работы и появление отдельных ошибок, чем продолжать пытаться полностью обслужить каждый запрос. Понимание того, где находятся эти пороги и как система ведет себя за их пределами, критически важно для владельцев сервисов, желающих избежать каскадных сбоев.

Без должного внимания некоторые изменения в системе, призванные снизить количество фоновых ошибок или способствовать поддержанию устойчивого состояния, могут подвергнуть сервис гораздо большему риску полного отключения. Повторные попытки выполнения запросов, отвод нагрузки от проблемных серверов и их отключение, выделение дополнительных ресурсов для улучшения производительности или снижения задержки — все это может помочь в нормальной ситуации, но также может и повысить вероятность возникновения крупномасштабного отказа. Будьте осторожны при оценке изменений, чтобы гарантировать, что вы не создадите новых проблем, исправив предыдущие.

[136](#) Иногда встречается перевод «технический евангелист». — Примеч. пер.

[137](#) О положительной обратной связи вы можете прочитать в «Википедии»:
https://en.wikipedia.org/wiki/Positive_feedback.

[138](#) Сторожевой таймер зачастую реализуется как поток, который периодически пробуждается, чтобы проверить, была ли выполнена работа с момента последней проверки. Если нет, он предполагает, что сервер завис, и убивает его. Например, запросы определенного типа могут отправляться на сервер с заданным интервалом; то, что один из них не был получен или обработан в ожидаемое время, может указывать на сбой — сбой сервера, системы, отправляющей запросы, или промежуточной сети.

[139](#) Такое предположение не всегда работает по географическим причинам, см. также подраздел «Организация задач и данных» раздела «Shakespeare: пример сервиса» главы 2.

[140](#) Упражнение для читателя: напишите простой симулятор и посмотрите, сколько полезной работы может выполнить бэкенд в зависимости от того, насколько он перегружен, и какое количество повторных попыток можно предпринять.

[141](#) Когда запросы клиента обслуживаются конкретным сервером. — Примеч. пер.

[142](#) В какой-то момент вы обнаружите, что значительная часть фактической нагрузочной способности вашего сервиса — это результат работы сервиса кэширования, и если доступ к этому кэшу будет утрачен, вы никогда не сможете обслуживать так много запросов. Это же наблюдение актуально и для задержки. Кэш

может помочь достичь целевых значений задержки (уменьшая среднее время ответа, когда запрос обслуживается из кэша), которые без него будут недостижимы.

23. Разрешение конфликтов: консенсус в распределенных системах и обеспечение надежности

Автор — Лаура Нолан

Под редакцией Тима Харви

Процессы дают сбои, и иногда их нужно перезапускать. Сбои дают и жесткие диски. Природные катастрофы могут разрушить несколько data-центров в каком-то регионе. SR-инженеры должны прогнозировать сбои такого рода и разработать стратегии, которые помогают поддерживать системы в рабочем состоянии, несмотря на все эти факторы. Эти стратегии обычно влекут за собой запуск подобных систем на нескольких сайтах. Географическое распределение системы относительно прямолинейно, но оно показывает необходимость постоянного наблюдения за состоянием системы, что имеет много нюансов и в целом сложновыполнимо.

Группы процессов могут хотеть надежно приходить к консенсусу по следующим вопросам.

- Какой процесс является лидером группы?
- Какие процессы входят в группу?
- Попало ли сообщение в распределенную очередь?
- Должен ли процесс удерживать аренду?
- Какое значение лежит в хранилище для заданного ключа?

Мы обнаружили, что консенсус в распределенных системах эффективен для сборки надежных общедоступных систем, за состоянием которых требуется постоянное наблюдение. Задача консенсуса в распределенных системах позволяет достичь соглашения для группы процессов, связанных ненадежной коммуникационной сетью. Например, несколько процессов распределенной системы могут нуждаться в возможности формировать устойчивое представление о критическом фрагменте конфигурации независимо от того, удерживается ли распределенная блокировка, или же сообщение в очереди было обработано. Это одна из основных концепций распределенных вычислений, на которую мы полагаемся при создании практически любого из предлагаемых нами сервисов. На рис. 23.1 показана простая модель того, как группа процессов может получить устойчивое представление о состоянии системы с помощью консенсуса.

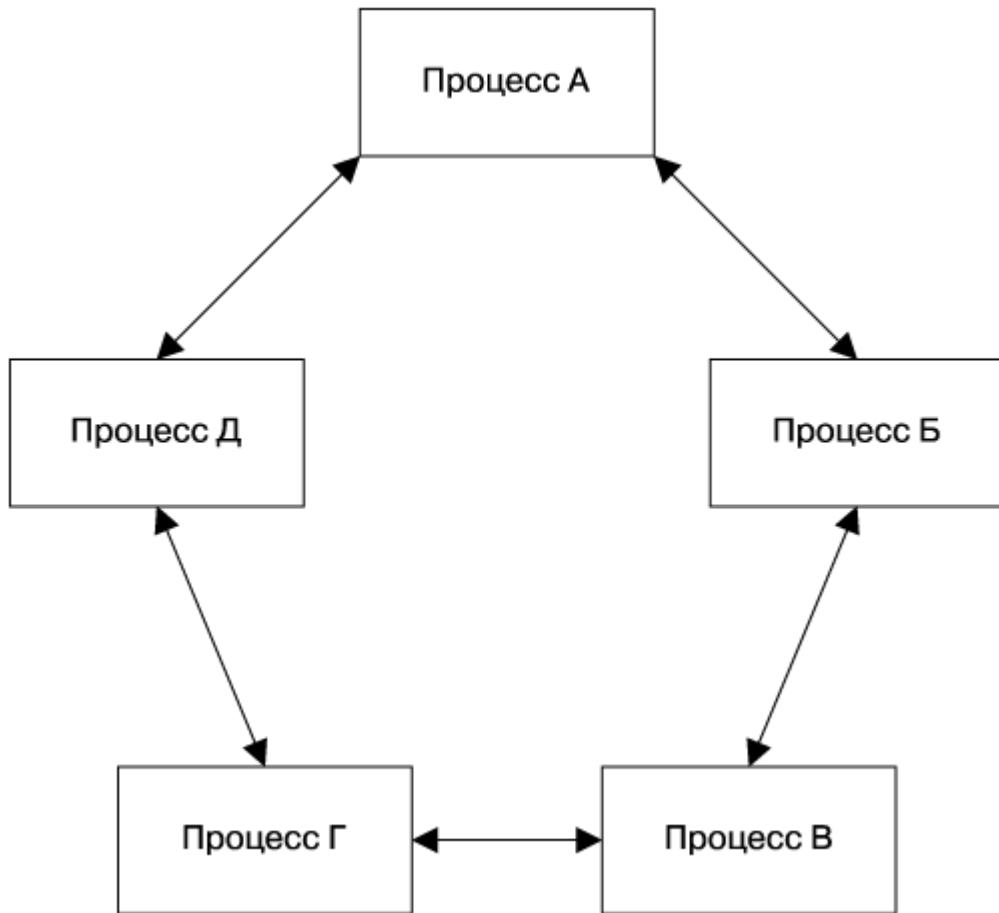


Рис. 23.1. Консенсус в распределенных системах: соглашение между группой процессов

Когда вы видите выборы лидера, разделяемое критическое состояние или распределенные блокировки, стоит использовать *распределенные системы, пришедшие к консенсусу, которые формально доказали свою полезность и были качественно протестированы*. Неформальные подходы к решению этой проблемы могут привести к сбоям и, что еще более неприятно, к неуловимым и сложным для исправления проблемам с устойчивостью данных, которые могут продлить время отключения системы.

ТЕОРЕМА CAP

Теорема CAP ([Fox, 1999], [Brewer, 2012]) утверждает, что распределенная система не может иметь одновременно три следующих свойства:

устойчивое представление о данных в каждом узле;
доступность данных в каждом узле;
толерантность к нарушениям связности сети [Gilbert, 2002].

Логика интуитивно понятна: если два узла не могут общаться друг с другом, поскольку связность сети нарушена, тогда система, рассматриваемая как целое, может остановить обслуживание некоторых или всех запросов на некоторых или всех узлах, снижая доступность, или же обслуживать запросы как обычно, что приведет к появлению неустойчивых представлений о данных в каждом узле.

Поскольку нарушений связности сети избежать нельзя (кабели могут быть перерезаны, пакеты могут потеряться или задержаться из-за перегрузки, аппаратная часть может сломаться, сетевые компоненты могут быть неверно сконфигурированы и т.д.), понимание консенсуса в распределенных системах эквивалентно пониманию устойчивости и возможности работать для конкретного приложения. Коммерческое давление зачастую требует высокого уровня доступности, и многим приложениям необходимо устойчивое представление их данных.

Системные и программные инженеры зачастую знакомы с традиционной семантикой хранилищ данных ACID (Atomicity, Consistency, Isolation, Durability — «атомарность, согласованность, изолированность, долговечность»), но

растущее число технологий создания распределенных хранилищ предоставляет другой набор семантик, известный как BASE (Basically Available, Soft state, Eventual consistency — «доступность в большинстве случаев, неустойчивое состояние, согласованность в конечном счете»). Хранилища данных, которые поддерживают семантику BASE, имеют полезные приложения для данных определенного рода и могут обработать большие объемы данных и транзакций, которые было бы гораздо дороже или даже невозможно обрабатывать с помощью хранилищ данных, поддерживающих семантику ACID.

Большинство систем, которые поддерживают семантику BASE, полагаются на репликацию с несколькими хозяевами, где операции записи могут быть выполнены для разных процессов многопоточно, а также имеется механизм для разрешения конфликтов, зачастую довольно простой — «кто позже, тот и победил». Такой подход часто называют *устойчивостью в конечном счете*. Однако такая устойчивость может привести к удивительным результатам [Lu, 2015], в частности в случае *дрейфа часов* (clock drift), что неизбежно в распределенных системах, или нарушения связности сети [Kingsbury, 2015]¹⁴³. Кроме того, разработчикам трудно создавать системы, которые хорошо работают с хранилищами данных, поддерживающими только семантику BASE. Джек Шуте [Shute, 2013], например, утверждает: «Мы находим, что разработчики тратят значительную часть своего времени на создание крайне сложных и ненадежных механизмов, помогающих справиться с устойчивостью в конечном счете и обрабатывать данные, которые могли устареть. Мы думаем, что такая носка неприемлема для разработчиков и проблемы с устойчивостью должны решаться на уровне баз данных».

Разработчики систем не могут пожертвовать корректностью для того, чтобы добиться надежности или производительности, особенно в критическом состоянии. Например, рассмотрим систему, которая обрабатывает финансовые транзакции: требования к надежности или производительности не приносят большой пользы, если финансовые данные некорректны. Системы должны надежно синхронизировать критические состояния между несколькими процессами. Алгоритмы достижения консенсуса для распределенных систем предоставляют такую функциональность.

Мотивация к использованию консенсуса: сбои координации распределенных систем

Распределенные системы сложны и трудны для понимания, наблюдения и поиска ошибок. Инженеров, работающих с ними, часто удивляет их поведение при сбоях. Последние происходят довольно редко, и, как правило, работа системы в таких условиях не тестируется. Очень трудно обосновывать поведение системы во время сбоев. Нарушения связности сети вызывают особые сложности — проблема, которая, как могло показаться, вызвана таким нарушением, может оказаться результатом:

- очень медленной работы сети;
- потери некоторых, но не всех сообщений;
- подавления, произошедшего только в одном направлении.

В следующих разделах рассмотрены примеры проблем, наблюдавшихся в реальных распределенных системах, также в

них обсуждается, как выбор лидера и алгоритмы достижения распределенного консенсуса можно было использовать для предотвращения таких проблем.

Пример 1: возникновение двух и более ведущих в схеме с ведущим и ведомыми

Сервис представляет собой репозиторий контента, который позволяет нескольким пользователям взаимодействовать друг с другом. Для надежности он применяет наборы, состоящие из двух реплицированных файловых серверов, находящихся в разных стойках. Сервис должен избежать записывания данных в оба файловых сервера, поскольку это может привести к повреждению данных, после которого их, возможно, нельзя будет восстановить.

Каждая пара файловых серверов имеет одного лидера и одного ведомого. Серверы наблюдают друг за другом с помощью контрольных сигналов. Если один файловый сервер не может связаться со своим партнером, он отправляет ему команду STONITH (Shoot The Other Node in the Head — «Выстрелить другому узлу в голову»), чтобы отключить его и завладеть его файлами. Такой прием является стандартным методом снижения количества проявлений этой проблемы, несмотря на то что, как мы увидим, это неправильный способ.

Что произойдет, если сеть станет медленной или начнет отклонять пакеты? В этом случае файловые серверы превысят тайм-ауты контрольных сигналов и, как и было задумано, отправят команды STONITH своим партнерам, чтобы стать ведущими. Однако команды могут быть не доставлены из-за ошибок работы сети. Пары файловых серверов в таком случае могут оказаться в состоянии, когда оба узла должны быть активными для одного ресурса или оба отключены, поскольку

получили команду STONITH. Это приводит либо к повреждению, либо к недоступности данных.

Проблема заключается в том, что система пытается решить задачу выбора лидера с помощью простых тайм-аутов. Выбор лидера — это переформулированная задача достижения распределенного асинхронного консенсуса, которую нельзя решить правильно с помощью контрольных сигналов.

Пример 2: восстановление после сбоя требует вмешательства человека

Высокосегментированная система баз данных имеет первичную реплику для каждого сегмента, который синхронно реплицируется во вторичный сегмент в другом дата-центре. Внешняя система проверяет состояние первичных реплик и, если они больше не работают, повышает ранг вторичной реплики до первичной. Если первичная реплика не может определить состояние своей вторичной реплики, она становится недоступной и отправляет сигнал человеку, чтобы избежать сценария, описанного в примере 1.

Это решение не приводит к потере данных, но ухудшает их доступность. Оно также без необходимости повышает операционную нагрузку на инженеров, поддерживающих систему, а потребность в человеческом вмешательстве плохо масштабируется. Такого рода события, когда первичная и вторичная реплики испытывают проблемы в общении, могут произойти, если возникнет крупная проблема с инфраструктурой, когда отвечающие за нее инженеры могут быть перегружены другими задачами. Если сеть так сильно затронута, что выбрать мастера с помощью распределенного консенсуса нельзя, человек, скорее всего, также не сможет сделать это.

Пример 3: некорректные алгоритмы членства в группе

Система имеет компонент, который выполняет индексирование и поиск по сервисам. В начале работы узлы используют протокол-«сплетник» для обнаружения друг друга и присоединения к кластеру. В случае, когда из-за нарушения связности сети кластер сегментируется, каждая сторона некорректно выбирает мастера и принимает операции по записи и удалению данных, что приводит к возникновению двух и более ведущих и повреждению данных.

Задача определения устойчивого представления членства в группе процессов — это еще один пример задачи распределенного консенсуса. Фактически многие проблемы распределенных систем на практике оказываются одной из множества версий распределенного консенсуса, включая выбор мастера, определение членства в группе, все виды распределенного блокирования и выдачи аренды, надежной распределенной буферизации и доставки сообщений, а также обслуживание любого рода критического разделенного состояния, за которым постоянно должна наблюдать группа процессов. Все эти проблемы могут быть решены путем использования алгоритма распределенного консенсуса, который формально доказал свою корректность и чья реализация была тщательно протестирована. Прямолинейные решения таких задач наподобие контрольных сигналов и протоколов-«сплетников» на практике всегда будут иметь проблемы с надежностью.

Как работает распределенный консенсус

Проблема консенсуса имеет несколько вариантов. При работе с распределенными системами мы заинтересованы в достижении асинхронного распределенного консенсуса, что

применяется к средам с потенциально несвязанными задержками при анализе сообщений. (Синхронный консенсус применяется к системам реального времени, в которых специализированное аппаратное обеспечение гарантирует, что сообщения будут приходить точно в обозначенный срок.)

Алгоритмы достижения распределенного консенсуса могут быть двух видов: *crash-fail* (что подразумевает, что «упавшие» узлы никогда не возвращаются в систему) или *crash-recover*. Алгоритмы *crash-recover* гораздо более полезны, чем *crash-fail*, поскольку большинство проблем в реальных системах являются временными по своей природе из-за медленной сети, перезапусков и т.д.

Алгоритмы могут работать как с византийскими, так и с невизантийскими сбоями. *Византийский сбой* происходит, когда процесс передает некорректное сообщение из-за ошибки или вредоносной активности, их относительно просто обрабатывать, но и встречаются они реже.

Технически решение задачи достижения распределенного асинхронного консенсуса в ограниченное время невозможно. Как доказано теоремой Фишера, Линча и Паттерсона о невозможности консенсуса в системе со сбоями (FLP impossibility result), удостоенной премии Дейкстры [Fischer, 1985], ни один алгоритм достижения распределенного асинхронного консенсуса не может гарантировать прогресс при ненадежной работе сети.

На практике мы подходим к решению проблемы достижения распределенного асинхронного консенсуса в ограниченное время, гарантуя, что система будет иметь достаточное количество рабочих реплик, а также соединение с сетью, что позволит стабильно прогрессировать большую часть времени. Вдобавок система должна иметь выдержку с рандомизированными задержками. Это позволяет пред-

отвращать ситуацию, когда выполнение повторных попыток вызовет каскадный эффект, и избежать проблемы конфликтующих заявителей, которая будет описана позже в этой главе. Протоколы гарантируют безопасность, а адекватная избыточность в системе способствует живучести.

Оригинальным решением проблемы достижения распределенного консенсуса был протокол Лампорта Paxos [Lamport, 1998], но существуют и другие протоколы, способные решить эту проблему, например Raft [Ongaro, 2014], Zab [Junqueira, 2011] и Mencius [Mao, 2008]. Сам протокол Paxos имеет множество вариаций, предназначенных для повышения производительности [ZooKeeper, 2014]. Они обычно отличаются одной деталью, например возможностью назначения на роль лидера одного процесса для упрощения протокола.

Обзор Paxos: пример протокола. Paxos работает как последовательность предложений, которые могут быть приняты или не приняты большинством процессов системы. Если предложение не принято, оно дает сбой. Каждое предложение имеет номер последовательности, который навязывает прямой порядок всех операций системы.

В первой фазе работы протокола заявитель отправляет порядковый номер получателям. Каждый получатель согласится с предложением только в том случае, если еще не видел предложения с более высоким номером. Заявители могут отправить более высокий порядковый номер, если это необходимо. Они должны использовать уникальные порядковые номера, например полученные из непересекающихся множеств или путем внедрения их имени хоста в порядковый номер.

Если заявитель получает согласие от большинства получателей, он может зафиксировать предложение, отправив

сообщение о завершении транзакции со значением.

Прямая последовательность предложений решает любые проблемы, связанные с порядком сообщений системы. Требование, которое заключается в том, что для фиксации необходимо большинство, означает, что два разных значения не могут быть зафиксированы для одного предложения, поскольку любые два большинства будут пересекаться как минимум в одном узле. Получатели должны вести журнал в постоянном хранилище, в который они будут записывать информацию о принятых предложениях, поскольку должны соблюдать эти гарантии после перезапуска.

Paxos сам по себе не очень полезен: все, что он позволяет вам сделать, — это достичь соглашения о значении и один раз предложить число. Поскольку лишь кворум определенного размера должен согласиться со значением, любой заданный узел может не иметь полного представления о наборе значений, с которыми уже согласились. Это ограничение верно для большинства алгоритмов достижения распределенного консенсуса.

Шаблоны системной архитектуры для распределенного консенсуса

Алгоритмы достижения распределенного консенсуса низкоуровневые и примитивные: они просто позволяют набору узлов один раз согласиться со значением. Они плохо подходят для выполнения реальных задач по проектированию. Полезными такие алгоритмы делает добавление высокоуровневых системных компонентов вроде хранилищ данных или конфигурации, очередей, блокировок и сервисов выбора лидеров — все это предоставляет практическую функциональность, которой не имеют сами алгоритмы.

Использование высокоуровневых компонентов снижает сложность для проектировщиков системы. Также это позволяет алгоритмам достижения распределенного консенсуса изменяться при необходимости в ответ на изменения среды, в которой работает система, или на изменения в нефункциональных требованиях.

Многие системы, которые успешно применяют алгоритмы достижения консенсуса, делают это как клиенты некоторого сервиса, реализующего эти алгоритмы, например Zookeeper, Consul и т.д.

Система Zookeeper [Hunt, 2010] стала первой системой достижения консенсуса с открытым исходным кодом и сразу завоевала популярность в отрасли, поскольку ее просто использовать даже в приложениях, которые не были разработаны для применения распределенного консенсуса. Сервис Chubby заполняет такую же нишу в компании Google. Его авторы указывают [Burrows, 2006], что предоставление примитивов консенсуса как сервиса, а не как библиотеки, которую инженеры встраивают в свои приложения, освобождает тех, кто будет поддерживать приложение, от необходимости развертывать системы так же, как и общедоступные системы достижения консенсуса: запуск правильного количества реплик, работа с членством в группе, налаживание производительности и т.д.

Надежные машины с реплицированным состоянием

Машина с реплицированным состоянием (replicated state machine, RSM) — это система, которая выполняет одинаковый набор операций в одном и том же порядке в нескольких процессах. RSM — это основной элемент полезных компонентов распределенных систем и сервисов, таких как

хранилища данных или конфигурации, блокировок и выбора лидера (более подробно будет описано позже).

Операции на RSM заказываются глобально с помощью алгоритма достижения консенсуса. Это очень мощная концепция: в нескольких статьях ([Aguilera, 2010], [Kirsch, 2008], [Schneider, 1990]) показывается, что любая детерминистская программа может быть реализована как общедоступный реплицированный сервис, если его выполнить как RSM.

Как показано на рис. 23.2, машина с реплицированным состоянием — это система, реализованная на логическом уровне, расположенным над алгоритмом достижения консенсуса. Этот алгоритм справляется с соглашением о порядке операций, и RSM выполняет операции в заданном порядке. Поскольку не каждый член группы консенсуса обязательно является членом каждого кворума, RSM должны синхронизировать состояние с помощью одноранговых узлов.

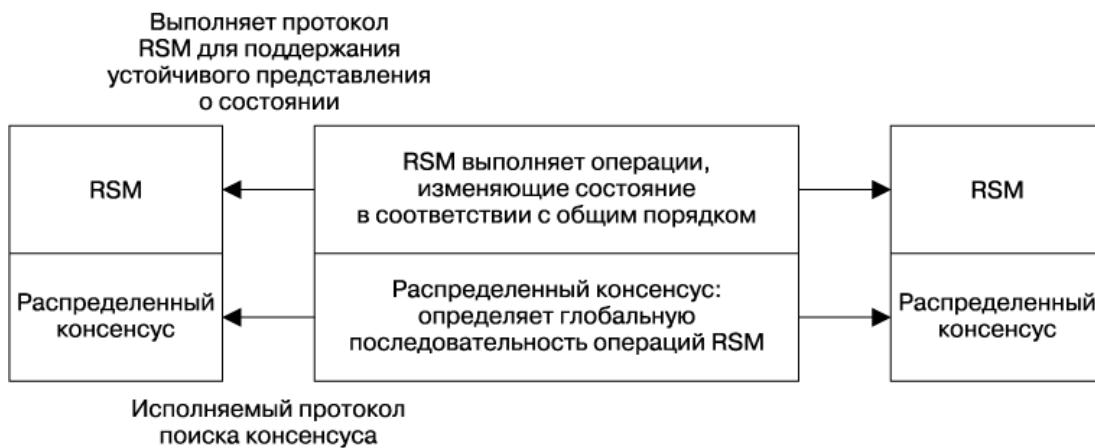


Рис. 23.2. Отношение между алгоритмом достижения консенсуса и машиной с реплицированным состоянием

Надежные реплицированные хранилища данных и конфигураций

Надежные реплицированные хранилища данных являются прикладным уровнем машин с реплицированным состоянием.

Реплицированные хранилища данных используют алгоритм достижения консенсуса на критическом этапе своей работы. Поэтому производительность, пропускная способность и возможность масштабироваться очень важны для таких проектов. Как и хранилища данных, построенные с помощью других технологий, основанные на консенсусе хранилища данных могут предоставить множество семантик устойчивости для операций чтения, что значительно отличает способ масштабирования этого хранилища от прочих. Эти компромиссы рассмотрены в разделе «Производительность систем с распределенным консенсусом» далее в этой главе.

Другие системы (нераспределенные, но основанные на консенсусе) зачастую попросту полагаются на временные метки для создания граничного значения для возраста возвращаемых данных. Временные метки очень проблематично создать в распределенных системах, поскольку невозможно гарантировать, что часы синхронизированы на всех машинах. Spanner [Corbett, 2012] переадресует эту проблему путем моделирования самого худшего случая неопределенности и замедления обработки там, где это необходимо, для разрешения неопределенности.

Общедоступная обработка, применяющая алгоритм выбора лидера

Выбор лидера в распределенных системах — это задача, эквивалентная достижению распределенного консенсуса. Реплицированные сервисы, которые используют одного лидера для выполнения некоего вида работы в системе, широко распространены, механизм одного лидера — это способ гарантировать взаимное исключение на низком уровне.

Дизайн такого вида подходит для тех случаев, когда функции сервиса-лидера могут быть выполнены одним процессом или сегментированы. Дизайнеры системы могут

создать общедоступный сервис, написав его как простую программу, реплицировав этот процесс и задействовав выбор лидера для того, чтобы гарантировать, что в любой заданный момент времени будет работать только один лидер (рис. 23.3). Зачастую работа, которую выполняет лидер, заключается в координировании некоторого пула рабочих систем. Этот шаблон был использован в GFS [Ghemawat, 2003] (которая была заменена на Colossus) и хранилище ключей-значений Bigtable [Chang, 2006].

В компонентах такого вида, в отличие от реплицированных хранилищ данных, алгоритм достижения консенсуса не находится на критическом этапе основной работы, которую выполняет система, поэтому пропускная способность обычно не является серьезной проблемой.

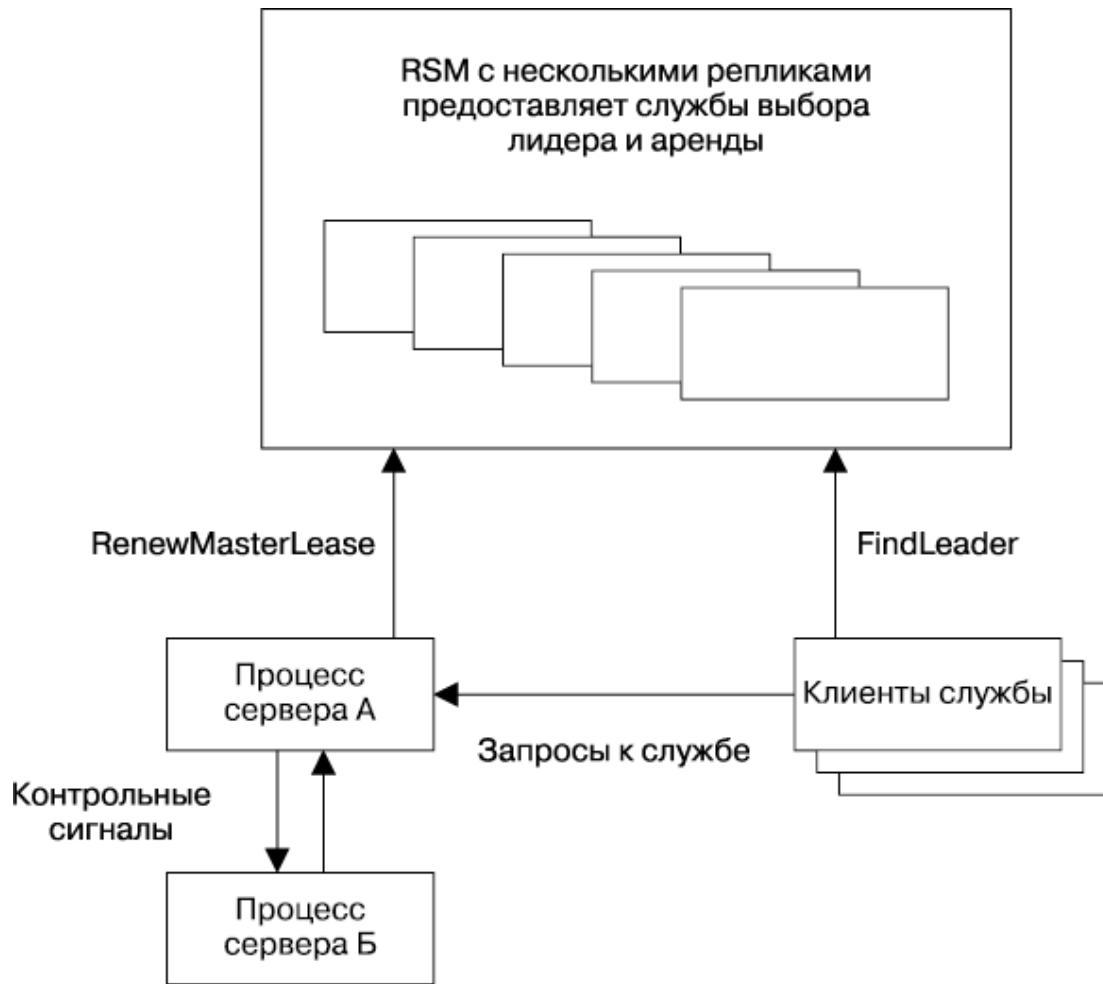


Рис. 23.3. Общедоступная система, применяющая реплицированный сервис для выбора лидера

Распределенная координация и блокировка сервисов

Барьером в распределенных вычислениях является примитив, который блокирует группу процессов до тех пор, пока не будет выполнено какое-то условие (например, пока не будут завершены все части одной фазы расчетов). Барьер, по сути, разделяет распределенные вычисления на логические фазы. Например, барьер может быть использован при реализации модели MapReduce [Dean, 2004], которая позволяет гарантировать, что вся фаза Map будет выполнена до того, как начнется фаза Reduce (рис. 23.4).

Барьер может быть реализован одним процессом-координатором, но это добавляет единую точку отказа, что обычно неприемлемо. Барьер также может быть реализован как RSM. Сервис консенсуса Zookeeper может реализовать шаблон барьера (см. [Hunt, 2010] и [ZooKeeper, 2014]).

Блокировки — это еще один полезный примитив, который может быть реализован как RSM. Рассмотрим распределенную систему, в которой рабочие процессы автоматически потребляют входные файлы и выдают результат. Распределенные блокировки могут быть использованы для предотвращения того, что множество работников будут обрабатывать один и тот же файл. На практике очень важно применять обновляемую аренду с тайм-аутами вместо блокировок, поскольку это предотвратит появление бесконечных блокировок, созданных давшими сбой процессорами. Мы не будем в этой главе рассматривать распределенные блокировки, но нужно иметь в виду, что это низкоуровневые системные примитивы, которые стоит использовать осторожно. Большая часть приложений должна задействовать более высокоуровневые системы, которые предоставляют распределенные транзакции.

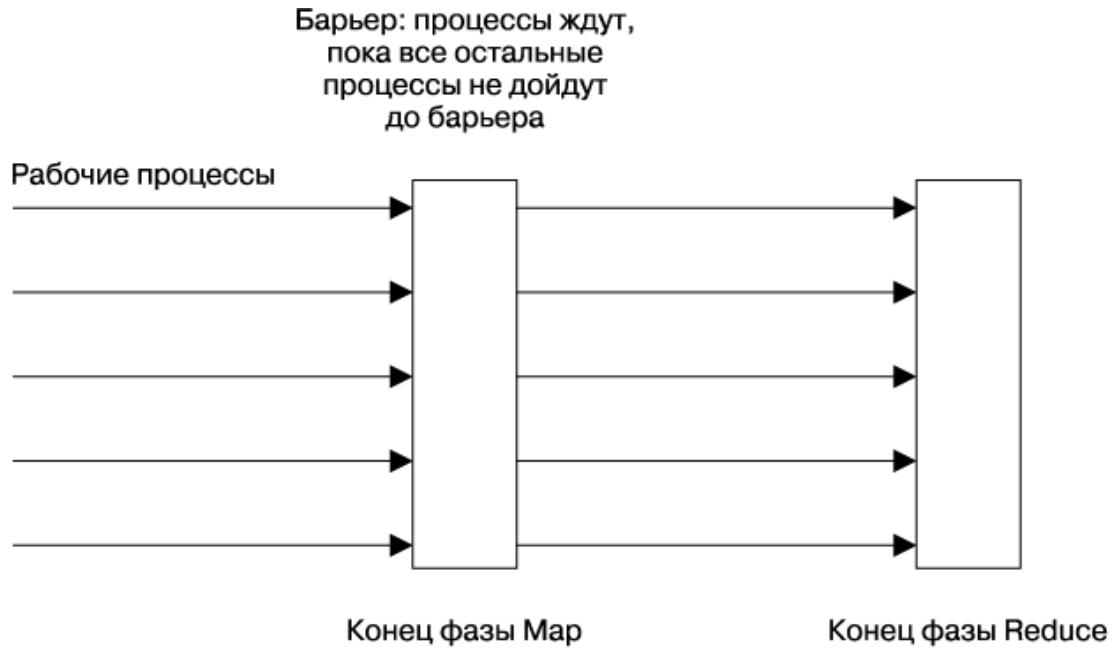


Рис. 23.4. Барьеры для координации процессов при вычислениях MapReduce

Надежный распределенный механизм помещения в очередь и отправки сообщений

Очереди распространены в структурах данных, зачастую их применяют для распределения задач между каким-то количеством рабочих процессов.

Системы, основанные на очередях, могут выдержать сбои и потерю рабочих узлов относительно легко. Однако система должна гарантировать, что указанные задачи будут успешно обработаны. Для этой цели рекомендуется использовать систему аренды (она рассмотрена ранее в разделе о блокировках) вместо прямого удаления из очереди. Недостатком систем, основанных на очередях, является то, что потеря очереди мешает работать всей системе. Реализация очереди как RSM может снизить этот риск и сделать всю систему гораздо более работоспособной.

Атомарная трансляция — это примитив распределенных систем, в котором сообщения доставляются надежно и в одинаковом порядке всем получателям. Он является особенно мощной концепцией распределенных систем, которая очень полезна при разработке дизайна практических систем. Множество инфраструктур публикации-подписки существует для использования дизайнераами системы, несмотря на то что не все они гарантируют атомарность. Чандра и Туг [Chandra, 1996] показывают равенство атомарной трансляции и консенсуса.

Шаблон размещения в очереди для распределения работы, который использует очереди как инструмент балансировки нагрузки (рис. 23.5), можно рассматривать как двухсторонний обмен сообщениями. Системы обмена сообщениями также обычно реализуют очередь публикации — подписки, где сообщения могут потребляться большим количеством клиентов, подписанных на канал или тему. В этом случае, если работа идет по схеме «один ко многим», сообщения в очереди хранятся в виде устойчивого упорядоченного списка.

Системы публикации — подписки могут быть использованы для многих типов приложений, которые требуют от клиента подписаться для получения уведомлений о том, что произошло какое-то событие. Системы публикации — подписки можно применить также для реализации понятных распределенных кэшей.

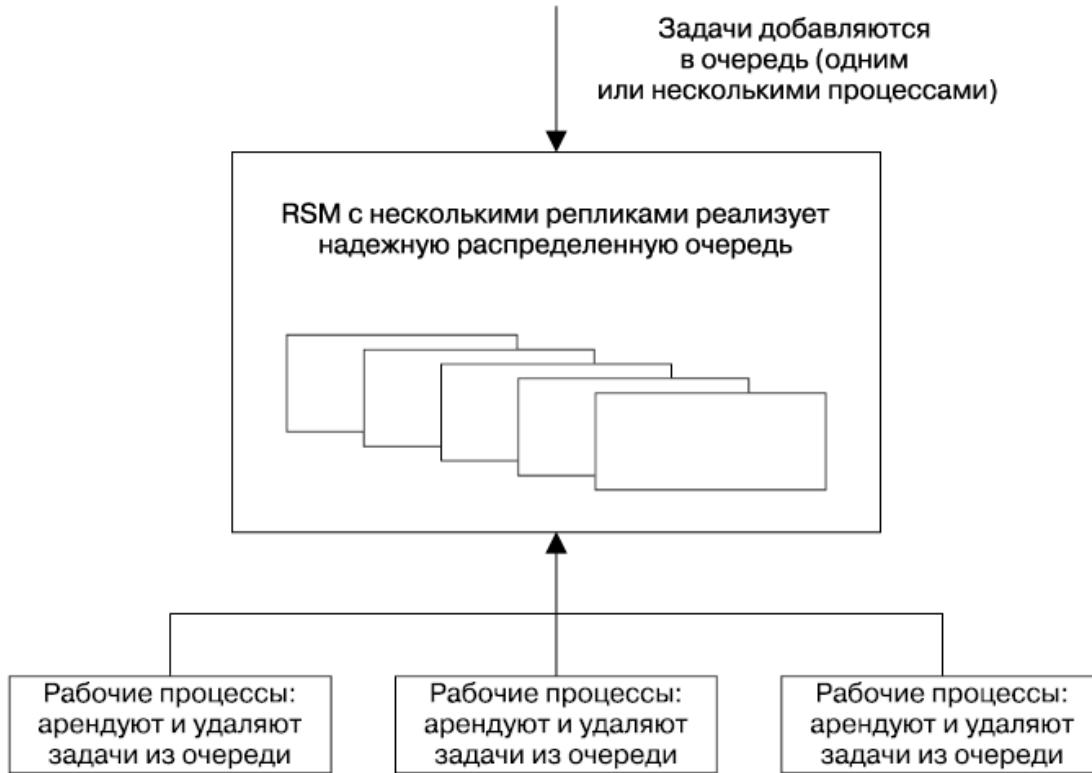


Рис. 23.5. Система распределения работы, основанная на очередях, использующая надежный, основанный на консенсусе компонент размещения в очередях

Системы для размещения в очереди и обмена сообщениями зачастую требуют отличной пропускной способности, но, поскольку пользователи работают с ними редко, им не нужна очень малая задержка. Однако очень длительные задержки в системах, похожих на описанную, которые имеют множество работников, получающих задачи из очереди, могут стать проблемой, если процент времени обработки для каждой задачи значительно вырастет.

Производительность систем с распределенным консенсусом

По общему мнению, алгоритмы достижения консенсуса слишком медленные и слишком дорогие для использования во многих системах, которые требуют широкой полосы пропускания и малой задержки [Bolosky, 2011]. Эта концепция

попросту неверна — несмотря на то что реализовываться она может медленно, существует несколько приемов, которые могут улучшить ее производительность. Алгоритмы достижения распределенного консенсуса лежат в основе многих критически важных систем компании Google, описанных в [Ananatharayan, 2013], [Burrows, 2006], [Corbett, 2012] и [Shute, 2013], и они доказали свою высокую эффективность на практике. Масштаб компании Google в этой ситуации является не преимуществом, а скорее недостатком, поскольку из-за него возникают две главные сложности: наши наборы данных, как правило, велики, а системы работают на больших географических расстояниях. Крупные наборы данных, к тому же имеющие несколько реплик, обусловливают значительную стоимость вычисления, а значительные расстояния увеличивают задержку между репликами, что, в свою очередь, снижает производительность.

Не существует «лучшего» алгоритма распределенного консенсуса и алгоритма репликации конечных машин с точки зрения производительности, поскольку она зависит от нескольких факторов, связанных с нагрузкой, целевыми значениями производительности системы и способом развертывания последней¹⁴⁴. В следующих разделах рассматриваются исследования, предназначенные для того, чтобы лучше понять способы достижения распределенного консенсуса. Многие описанные системы уже доступны и используются.

Нагрузку можно варьировать множеством способов, и понимание того, как она может изменяться, критически важно для обсуждения производительности. В системах с консенсусом нагрузка может зависеть от следующих условий:

- пропускной способности — количества предложений, сделанных элементом за единицу времени при пиковой нагрузке;
- типа запросов — доли операций, которые могут изменять состояние;
- семантики устойчивости, необходимой для операций чтения;
- размеров запросов, если размер данных, представляющих собой полезную нагрузку, может различаться.

Стратегии развертывания также могут быть различными.

- Развертывание проводится локально или на большой площади?
- Какого рода кворумы применяются и где находится большинство процессов?
- Использует ли система сегментирование, конвейерную или пакетную обработку?

Многие системы с консенсусом задействуют определенный процесс-лидер и требуют, чтобы все запросы переходили на этот особый узел. Как показано на рис. 23.6, в результате этого производительность системы, воспринимаемая клиентами в разных географических локациях, может значительно различаться просто потому, что множество узлов имеют длительное время обращения к процессу-лидеру.

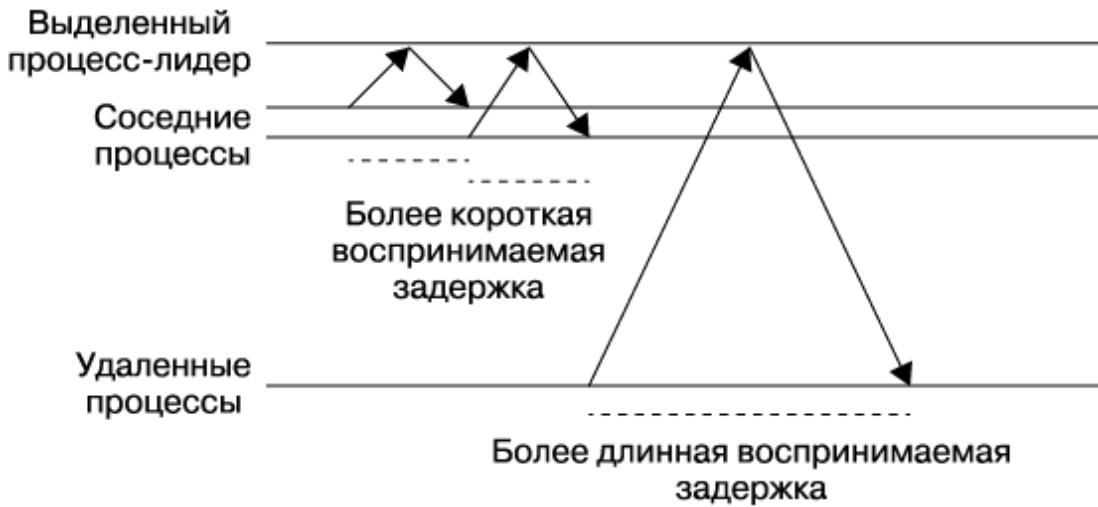


Рис. 23.6. Влияние удаленности от процесса-сервера на воспринимаемую производительность

Multi-Paxos: детализированный поток сообщений

Протокол Multi-Paxos использует *сильный процесс-лидер*: если лидер не был выбран или произошел какой-то сбой, требуется прохождение всего одного цикла между заявителем и кворумом получателей для достижения консенсуса. Использование сильного процесса-лидера оптимально с точки зрения количества передаваемых сообщений, этот подход применяется во многих протоколах достижения консенсуса.

На рис. 23.7 показано исходное состояние, когда заявитель выполняет первую фазу подготовки/обещания. На этом этапе создается новое пронумерованное представление — время действия лидера. При последующих вызовах протокола, поскольку представление остается тем же самым, выполнять первую фазу не обязательно, так как заявитель, который создал представление, может просто отправлять сообщения Accept, и консенсус будет достигнут, как только будет получен кворум ответов.

Еще один процесс в группе может принять на себя роль заявителя для создания сообщений в любой момент, но

изменение заявителя может стоить производительности. Это потребует дополнительного цикла для выполнения фазы 1 протокола и, что более важно, может вызвать появление конфликтующих заявителей, когда предложения раз за разом прерывают друг друга и ни одно не принимается (рис. 23.8). Поскольку этот сценарий является формой активной блокировки, он может продолжаться бесконечно.

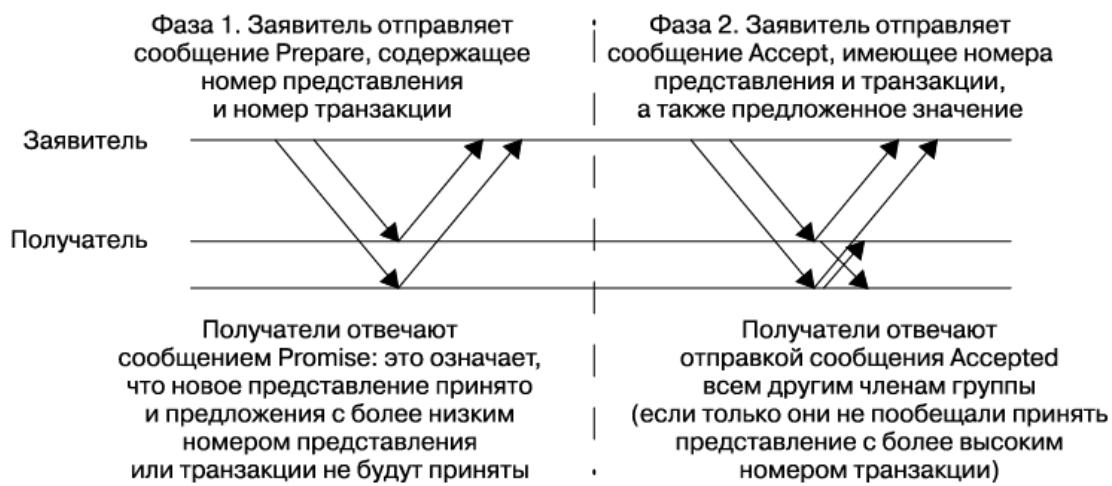


Рис. 23.7. Базовый поток сообщений протокола Multi-Paxos

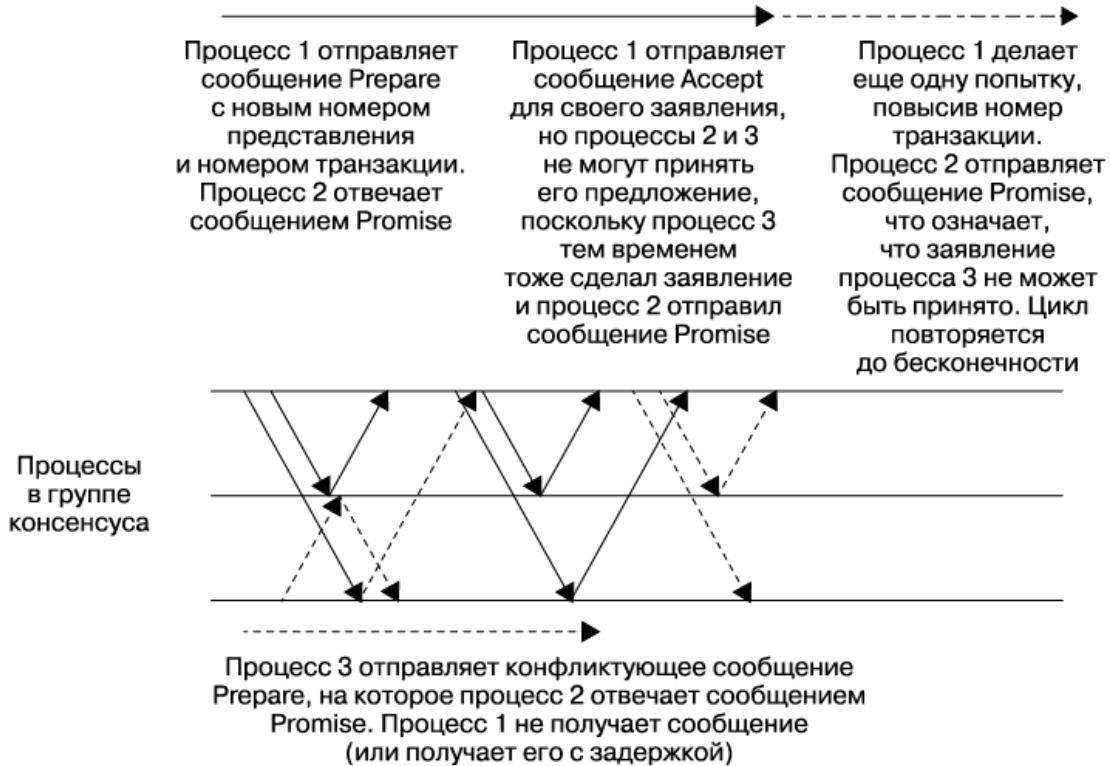


Рис. 23.8. Конфликтующие заявители в Multi-Paxos

Все применяемые на практике системы достижения консенсуса обычно решают эту проблему коллизий выбором процесса-заявителя, который будет создавать все предложения системы, или чередованием заявителя, выделяющего каждому процессу отдельное место в этих предложениях.

Для систем, которые используют процесс-лидер, его выбор должен быть тщательно настроен для того, чтобы сбалансировать вероятность недоступности системы из-за того, что лидера нет, с риском появления конфликтующих заявителей. Важно реализовать правильные стратегии таймаутов и выдержки. Если несколько процессов обнаруживают, что лидера нет, и пытаются стать лидером одновременно, то, скорее всего, ни один из них не преуспеет (опять же из-за конфликтующих заявителей). Здесь лучше всего поможет введение элемента случайности. Raft [Ongaro, 2014], например, имеет хорошо продуманный метод выбора процесса-лидера.

Масштабирование нагрузки, связанной с операциями чтения

Масштабирование нагрузки, связанной с чтением, зачастую критически важно, поскольку большая ее часть связана именно с этими операциями. Реплицированные хранилища данных обладают преимуществом, которое заключается в том, что данные доступны в нескольких местах. Это означает, что, если не требуется высокий уровень устойчивости для всех операций чтения, данные могут быть считаны из любой реплики. Прием чтения из реплик хорошо работает для определенных приложений, например системы Photon компании Google [Ananatharayan, 2013], которая использует распределенный консенсус для того, чтобы координировать работу нескольких конвейеров. Photon применяет атомарные операции сравнения и установки для изменения состояния, основанные на атомарных регистрах, которые гарантируют корректность и целостность данных. Однако операции чтения могут быть обслужены с помощью любой реплики, поскольку, если данные устареют, будет только выполнена лишняя работа — это не будет считаться ошибкой [Gupta, 2015]. Компромисс стоит того.

Для того чтобы гарантировать актуальность и целостность считываемых данных при любых изменениях, сделанных до операции чтения, необходимо произвести одну из следующих операций.

- Выполнить операцию по достижению консенсуса в режиме «только чтение».
- Считать данные из реплики, которая гарантированно является самой свежей. В системе, использующей стабильный процесс-лидер (как поступают многие реализации распределенного консенсуса), лидер может дать такую гарантию.

- Применить кворумные аренды, в которых некоторым репликам выдается аренда на все данные системы или их часть, что позволяет выполнять очень устойчивые операции чтения локально за счет некоторого количества производительности при записи. Этот прием детально рассматривается в следующем разделе.

Кворумная аренда

Кворумная аренда [Moraru, 2014] – это недавно разработанный способ оптимизации производительности распределенного консенсуса, нацеленный на снижение задержки и повышение пропускной способности для операций чтения. Как говорилось ранее, в случае использования классического протокола Paxos и большинства других протоколов распределенного консенсуса выполнение очень устойчивых операций чтения (например, гарантирующих наличие самых свежих данных) требует либо реализации операции распределенного консенсуса, которая читает из кворума реплик, либо наличия стабильной реплики-лидера, которая гарантированно видела все недавние операции изменения статуса. Во многих системах число операций чтения значительно превосходит количество операций записи, поэтому такая зависимость от распределенной операции либо одной реплики ухудшает задержку и системную полосу пропускания.

При кворумной аренде попросту выдается аренда на чтение некоторому подмножеству состояний реплицированных хранилищ данных. Аренда выдается на конкретный (обычно небольшой) промежуток времени. Любая операция, которая изменяет состояние этих данных, должна быть подтверждена всеми репликами кворума. Если любая из реплик становится недоступной, данные не могут быть модифицированы до тех пор, пока не истечет срок аренды.

Кворумная аренда особенно полезна для нагрузки, когда операции чтения для определенных подмножеств данных сконцентрированы в одном географическом регионе.

Производительность распределенного консенсуса и задержка сети

Системы консенсуса сталкиваются с двумя серьезными физическими ограничениями производительности, когда изменяют состояние. Одно из них заключается во времени обращения к сети, а второе — во времени, которое требуется для записи данных в устойчивое хранилище (его мы рассмотрим позже).

Время обращения к сети может значительно различаться в зависимости от местоположения источника и приемника, а также наличия перегрузки сети. Время цикла прохождения данных между машинами внутри одного дата-центра должно измеряться миллисекундами. Обычное время цикла (round-trip-time, RTT) внутри Соединенных Штатов составляет 45 миллисекунд, а между Нью-Йорком и Лондоном — 70 миллисекунд.

Производительность системы с консенсусом внутри локальной сети может быть сравнима с производительностью асинхронной системы «лидер — ведомый» [Bolosky, 2011], которую многие традиционные базы данных применяют для репликации. Но, чтобы воспользоваться большей частью преимуществ доступности систем с распределенным консенсусом, нужно, чтобы реплики были удалены друг от друга, для того чтобы они находились в разных областях отказов.

Многие системы с консенсусом используют в качестве коммуникационного протокола TCP/IP. Протокол TCP/IP ориентирован на соединения и обеспечивает высокие гарантии

надежности в отношении последовательности сообщений FIFO. Однако при создании нового TCP/IP-подключения требуется перед получением или отправкой данных получить трехстороннее подтверждение установки соединения, с проходом информации по сети туда и обратно. Медленный старт протокола TCP/IP изначально ограничивает полосу пропускания соединения до тех пор, пока не будут установлены его границы. Начальный размер окна TCP/IP изменяется от 4 до 15 Кбайт.

Медленный старт протокола TCP/IP, возможно, не является проблемой для процессов, которые формируют группу консенсуса: они устанавливают соединения друг с другом и будут поддерживать их открытыми для повторного использования, поскольку часто общаются друг с другом. Однако для систем с очень большим количеством клиентов поддержание постоянных открытых соединений с кластерами консенсуса может быть нецелесообразным, поскольку открытые соединения TCP/IP потребляют некоторое количество ресурсов, например дескрипторы файлов, в дополнение к генерируемому keepalive-сообщений. Эти издержки могут оказаться серьезной проблемой для приложений, которые задействуют высокосегментированные хранилища данных, основанные на консенсусе, содержащие тысячи реплик и еще большее количество клиентов. Решением проблемы является использование пула региональных прокси (рис. 23.9), которые будут хранить постоянные соединения TCP/IP с группой консенсуса для того, чтобы избежать появления накладных расходов, связанных с установкой соединения для длинных дистанций. Прокси также могут помочь вам инкапсулировать сегментирование и стратегии балансировки нагрузки, а также обнаружение членов кластеров и лидеров.

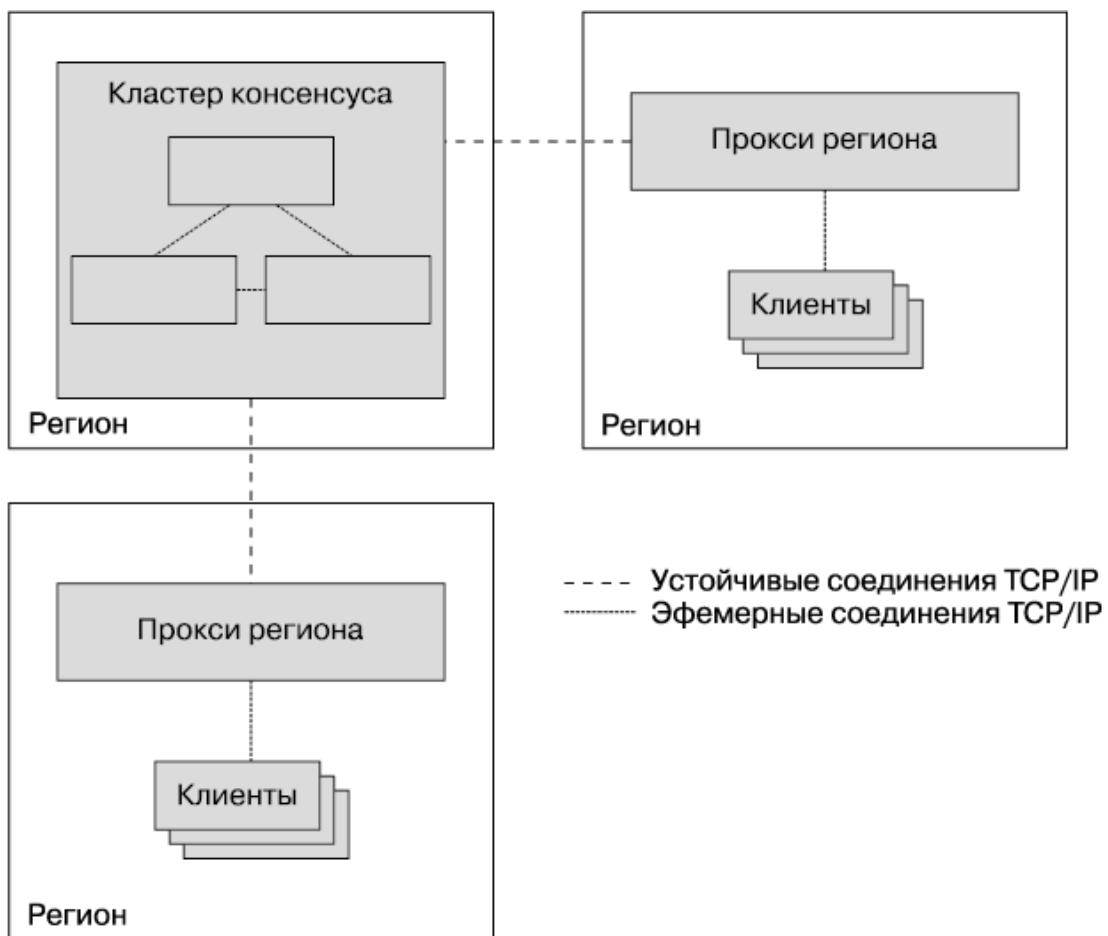


Рис. 23.9. Использование прокси для снижения необходимости открытия соединений TCP/IP между регионами

Размышляем о производительности: Fast Paxos

Fast Paxos [Lamport, 2006] — это версия алгоритма Paxos, разработанная для улучшения производительности для широкомасштабных сетей. Применив Fast Paxos, каждый клиент может отправить сообщения с предложениями непосредственно каждому члену группы приемников вместо того, чтобы действовать через лидера, как он делал бы при использовании алгоритмов Classic Paxos или Multi-Paxos. Суть заключается в том, чтобы заменить одну операцию отправки параллельных сообщений от клиента всем приемникам Fast

Paxos двумя операциями отправки сообщений с помощью Classic Paxos:

- одно сообщение от клиента одному заявителю;
- параллельное сообщение отправляет операцию от заявителя другим репликам.

На первый взгляд может показаться, что Fast Paxos должен быть быстрее алгоритма Classic Paxos. Однако это не так: если клиент системы Fast Paxos имеет высокое RTT (round-trip time — «время оборота») с приемниками, а приемники выполняют быстрые соединения друг с другом, мы заменяем N параллельных сообщений, идущих через медленные узлы сети (в Fast Paxos), на одно сообщение, идущее через медленные узлы, плюс N параллельных сообщений между быстрыми узлами (Classic Paxos). Из-за эффекта хвостовой задержки, наблюдающегося большую часть времени, выполнение одного цикла между медленными узлами с распределенными задержками будет быстрее, чем кворум (как показано в [Junqueira, 2007]), и поэтому в данном случае Fast Paxos окажется медленнее, чем Classic Paxos.

Многие системы объединяют несколько операций в одну транзакцию для получателей, чтобы повысить пропускную способность. Поскольку клиенты действуют как заявители, затрудняется возможность отправки пакетных предложений. Причина заключается в том, что предложения поступают независимо от приемников, поэтому вы не можете устойчиво их объединять.

Стабильные лидеры

Мы видели, как Multi-Paxos выбирает стабильного лидера для того, чтобы улучшить производительность. Zab [Junqueira, 2011] и Raft [Ongaro, 2014] также являются примерами протоколов, которые выбирают стабильного лидера из соображений производительности. Этот подход позволяет оптимизировать операции чтения, поскольку лидер находится в самом новом состоянии, но в то же время несет в себе следующие проблемы.

- Все операции, которые изменяют состояние, должны быть отправлены через лидера, это требование повышает задержку сети для клиентов, которые расположены далеко от лидера.
- Выходная пропускная способность сети процесса-лидера является узким местом сети [Мао, 2008], поскольку его сообщение Accept содержит все данные, связанные со всеми предложениями, а другие сообщения содержат только подтверждения пронумерованных транзакций, не имеющих полезной нагрузки.
- Если лидер располагается на машине, которая испытывает проблемы с производительностью, то пропускная способность всей системы будет понижена.

Практически все системы, использующие распределенный консенсус, которые были спроектированы так, чтобы обращать внимание на производительность, действуют либо шаблон единого стабильного лидера, либо систему сменяемого руководства, в которой каждый пронумерованный алгоритм достижения распределенного консенсуса заранее присвоен реплике (обычно с помощью простого численного идентификатора транзакции). Этот подход применяют

множество алгоритмов, в том числе Mencius [Mao, 2008] и Egalitarian Paxos [Moraru, 2012].

Для широкомасштабных сетей, чьи клиенты распределены географически, а реплики групп консенсуса расположены довольно близко к клиентам, такой выбор лидера приводит к уменьшению наблюдаемой задержки для клиентов, поскольку время обращения запроса к сети для ближайшей реплики в среднем будет меньше, чем для произвольного лидера.

Пакетная обработка

Пакетная обработка, как описано в подразделе «Размышляем о производительности: Fast Paxos» ранее, повышает пропускную способность системы, но все еще оставляет реплики в состоянии покоя, пока они ожидают ответа на отправленные ими сообщения. Неэффективность, представленную свободными репликами, можно исправить, используя *конвейерную обработку*, которая позволяет отправлять по нескольку предложений сразу. Такая оптимизация очень похожа на оптимизацию, которая применялась в случае с протоколом TCP/IP, где протокол пытался поддерживать конвейер заполненным с помощью подхода, который называется «скользящее окно». Конвейерная обработка зачастую комбинируется с пакетной обработкой.

Пакеты запросов в конвейере все еще упорядочиваются глобально с учетом номеров представления и транзакции, поэтому этот метод не нарушает свойства глобального упорядочения, необходимые для запуска машины с реплицированным состоянием. Такой метод оптимизации рассматривается в [Bolosky, 2011] и [Santos, 2011].

Доступ к диску

Вносить записи в журнал, находящийся в устойчивом хранилище, необходимо, поэтому узел, который дал сбой и вернулся в кластер, учитывает любые предыдущие фиксации, которые он сделал по отношению к протекающим транзакциям консенсуса. В протоколе Paxos, например, приемники не могут согласиться с предложением, если они уже согласились с предложением, имеющим больший порядковый номер. Если детали предложения, с которым согласились и которое зафиксировали, не записаны в устойчивое хранилище, то приемник может нарушить протокол, если он даст сбой и перезапустится, что приведет к неустойчивому состоянию

Время, которое требуется для создания записи в журнале на диске, значительно различается в зависимости от используемой аппаратной части или виртуальной среды, но, скорее всего, не превысит нескольких миллисекунд.

Поток сообщений для протокола Multi-Paxos рассматривался в подразделе «Multi-Paxos: детализированный поток сообщений» ранее, но в нем не говорилось, где протокол должен вносить изменения состояния в журнал, хранящийся на диске. Запись на диск должна выполняться, когда процесс внесет изменение, с которым следует считаться. Во второй фазе протокола Multi-Paxos, зависимой от производительности, это будет происходить до того, как приемник отправит сообщение Accepted в ответ на предложение, и до того, как заявитель отправит сообщение Accept, поскольку это сообщение Accept также неявно является сообщением Accepted [Lamport, 1998].

Это означает, что задержка для операции с одним консенсусом включает в себя:

- одну операцию записи для заявителя;
- параллельные сообщения для приемников;

- параллельные записи на диск для приемников;
- возвращаемые сообщения.

Существует версия протокола Multi-Paxos, которая полезна для ситуаций, когда преобладают операции записи на диск: этот вариант не рассматривает сообщение Accept заявителя как неявное сообщение Accepted. Вместо этого заявитель выполняет запись на диск одновременно с другими процессами и отправляет явное сообщение Accept. Далее задержка становится пропорциональной времени, затраченному на отправку двух сообщений и на выполнение синхронной операции записи на диск для кворума процессов.

Если задержка для выполнения небольшой произвольной записи на диск измеряется десятками миллисекунд, уровень операций консенсуса будет ограничен примерно сотней операций в минуту. Эти показатели времени подразумевают, что временем обращения к сети можно пренебречь, и заявитель выполняет журналирование одновременно с приемниками.

Как мы уже видели, алгоритмы достижения распределенного консенсуса зачастую используются как основа для создания машин с реплицированным состоянием. RSM также должны сохранять журналы транзакций для потенциального восстановления (по тем же причинам, что и для любого другого хранилища данных). Журнал алгоритма консенсуса и журнал транзакций RSM могут быть объединены в один. Объединение этих журналов позволяет избежать необходимости постоянно переключаться между записями в два разных места на диске [Boulosky, 2011], что уменьшает время, потраченное на операции поиска. Диски могут

выдерживать больше операций в секунду, и поэтому система как единое целое может выполнять больше транзакций.

В хранилище данных диски имеют и другое назначение, помимо поддержки журналов: состояние системы обычно поддерживается на диске. Записи в журнал должны быть сброшены непосредственно на диск, но записи об изменении состояния могут быть добавлены в кэш и сброшены на диск позже, будучи перестроенными для того, чтобы использовать наиболее эффективное расписание [Bolosky, 2011].

Еще одним вариантом оптимизации является пакетная обработка операций клиента и размещение их в одной операции на заявителе ([Ananatharayan, 2013], [Bolosky, 2011], [Chandra, 2007], [Junqueira, 2011], [Mao, 2008], [Moraru, 2012]). Это погашает фиксированную стоимость журналирования диска и задержки сети для большого количества операций, повышая пропускную способность.

Развертывание распределенных систем, основанных на консенсусе

Наиболее важные решения, которые проектировщики систем должны принимать при развертывании систем, основанных на консенсусе, касаются количества развертываемых реплик и их местоположения.

Количество реплик

Как правило, системы, основанные на консенсусе, работают с использованием *большинства кворумов*, то есть группы из $2f + 1$ реплик могут выдержать f сбоев (для византийских сбоев, когда требуется устойчивость системы к репликам, возвращающим

некорректный результат, нужно применить $3f + 1$ реплик для устойчивости к f сбоям [Castro, 1999]). Для невизантийских сбоев минимальным количеством развертываемых реплик может быть три — если развернуты две реплики, то не будет никакой устойчивости к сбою любого процесса. Три реплики могут выдержать один сбой. Простой большинства систем — это результат запланированного отключения [Kendrick, 2012]: три реплики позволяют системе работать в обычном режиме, когда одна из них отключена для обслуживания (предполагается, что две оставшиеся реплики смогут обработать нагрузку системы с приемлемой производительностью).

Если незапланированный сбой происходит во время окна обслуживания, то система, основанная на консенсусе, становится недоступной. Недоступность такой системы обычно неприемлема, поэтому следует запускать пять реплик, что позволит ей работать и после двух сбоев. Вмешательства не потребуется, если в системе консенсуса останутся работать четыре из пяти реплик, но если остается только три реплики, следует добавить одну или две.

Если система консенсуса теряет так много своих реплик, что не может сформировать кворум, то теоретически она находится в невосстанавливаемом состоянии, поскольку нельзя получить доступ к долговечным журналам как минимум одной из недостающих реплик. Если кворума нет, то, возможно, было принято решение, видимое только недостающим репликам. Администраторы могут иметь возможность навязать изменение членов группы и добавить новые реплики, которые получат всю необходимую информацию у уже существующих реплик, но вероятность потери данных все еще остается — этой ситуации лучше всего избегать.

В случае катастрофы администраторы должны решить, выполнять такое переконфигурирование или подождать какое-то время, пока машины, хранящие состояние системы, не станут доступными. Когда принимаются подобные решения, управление системным журналом (в дополнение к данным наблюдения) становится критически важным. Теоретические статьи зачастую указывают, что консенсус может быть использован для создания реплицированного журнала, но не могут показать, как справляться с репликами, которые могут давать сбой и восстанавливаться (и поэтому пропускать некоторые последовательности решений консенсуса) или подвисать из-за низкой скорости своей работы. Для поддержки устойчивости системы важно, чтобы эти реплики наверстали упущенное.

Реплицированный журнал не всегда является персоной первого класса в теории распределенного консенсуса, но он представляет собой очень важный аспект производственных систем. Raft описывает метод управления устойчивостью реплицированных журналов [Ongaro, 2014], явно определяя, как должны быть заполнены любые лакуны в журналах. Если система Raft, состоящая из пяти экземпляров, потеряет всех своих членов, кроме загрузчика, лидер все еще будет иметь всю информацию о принятых решениях. В то же время, если среди недостающих членов будет находиться и лидер, то нельзя будет гарантировать, что данные реплики будут свежими.

Существует соотношение между производительностью и количеством реплик системы, которым не нужно формировать часть кворума: более медленные реплики, находящиеся в меньшинстве, могут отставать, позволяя кворому быстро работающих реплик действовать быстрее (до тех пор пока быстро работают лидеры). Если производительность реплики значительно меняется, то каждый сбой может снизить

производительность всей системы, поскольку медленные, не поддерживающие общего темпа реплики должны будут войти в кворум. Чем больше сбоев или висящих реплик может выдержать система, тем выше будет ее общая производительность.

При управлении репликами стоит рассмотреть также проблему стоимости, так как каждая реплика задействует ценные вычислительные ресурсы. Если система является одним кластером процессов, стоимость запуска реплик будет относительно небольшой. Однако она может оказаться значительной для систем вроде Photon [Ananatharayan, 2013], которые используют сегментацию, где каждый сегмент представляет собой группу процессов, для которых запущен алгоритм консенсуса. По мере увеличения числа сегментов увеличивается и стоимость каждой дополнительной реплики, поскольку в систему необходимо добавить процессы в количестве, эквивалентном количеству сегментов.

Решение о количестве реплик для любой системы является компромиссом для следующих факторов:

- необходимой надежности;
- частоты плановых отключений, влияющих на систему;
- риска;
- производительности;
- стоимости.

Такие расчеты могут быть своими для каждой системы, так как системы имеют разные целевые значения надежности; некоторые организации выполняют отключения более

регулярно, чем другие; организации используют аппаратное обеспечение разных стоимости, качества и надежности.

Расположение реплик

Решение о том, где разворачивать процессы, которые составляют кластер консенсуса, принимается на основе двух факторов: компромисса между областями отказов, которые система должна обработать, и требований к задержке для системы. На определение расположения реплик влияет множество сложных моментов.

Область отказов является набором компонентов системы, который может стать недоступным в результате одного сбоя. Примеры областей отказов:

- физическая машина;
- стойка дата-центра, которую обслуживает один источник питания;
- несколько стоек дата-центра, которые обслуживает один элемент сетевого оборудования;
- дата-центр, который может стать недоступным из-за того, что был перерезан оптоволоконный кабель;
- множество дата-центров в одной географической области, где произошла природная катастрофа, например ураган.

В общем, по мере увеличения расстояния между репликами увеличивается и время прохождения данных между ними, а также размер сбоя, который система сможет выдержать.

Для большинства систем, основанных на консенсусе, увеличение времени прохождения данных между репликами увеличит и задержку выполнения операций.

Влияние задержки, а также способность пережить сбой в заданной области очень зависят от системы. Архитектуры некоторых систем, основанных на консенсусе, не требуют особенно большой пропускной способности или малой задержки: например, система, которая должна предоставлять услуги по размещению членов группы и выбору лидера, скорее всего, не будет сильно загружена, и если время транзакции консенсуса составляет лишь небольшую долю времени аренды лидера, то производительность некритична.

Системы, ориентированные на пакетную обработку, также меньше зависят от задержки: для увеличения пропускной способности можно увеличить размеры пакетов.

Не всегда имеет смысл последовательно увеличивать размер области отказа, потерю которой система может пережить. Например, если все клиенты, использующие систему, основанную на консенсусе, работают внутри одной области отказа (допустим, в районе Нью-Йорка) и развертывание распределенной системы, основанной на консенсусе, на большей территории позволит ей оставаться доступной во время сбоев в этой области отказа (к примеру, вызванных ураганом «Сэнди»), будет ли оно того стоить? Наверняка нет, поскольку клиенты системы также будут отключены и система не увидит трафик. Дополнительная стоимость с точки зрения задержки, пропускной способности и вычислительных ресурсов не принесет пользы.

Вы должны учитывать необходимость восстановления после катастроф при определении места, где будут размещаться реплики: в системе, которая хранит критически важные данные, реплики консенсуса, по сути, также являются онлайн-

копиями этих данных. Но когда такие данные находятся под угрозой, важно создавать их резервные копии везде, где можно, даже если устойчивые системы, основанные на консенсусе, размещены в нескольких областях отказа. Существует две области отказа, от которых вам не удастся спрятаться: само ПО и человеческие ошибки, допускаемые системными администраторами. Баги в ПО могут проявиться при неожиданных обстоятельствах и вызвать потерю данных, к подобной ситуации может привести и неверное конфигурирование системы. Люди-операторы также могут ошибаться или саботировать работу, из-за чего данные будут потеряны.

Принимая решение о размещении реплик, помните, что наиболее важной мерой производительности является восприятие клиента: в идеале время обращения клиентов к репликам системы, основанной на консенсусе, должно быть минимизировано. В широкомасштабных сетях протоколы без лидера вроде Mencius или Egalitarian Paxos могут иметь лучшую производительность, особенно если ограничения устойчивости приложения означают, что для любой реплики системы можно выполнять только операции чтения, если не выполняется операция консенсуса.

Производительность и балансировка нагрузки

При проектировании развертывания вы должны убедиться, что располагаете достаточной производительностью, способной справиться с загрузкой. В случае *сегментированного развертывания* можете скорректировать производительность, изменив количество сегментов. Однако в системах, которые могут выполнять чтение с членов группы консенсуса, не являющихся лидерами, вы можете повысить производительность для операций чтения, добавив

дополнительные реплики. Добавление реплик имеет свою цену: в алгоритме, который задействует сильного лидера, оно повлечет за собой увеличение нагрузки на процесс лидера, а при использовании протокола взаимодействия равноправных систем увеличит нагрузку на все процессы. Если же существует резерв производительности для операций записи, но нагрузка, состоящая из таких операций, напрягает систему, добавление реплик может оказаться лучшим выходом.

Следует заметить, что добавление реплики в большинство систем кворумов может потенциально снизить доступность системы (рис. 23.10). Типичное развертывание Zookeeper или Chubby задействует пять реплик, поэтому бо́льшая часть кворумов требует наличия трех. Система все еще будет работать, если две реплики, или 40 %, недоступны. При наличии шести реплик кворум требует наличия четырех действующих: всего 33 % реплик могут быть недоступны, чтобы система продолжала работу.

Ограничения, связанные с областями отказа, становятся еще строже по мере добавления шестой реплики: если организация имеет пять дата-центров и обычно запускает группы консенсуса с пятью процессами, то потеря одного дата-центра все еще оставляет одну свободную реплику в каждой группе. Если шестая реплика развертывается в одном из пяти дата-центров, то его отключение лишает группы обеих свободных реплик, снижая этим производительность на 33 %.

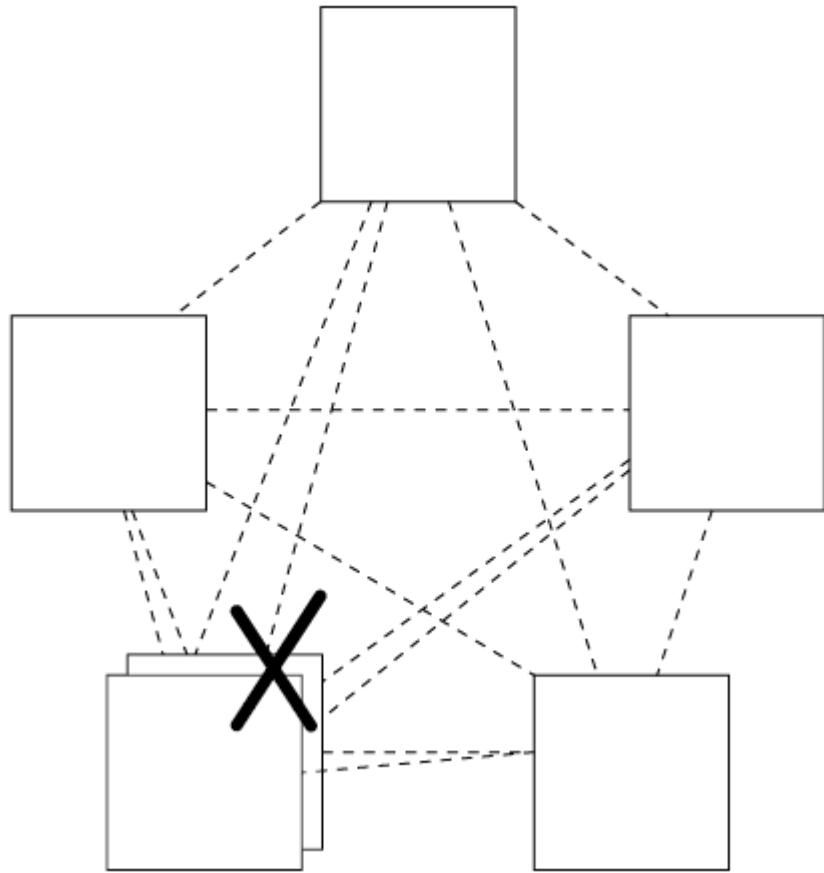


Рис. 23.10. Добавление реплики в одном регионе может уменьшить доступность системы.

Совместное размещение нескольких реплик в одном data-центре может уменьшить доступность системы: при этом существует кворум, у которого нет избыточности

Если клиенты кучно располагаются в одном географическом регионе, лучше всего разместить эти реплики ближе к ним. Однако определение того, где именно их размещать, может потребовать тщательного обдумывания балансировки нагрузки и преодоления перегрузок. Как показано на рис. 23.11, если система просто направляет запросы клиента на чтение ближайшей реплике, то большой скачок нагрузки, сконцентрированный в одном регионе, может перегрузить ближайшую реплику, затем следующую и т.д. — это называется *каскадным сбоем* (см. главу 22). Перегрузки такого типа зачастую возникают в начале выполнения пакетных задач, особенно одновременно нескольких.

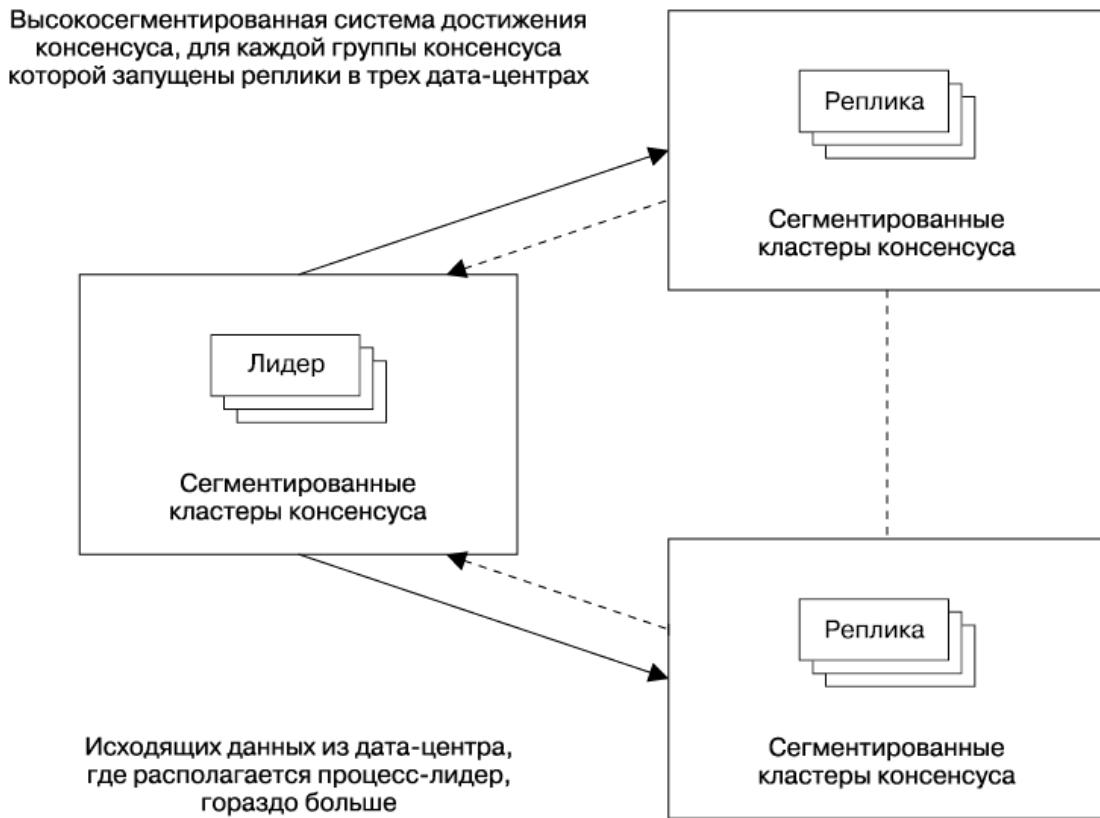


Рис. 23.11. Совместное размещение процессов-лидеров приводит к неравномерному использованию полосы пропускания

Мы уже рассмотрели причину, по которой многие системы, основанные на распределенном консенсусе, используют процесс-лидер для улучшения производительности. Однако важно понимать, что реплики-лидеры будут действовать большее количество вычислительных ресурсов, особенно исходящую производительность сети. Это происходит потому, что лидер отправляет сообщения-предложения, который включают в себя предлагаемые данные, но реплики отправляют более мелкие сообщения, обычно содержащие лишь соглашения с идентификатором транзакции определенного консенсуса. Организации, которые запускают высокосегментированные системы консенсуса с очень большим количеством процессов, могут посчитать необходимым гарантировать, что процессы-лидеры для

разных сегментов относительно равномерно распределены между дата-центрами. Это позволяет предотвратить ситуацию, когда система как единое целое становится узким местом для исходящей производительности сети лишь для одного дата-центра, и позволяет гораздо лучше использовать производительность системы.

Еще одним недостатком размещения групп консенсуса в разных дата-центрах (см. рис. 23.11) является экстремальное изменение системы, которое может произойти, если дата-центр, в котором находятся лидеры, испытает масштабный сбой (например, произойдет отключение питания, поломка сетевого оборудования или будет перерезан провод). При таком сценарии сбоя все лидеры должны переключиться на другой дата-центр, либо разделившись поровну, либо массово направившись в один дата-центр (рис. 23.12). В любом случае узел, расположенный между двумя другими дата-центрами, получит гораздо больше трафика от этой системы. В этот момент меньше всего хочется обнаружить, что производительности этого узла недостаточно.

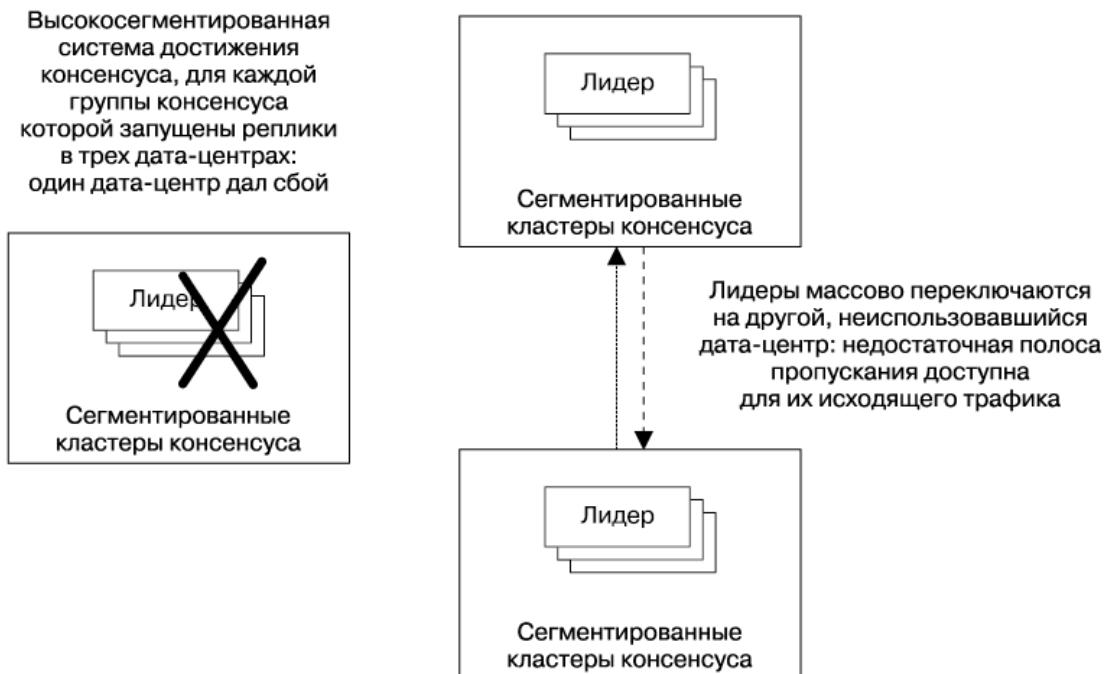


Рис. 23.12. Когда совместно размещенные лидеры массово переключаются на другой дата-центр, шаблоны использования сети могут значительно измениться

Однако такой тип развертывания легко может оказаться непредусмотренным результатом протекающих в системе автоматических процессов, которые влияют на выбор лидеров, например.

- Любые операции, обрабатываемые лидером, будут протекать с меньшей задержкой, если он располагается очень близко к клиентам. Алгоритм, который пытается расположить лидеров рядом с большинством клиентов, может этим воспользоваться.
- Алгоритм может попытаться определить местоположение лидеров на машинах с наилучшей производительностью. Ловушка, связанная с этим подходом, заключается в том, что если один из трех дата-центров имеет более быстрые машины, то ему будет отправлено непропорциональное количество трафика, что приведет к экстремальным изменениям уровня последнего, если дата-центр станет недоступным. Для того чтобы избежать этой проблемы, алгоритмы должны учитывать возможности машин при их выборе.
- Алгоритм выбора лидера может отдавать предпочтение процессам, которые работают дольше. Более долгоиграющие процессы с большей вероятностью будут связаны с местоположением, если релизы ПО будут выполняться для каждого дата-центра.

Композиция кворумов

При определении расположения реплик в группе консенсуса важно учитывать эффект географического распределения (или,

точнее, сетевые задержки между репликами), который будет влиять на производительность группы.

Одним из подходов является максимально равномерное распределение реплик, чтобы RTT для них были примерно равными. Если равными будут и все прочие факторы, такие как нагрузка, аппаратная часть и производительность сети, это даст относительно стабильное значение производительности для всех регионов независимо от того, где находится лидер группы (или для каждого члена группы консенсуса, если используется протокол без лидера).

География может значительно усложнить применение этого подхода. Это особенно верно при сравнении внутриконтинентального трафика и трафика, пересекающего Тихий или Атлантический океан. Рассмотрим систему, которая охватывает Северную Америку и Европу: невозможно расположить реплики равноудаленно друг от друга, поскольку всегда будет существовать увеличение задержки для трансатлантического трафика по сравнению с внутриконтинентальным. Несмотря ни на что, транзакции из одного региона должны будут пересечь океан, для того чтобы достичь консенсуса.

Однако, чтобы попытаться распределить трафик максимально равномерно, проектировщики систем могут выбрать следующий вариант размещения пяти реплик: две в центре Соединенных Штатов, одна на Восточном побережье и две в Европе (рис. 23.13). Такое распределение будет означать, что в типичном случае консенсуса можно достичь в Северной Америке, не дожидаясь ответа реплик из Европы, а если консенсуса нужно достичь из Европы, общаться нужно будет только с репликой, расположенной на побережье. Реплика, размещенная на Восточном побережье, выступает своего рода связующим звеном, где пересекаются два возможных кворума.



Рис. 23.13. Пересечение квorumов, где одна реплика играет роль связующего звена

Потеря этой реплики означает, что задержка системы значительно увеличится: вместо того чтобы сильно зависеть от RTT либо между центром США и Восточным побережьем, либо между Восточным побережьем и Европой, она будет основана на RTT между Европой и центром США (рис. 23.14) и окажется на 50 % выше, чем задержка, основанная на RTT между Европой и Восточным побережьем. Географическая дистанция и RTT между ближайшим возможным квorumом значительно увеличиваются.



Рис. 23.14. Потеря связующей реплики немедленно приводит к повышению RTT для любого кворума

Этот сценарий является основным слабым местом простого кворума большинства, когда он применим к группам, состоящим из реплик, имеющих различающиеся RTT. В таких случаях может оказаться полезным иерархический подход сборки кворума. Десять реплик могут быть развернуты в трех группах по три реплики (рис. 23.15). Кворум может быть сформирован по принципу большинства групп, и группа может быть включена в кворум, если доступна большая часть ее членов. Это означает, что даже потеря реплики центральной группы не повлияет на общую производительность системы, поскольку центральная группа все еще может голосовать за транзакции с помощью двух и трех реплик. Существует, однако, проблема расхода ресурсов, связанная с запуском большего количества реплик. В высокосегментированной системе с нагрузкой, состоящей из операций чтения, которая качественно обрабатывается репликами, мы можем справиться с этой проблемой, используя меньшее количество групп

консенсуса. Такая стратегия означает, что общее количество процессов в системе не изменится.



Рис. 23.15. Иерархические кворумы могут быть применены для снижения зависимости от центральной реплики

Наблюдение за распределенными системами, основанными на консенсусе

Как мы уже видели, алгоритмы достижения распределенного консенсуса лежат в основе большинства критически важных систем компании Google ([Ananatharayan, 2013], [Burrows, 2006], [Corbett, 2012], [Shute, 2013]). Все важные производственные системы нуждаются в наблюдении для того, чтобы обнаруживать отключения или проблемы, а также для поиска проблем. Опыт показывает, что существуют некоторые аспекты распределенных систем, основанных на консенсусе, которым требуется особое внимание. Давайте их рассмотрим.

- *Количество членов, запущенных в каждой группе консенсуса, и состояние каждого процесса (работоспособный или нет).* Процесс может быть запущен, но по какой-то причине, например связанной с аппаратной частью, не способен развиваться.
- *Постоянно отстающие реплики.* Работоспособные члены группы консенсуса могут находиться в разных состояниях. Член группы может находиться в состоянии восстановления, выполняемого с помощью одноранговых узлов после запуска, или отставать от группы кворума, или полноценно участвовать в работе, или быть лидером.
- *Существует лидер или нет.* Необходимо контролировать систему, основанную на алгоритме вроде Multi-Paxos, который берет на себя роль лидера. Важно гарантировать, что лидер существует, поскольку, если у системы нет лидера, она будет недоступна.
- *Количество смен лидера.* Многократная быстрая смена лидера снижает производительность систем, основанных на консенсусе, которые используют стабильного лидера, поэтому нужно следить за количеством смен лидера. Алгоритмы консенсуса зачастую помечают изменение лидерства с помощью нового срока или номера представления, это число является полезным показателем для наблюдения. Слишком быстрое увеличение частоты смены лидера может быть связано с проблемой с сетевым соединением. Снижение частоты может указывать на серьезную ошибку.
- *Номер транзакции консенсуса.* Операторам нужно знать, продвигается ли работа в системе, основанной на консенсусе. Большинство алгоритмов достижения

консенсуса обозначают прогресс, увеличивая номер транзакции консенсуса. Если система работоспособная, этот номер должен увеличиваться с течением времени.

- *Видимое количество предложений; количество предложений, с которыми согласились.* Эти числа показывают, корректно ли работает система.
- *Пропускная способность и латентность.* Несмотря на то что эти характеристики свойственны не только для распределенных систем, администраторы должны следить за ними.

Для того чтобы понять, какова производительность системы, и упростить поиск связанных с ней проблем, вы можете наблюдать:

- за распределением задержки для принятия предложений;
- распределением сетевых задержек, наблюдаемых между частями системы в разных локациях;
- количеством времени, которое приемники потратили на долговечное журналирование;
- общим количеством байтов, полученных системой за секунду.

Итоги главы

Мы рассмотрели задачу распределенного консенсуса и представили некоторые типовые архитектурные решения для систем, основанных на распределенном консенсусе, а также

обсудили вопросы производительности и некоторые эксплуатационные аспекты для таких систем.

В рамках этой главы мы намеренно избегали глубокой дискуссии о конкретных алгоритмах, протоколах и реализациях. Средства координирования распределенных систем и технологии, лежащие в их основе, меняются быстро, и эта информация очень скоро устареет, в отличие от основ, которые мы рассмотрели. А эти основы наряду с упомянутыми в главе статьями позволяют вам применять как инструменты распределенного координирования, доступные сегодня, так и ПО, которое появится в будущем.

Если вы не запомните ничего другого из этой главы, не забывайте хотя бы о проблемах, которые могут быть решены с помощью распределенного консенсуса, и о тех проблемах, которые могут появиться при использовании ситуационных методов управления наподобие контрольных сигналов. Когда вы встретите ситуацию выбора лидера, критическое разделяемое состояние или распределенные блокировки, вспомните о распределенном консенсусе: любой другой подход будет напоминать бомбу с часовым механизмом, готовую разрушить ваши системы.

[143](#) Кайл Кингберри написал большую серию статей о корректности распределенных систем, которая содержит множество примеров неожиданного или некорректного поведения хранилищ данных такого вида. См. <https://aphyr.com/tags/jepsen>.

[144](#) В частности, производительность оригинального алгоритма Paxos неидеальна, но она значительно улучшилась с течением времени.

24. Cron: планирование и расписание в распределенных системах

Автор — Штепан Давидович [145](#)

Под редакцией Кавиты Гулианы

В этой главе описывается реализация компанией Google сервиса cron, обслуживающего большинство внутренних команд, которым нужно периодическое планирование вычислительных задач. На протяжении всего периода существования сервиса cron мы усвоили много уроков, касающихся проектирования и реализации того, что может показаться простым сервисом. Здесь мы обсудим проблемы, с которыми сталкиваются распределенные реализации сервиса cron, и покажем некоторые потенциальные решения.

Cron — это широко распространенная утилита Unix, спроектированная для периодического запуска произвольных пользовательских задач в заданные время или интервалы времени. Сначала мы проанализируем основные принципы cron и ее самые распространенные реализации, далее рассмотрим, как приложение наподобие cron может работать в крупных распределенных средах для того, чтобы повысить надежность систем при сбое одной машины. Мы опишем распределенную систему cron, которая развернута на небольшом количестве машин, но может запускать задачи cron во всем дата-центре вместе с системой планирования для дата-центра, такой как Borg [Verma, 2015].

Cron

Рассмотрим, как сервис cron обычно используется на одной машине, прежде чем рассматривать варианты его запуска как сервиса, который работает для нескольких data-центров.

Введение

Сервис cron спроектирован таким образом, что системные администраторы и обычные пользователи могут указывать необходимые для запуска команды, а также время, когда их нужно запустить. Сервис cron выполняет разные виды работ, включая сборку мусора и периодический анализ данных. Наиболее распространенный формат указания времени называется crontab. Он поддерживает простые интервалы (например, «каждый полдень» или «в начале каждого часа»). Вы также можете задать сложные интервалы, например «каждую субботу, если она приходится на 30-е число месяца».

Сервис cron обычно реализуется с помощью одного компонента, который называют crond. crond — это демон, который загружает список запланированных задач сервиса cron. Эти задачи запускаются в соответствии с заданным временем выполнения.

Надежность

Некоторые аспекты сервиса cron следует рассмотреть с точки зрения надежности.

- Его область отказа, по сути, является одной машиной. Если машина не работает, то ни планировщик, ни задачи, которые он должен запустить, не будут запущены¹⁴⁶. Рассмотрим очень простой случай: есть две машины, для которых планировщик запускает задачи на другой рабочей машине (например, с помощью SSH). Этот сценарий

представляет две отдельные области отказа, которые могут повлиять на способность запускать задачи: могут дать сбой либо машина-планировщик, либо машина-получатель.

- Единственное состояние, которое нужно сохранить при перезапусках crond, включая перезагрузки самой машины, — это конфигурация crontab. Запуски cron работают по принципу «запустил и забыл», и crond даже не пытается отследить их.
- anacron является заметным исключением из этого правила. anacron пытается запускать задачи, которые должны были быть запущены, пока система была отключена. Возможность перезапуска ограничена задачами, которые запускаются раз в день или реже. Эта функциональность очень полезна для запуска заданий по обслуживанию на рабочих станциях и ноутбуках, ее поддерживает файл, в котором сохраняются временные метки последнего запуска для всех зарегистрированных задач cron.

Задачи cron и идемпотентность

Задачи cron разработаны для того, чтобы выполнять повторяющуюся работу, но, помимо этого, трудно знать заранее, какие функции они получат. Разнообразие требований, которое влечет за собой широкий набор задач cron, очевидным образом влияет на требования к надежности.

Некоторые задачи cron, например процессы сборки мусора, являются идемпотентными. В случае сбоя системы будет безопасно запустить их несколько раз. Другие задачи cron, например процессы, которые отправляют новости по электронной почте большому кругу получателей, не должны запускаться больше одного раза.

Ситуацию усложняет тот, что для одних задач cron сбой запуска приемлем, а для других — нет. Например, запланировано выполнять задачу по сборке мусора каждые пять минут, и один запуск можно пропустить, но задача cron, отвечающая за начисление зарплаты и запускающаяся раз месяц, не может быть пропущена.

Такое разнообразие задач cron затрудняет рассуждения о типах сбоев: системы вроде cron не дают единого ответа, который подошел бы для всех ситуаций. В общем случае мы предпочитаем пропускать запуски, чем рисковать, запуская задачу дважды, насколько это позволяет инфраструктура. Мы выбираем такой вариант, потому что восстановиться после пропущенного запуска проще, чем после двойного. Владельцы задач cron могут (и должны!) наблюдать за своими задачами. Например, владелец может заставить сервис cron показывать состояние управляемых им задач или настроить независимое наблюдение за воздействием, которое оказывают эти задачи. В случае пропущенного запуска владельцы задач cron могут действовать в соответствии с природой задачи cron. Однако ликвидировать последствия двойного запуска, например, упомянутой рассылки новостей может быть трудно или невозможно. Поэтому мы предпочитаем «закрываться при отказе», для того чтобы избежать систематического возникновения плохого состояния.

Масштабирование Cron

Переход от отдельных компьютеров к крупномасштабным развертываниям требует фундаментального пересмотра способа заставить cron хорошо работать в такой среде.

Перед тем как представить детали решения компании Google, мы обсудим различия между развертываниями в малом

и крупном масштабе, а также опишем, как нужно изменить проект для проведения крупномасштабных развертываний.

Расширенная инфраструктура

В своих обычных реализациях сервис cron ограничен одной машиной. Крупномасштабные развертывания распространяют решение на несколько машин.

Размещение сервиса cron на одной машине может быть катастрофическим с точки зрения надежности. Предположим, что такая машина находится в data-центре, содержащем 1000 машин. Сбой всего 1/1000-й из доступного парка отключит весь сервис cron. По очевидным причинам такая реализация неприемлема.

Для того чтобы повысить надежность сервиса cron, мы отвязываем процессы от машин. Если вы хотите запустить сервис, просто укажите требования к нему и data-центр, в котором он должен быть запущен. Система планирования для data-центров (которая должна быть надежной сама по себе) определяет машину или машины, на которых следует развернуть сервис, а вдобавок обрабатывает «падения» машин. Запуск задачи в data-центре, по сути, станет отправкой одной или нескольких RPC планировщику data-центра.

Однако этот процесс нельзя назвать мгновенным. Определение мертвых машин влечет за собой тайм-ауты для проверки состояния, а перевод сервиса на другую машину потребует времени для установки ПО и запуска нового процесса.

Поскольку перемещение процесса на другую машину зачастую означает потерю состояния, хранимого локально на старой машине (если только не используется живая миграция), а время на перепланирование может превысить минимальный

интервал, равный 1 минуте, нужны процедуры для того, чтобы смягчить как негативные последствия потери данных, так и повышенные требования к времени. Вы можете при необходимости сохранить состояние старых машин в распределенной файловой системе наподобие GFS, а затем использовать ее во время запуска для того, чтобы определить задачи, давшие сбой из-за перепланирования. Однако такое решение не оправдывает ожиданий, связанных со своевременностью: если вы будете запускать задачу cron каждые 5 минут, задержка 1 или 2 минуты, вызванная общей загрузкой системы перепланирования, потенциально может быть неприемлемо весомой. В таком случае горячий резерв, который способен быстро включиться и возобновить работу, может значительно сократить это временное окно.

Расширенные требования

В одномашинных системах все работающие процессы обычно изолируются лишь в ограниченной степени. Несмотря на то что контейнеры сейчас широко распространены, нет необходимости использовать их для изоляции каждого компонента сервиса, который развернут на одной машине. Поэтому, если cron был развернут на одной машине, crond и все задачи cron, которые он запускает, скорее всего, не будут изолированы.

Развертывание в масштабе дата-центра обычно означает развертывание в контейнеры с принудительной изоляцией. Изоляция в этом случае необходима, поскольку все базируется на том, что независимые процессы, запущенные в одном дата-центре, не должны негативно влиять друг на друга. Для того чтобы выполнить это, вы должны знать количество ресурсов, которые понадобятся для того, чтобы запустить любой желаемый процесс, как для системы cron, так и для задач,

которые она запускает. Задача cron может быть отложена, если ресурсов дата-центра недостаточно для того, чтобы соответствовать ее запросам. Требования к ресурсам, в дополнение к спросу пользователей на наблюдение за запусками задач cron, означают, что нужно отслеживать состояние запусков задач cron в целом, начиная от запланированного запуска до их отключения.

Отвязывание запуска процессов от конкретной машины грозит системе cron частичным сбоем при запуске. Гибкость конфигурации задач cron также означает, что для запуска новой задачи cron в дата-центре может понадобиться несколько RPC, таким образом, иногда мы сталкиваемся со сценарием, когда некоторые RPC выполняются успешно, а некоторые — нет, например, из-за того, что процесс, отправляющий RPC, «умер» во время выполнения этих задач. Процедура восстановления cron также должна учитывать этот сценарий.

С точки зрения типов сбоев дата-центр является значительно более крупной экосистемой, чем одна машина. Сервис cron, который начал свой путь как относительно простой бинарный файл, при развертывании в больших масштабах имеет множество очевидных и неочевидных зависимостей. Мы хотим гарантировать, что, даже если дата-центр даст частичный сбой (например, из-за отключения питания или проблем с сервисами хранилища), сервис, такой же простой, как cron, все еще сможет функционировать. Требуя, чтобы планировщик дата-центра располагал реплики сервиса cron в разных локациях внутри дата-центра, мы избегаем сценария, при котором один элемент распределения питания даст сбой и это уничтожит все процессы сервиса cron.

Можно развернуть один сервис cron для всего мира, но развертывание сервиса внутри одного дата-центра имеет свои

преимущества: сервис насладится малой задержкой и разделит судьбу с планировщиком дата-центра, от которого он в основном зависит.

Создание cron в Google

В этом разделе рассматриваются проблемы, которые должны быть решены для того, чтобы надежно развернуть крупномасштабный сервис cron. Также в нем подчеркиваются некоторые важные решения, сделанные для распределенного сервиса cron в компании Google

Отслеживание состояния задач cron

Как говорилось в предыдущих разделах, нам нужно удерживать какую-то информацию о состояниях задач cron и иметь возможность восстановить ее в случае сбоя. Более того, устойчивость этого состояния чрезвычайно важна. Вспомните: многие задачи cron наподобие расчета зарплаты или отправки новостной рассылки, не являются идемпотентными.

У нас есть два способа отслеживания задач cron:

- хранение данных снаружи в общедоступных распределенных хранилищах;
- использование системы, которая сохраняет небольшое количество информации о состоянии внутри самого сервиса cron.

При проектировании распределенного сервиса cron мы выбрали второй вариант. Остановились на нем по нескольким причинам.

- Распределенные файловые системы наподобие GFS или HDFS часто поддерживают очень большие файлы (например, выходных данных веб-краулеров), в то время как информация о задачах cron, которую нужно сохранить, незначительного размера. Небольшие операции записи для распределенных файловых систем выполнять очень дорого. Также нам придется столкнуться с большой задержкой, поскольку файловая система не оптимизирована для выполнения подобных операций записи.
- Базовые сервисы, на которые отключения будут серьезно влиять, например cron, должны иметь очень мало зависимостей. Даже если мы потеряем какую-то часть data-центра, сервис cron должен функционировать еще какое-то время. Но это требование не означает, что хранилище должно стать непосредственной частью процесса cron (работа с хранилищем — это уже детали реализации). Однако сервис cron должен иметь возможность работать независимо от систем, расположенных далее в технологической цепочке, которые обслуживают большое количество внутренних пользователей.

Использование Paxos

Мы развертываем несколько реплик сервиса cron и используем алгоритм достижения распределенного консенсуса Paxos (см. главу 23), для того чтобы убедиться, что они устойчивы. До тех пор пока большая часть групп доступна, распределенная система может обрабатывать как единое целое изменения состояния, несмотря на природу связанных подмножеств инфраструктуры.

Распределенная система cron использует одну задачу-лидера, которая является единственной репликой, способной

модифицировать общее состояние, а также запускать задачи cron (рис. 24.1). Мы воспользовались тем, что наш вариант протокола Paxos, Fast Paxos [Lamport, 2006], использует реплику-лидер для оптимизации — лидер-реплика протокола Fast Paxos выступает также в роли лидера сервиса cron.

Если реплика-лидер погибает, то механизм, проверяющий состояние группы Paxos, быстро определяет это — в течение нескольких секунд. Поскольку другой процесс cron уже запущен и доступен, можно выбрать нового лидера. Как только новый лидер выбран, мы следуем протоколу выбора лидера, характерному для сервиса cron, который отвечает за выполнение всей работы, оставленной предыдущим лидером. Лидер для сервиса cron является лидером также для протокола Paxos, но сервису cron нужно выполнить дополнительные действия. Высокая скорость реакции при перевыборе лидера позволяет оставаться в рамках приемлемого времени перехода, равного 1 минуте.

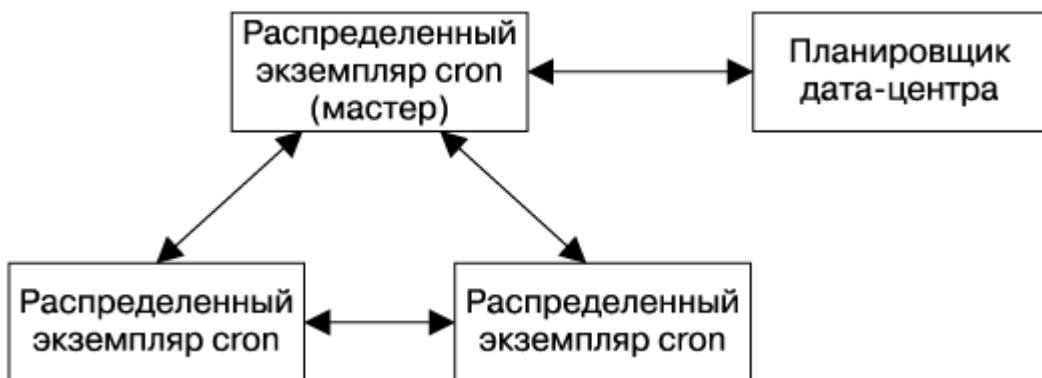


Рис. 24.1. Взаимодействия между распределенными репликами сервиса cron

Наиболее важное состояние, которое мы храним с помощью протокола Paxos, — это информация о том, какие задачи cron сейчас запущены. Мы синхронно информируем кворум реплик о начале и завершении каждой задачи cron.

Роли лидера и последователя

Как было описано ранее, протокол Paxos и его развертывания в сервисе cron имеют две роли: лидера и ведомого. В следующих разделах описана каждая из них.

Лидер

Реплика-лидер — единственная, которая активно запускает задачи cron. Лидер имеет внутренний планировщик, который, как и простой демон crond, описанный в начале этой главы, поддерживает список задач cron, упорядоченный по времени их запуска.

Реплика-лидер ожидает наступления времени запуска первой задачи. Когда это произойдет, она объявляет, что готова запустить задачу, и определяет новое время запуска, как сделала бы обычная реализация демона crond. Конечно же, как и в случае с обычным демоном crond, спецификация запуска задачи cron могла измениться с момента последнего запуска, поэтому она должна быть синхронизирована с последователями. Простой идентификации задачи cron недостаточно, мы также должны уникально идентифицировать конкретный запуск, используя время, в которое он выполняется, в противном случае при запуске задачи cron вы можете столкнуться с неоднозначностью. (Возникновение неоднозначности особенно вероятно в случае, если задачи стартуют каждую минуту.) Такое взаимодействие осуществляется с помощью протокола Paxos (рис. 24.2).

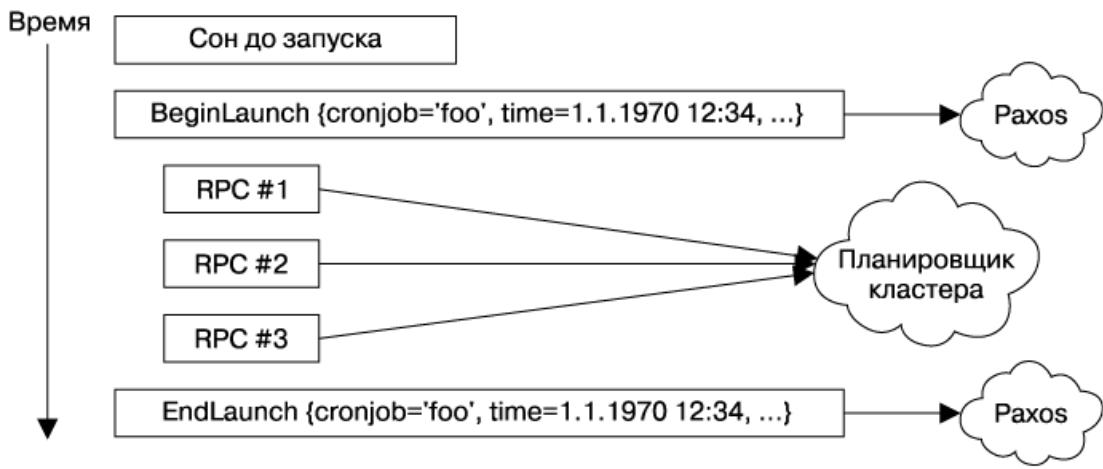


Рис. 24.2. Иллюстрация процесса запуска задачи cron с точки зрения лидера

Важно, чтобы взаимодействие, обеспечиваемое протоколом Paxos, оставалось синхронным и реальные запуски задач cron не происходили до того, как кворум протокола Paxos получит уведомление о запуске. Сервис cron должен понять, запущена ли каждая задача cron, для того чтобы определиться с действиями на случай перехода лидера на другой ресурс. Выполнение этой задачи асинхронно означает, что запуск задачи cron произойдет на лидере и об этом не будут знать другие реплики. В случае перехода реплики-последователи могут попробовать вновь выполнить тот же запуск, поскольку они не знают, что он уже состоялся.

О завершении запуска задачи cron другим репликам объявляется синхронно с помощью протокола Paxos. Обратите внимание на то, что неважно, насколько успешным будет этот запуск из-за внешних причин, например при недоступности планировщика дата-центра. Здесь мы просто отслеживаем тот факт, что сервис cron попытался выполнить запуск в запланированное время. Также нам нужно разрешить сбои системы cron в середине этой операции, что будет рассмотрено в следующем разделе.

Еще одна очень важная особенность лидера заключается в том, что как только он по какой-то причине теряет лидерство, то должен немедленно перестать взаимодействовать с планировщиком дата-центра. Удержание лидерства должно гарантировать взаимное исключение доступа к планировщику дата-центра. В отсутствие этого условия взаимного исключения старый и новый лидеры могут выполнять конфликтующие между собой действия на планировщике дата-центра.

Последователь

Реплики-последователи отслеживают состояние среды выполнения, предоставленное лидером, для того чтобы в любой момент взять управление на себя, если это необходимо. Все изменения состояния, отслеживаемые репликами-последователями, они получают с помощью протокола Paxos от реплики-лидера. Как и лидер, последователи должны поддерживать список всех задач cron системы, и этот список должен быть одинаковым во всех репликах (это достигается с помощью протокола Paxos).

При получении уведомления о произведенном запуске реплика-последователь обновляет локальное время следующего запуска для заданной задачи cron. Это очень важное изменение состояния, выполняемое синхронно, гарантирует, что все запланированные задачи cron внутри системы будут устойчивыми. Мы отслеживаем все открытые запуски (начатые, но не завершенные).

Если реплика-лидер погибает или работает неверно, например отсоединяется от других реплик сети, один из последователей должен быть выбран новым лидером. Выбор должен быть выполнен быстрее чем за 1 минуту, для того чтобы избежать риска пропустить или отложить запуск новых

задач. Как только лидер будет выбран, все открытые запуски (то есть частичные сбои) должны быть завершены. Этот процесс может быть довольно сложным, предъявляющим дополнительные требования как к системе cron, так и к инфраструктуре дата-центра. В следующем разделе рассматривается разрешение подобных частичных сбоев.

Разрешение частичных сбоев

Как упоминалось ранее, взаимодействие между репликой-лидером и планировщиком дата-центра может дать сбой в процессе отправки нескольких RPC, которые описывают один логический запуск задач cron. Наши системы должны иметь возможность отследить этот состояния. Вспомним, что каждая задача cron имеет две точки синхронизации:

- когда мы готовы выполнить запуск;
- когда завершили запуск.

Эти две точки позволяют установить границы для запуска. Если запуск состоит из одной RPC, то как мы узнаем, что она действительно была отправлена? Рассмотрим случай, когда мы знаем, что запланированный запуск был начат, но мы не были оповещены о том, что он завершился перед тем, как реплика погибла.

Для того чтобы определить, была ли отправлена RPC, нужно, чтобы соблюдалось одно из следующих условий.

- Все операции для внешних систем, которые может понадобиться продолжить после перевыбора лидера, должны быть идемпотентными, то есть их можно безопасно выполнить снова.

- Должна существовать возможность выполнить поиск состояния всех операций для внешних систем, чтобы однозначно определить, завершились ли они.

Каждое из этих условий накладывает значительные ограничения, и их может быть трудно реализовать, но возможность соответствовать хотя бы одному из этих условий лежит в основе исправной работы сервиса cron в распределенной среде, в которой могут произойти один или несколько частичных сбоев. Недостаточно качественная обработка этих условий может привести к пропущенным или двойным вызовам задачи.

Большая часть инфраструктуры, которая запускает логические задачи в дата-центрах, например Mesos, предоставляет функциональность для именования этих задач, что позволяет выполнять поиск состояния задач, останавливать их или реализовывать другие виды обслуживания. Разумное решение проблемы идемпотентности заключается в предварительном создании имен задач (тем самым можно избежать создания мутирующих операций в планировщике дата-центра), а затем их распространении по всем репликам сервиса cron. Если лидер сервиса cron погибнет во время запуска, новый лидер просто выполнит поиск состояния всех заранее вычисленных имен и запустит задачи с недостающими именами.

Обратите внимание на следующий момент: как и при определении отдельных запусков задач cron с помощью их имени и времени запуска, важно, чтобы созданные имена задач содержали в планировщике дата-центра определенное время запуска или чтобы эту информацию можно было получить другим способом. В режиме плановой эксплуатации

при сбое лидера сервис cron должен выполнять переход быстро, но так получается не всегда.

Давайте вспомним, что мы отслеживаем запланированное время запуска, когда сохраняем внутреннее состояние между репликами. Аналогично следует снимать неоднозначность взаимодействия с планировщиком дата-центра также с помощью запланированного времени запуска. Например, рассмотрим недолговечную, но часто запускаемую задачу cron. Задача cron запускается, но до того, как информация об этом отсылается всем репликам, лидер дает сбой и необычно долго выполняет переход — достаточно долго для того, чтобы задача успешно завершилась. Новый лидер выполняет поиск состояния этой задачи, видит, что она завершилась, и пытается выполнить ее снова. Если бы он знал о времени запуска, то понял бы, что задача в планировщике дата-центра появилась из-за этого вызова, и в таком случае двойного запуска не было бы.

Реальная реализация имеет более сложную систему поиска состояния, которой управляют детали реализации лежащей в ее основе инфраструктуры. Однако в приведенном здесь описании рассматриваются независимые от реализации требования к любой подобной системе. В зависимости от того, какая инфраструктура вам доступна, может понадобиться рассмотреть также возможность компромисса между риском двойного запуска и риском незапуска.

Сохраняем состояние

Использование протокола Paxos для достижения консенсуса — это только часть задачи сохранения состояния. Paxos, по сути, является журналом, хранящим изменения состояния, запись в который производится синхронно с выполнением изменений. Из этой характеристики протокола Paxos вытекает следующее.

- Чтобы журнал не разросся до бесконечности, он должен быть сжат.
- Журнал нужно где-то хранить.

Для того чтобы предотвратить бесконечное разрастание журнала Paxos, можно просто сделать снимок текущего состояния, что позволит восстановить его, не воспроизводя все изменения, записанные в журнале. Например, если в журнале сохранено изменение состояния «увеличить счетчик на 1», то после 1000 итераций у там будет 1000 записей, которые можно заменить на снимок «установить значение счетчика равным 1000».

Если журнал будет утрачен, мы потеряем лишь те изменения состояния, которые произошли после создания последнего снимка. Снимки фактически являются наиболее критической сущностью — если мы лишимся снимков, то, по сути, придется начинать с нуля, поскольку будет утрачено внутреннее состояние. В то же время потеря журналов просто вызовет утрату состояния и отправит систему cron назад во времени, к точке, когда был сделан последний снимок.

Есть два основных варианта хранения данных:

- во внешнем распределенном хранилище, к которому имеется свободный доступ;
- в системе, которая сохраняет небольшую часть информации о состоянии как часть самого сервиса cron.

При проектировании системы мы объединяем элементы обоих вариантов.

Мы храним журналы протокола Paxos на локальном диске машины, где запланированы реплики сервиса cron. Наличие трех реплик при плановой эксплуатации подразумевает, что у нас будут три копии журналов. Мы также храним эти снимки на локальном диске. Но, поскольку они являются критически важными, мы храним их резервную копию в распределенной файловой системе, защищая себя от сбоев, влияющих на все три машины.

Мы не храним журналы в распределенной файловой системе. Мы приняли осознанное решение, которое заключается в том, что потеря журналов, представляющих небольшой объем самых недавних изменений состояния, — это приемлемый риск. Хранение журналов в распределенной файловой системе может повлечь за собой значительное снижение производительности, вызванное выполнением множества небольших операций записи. Одновременная потеря всех трех машин маловероятна, а если это все-таки случится, данные будут автоматически восстановлены с помощью снимка. Таким образом, мы теряем лишь небольшое количество журналов — записи, сделанные с момента последнего снимка, которые выполняются через конфигурируемые промежутки времени. Конечно, эти компромиссы могут различаться в зависимости от деталей инфраструктуры, как и требования, предъявляемые к системе cron.

В дополнение к тому, что журналы и снимки хранятся на локальном диске, а резервные копии снимков — в распределенной файловой системе, только что запущенная реплика может получить снимок состояния и все журналы от уже работающей реплики по сети. Эта способность позволяет сделать запуск реплики независимым от состояния локальной машины. Поэтому переназначение реплики на другую машину

после перезапуска (или «смерти» машины), по сути, не влияет на надежность сервиса.

Запуск большого экземпляра Cron

Существуют и более простые, но такие же интересные способы применения запуска крупного развертывания сервиса cron. Традиционный сервис cron имеет небольшую емкость: обычно он содержит не более нескольких десятков задач cron. Но если вы запустите сервис cron для тысяч машин дата-центра, нагрузка на него увеличится и вы можете столкнуться с проблемами.

Остерегайтесь серьезной и хорошо известной проблемы распределенных систем, которая называется «шумная толпа». Основываясь на конфигурации пользователя, сервис cron может вызвать значительные скачки загруженности дата-центра. Когда люди думают о ежедневно выполняемой задаче cron, они обычно конфигурируют ее так, чтобы она выполнялась в полночь. Это сработает, если задача будет запущена на той же машине, но что, если задача cron может запускать задачу MapReduce с тысячей процессов-исполнителей? И что, если 30 разных команд решат запустить таким образом задачи cron в одном дата-центре? Для того чтобы решить эту проблему, мы ввели расширение для формата crontab.

В обычном crontab пользователи указывают минуту, час, день месяца (или недели) и месяц, в которые должна быть выполнена задача, или звездочку, если может быть использовано любое значение. Для запуска задачи каждый день в полночь следует использовать нотацию «0 0 * * *» (то есть нулевая минута нулевого часа каждого дня недели каждого месяца и каждый день недели). Мы также используем знак вопроса, который означает, что приемлемо любое значение и

система cron сама может его выбирать. Пользователи задают это значение путем хеширования конфигурации задачи cron в заданном временном промежутке (например, 0...23 для часов), что позволяет распределить запуски более равномерно.

Несмотря на это изменение, загрузка, вызванная задачами cron, все еще может сильно скакать. График, показанный на рис. 24.3, иллюстрирует среднее глобальное количество запусков задач cron в Google. Он подчеркивает частые пики запусков задач cron, которые зачастую бывают вызваны задачами cron, которые должны быть запущены в определенное время, например, из-за временной зависимости или внешних событий.

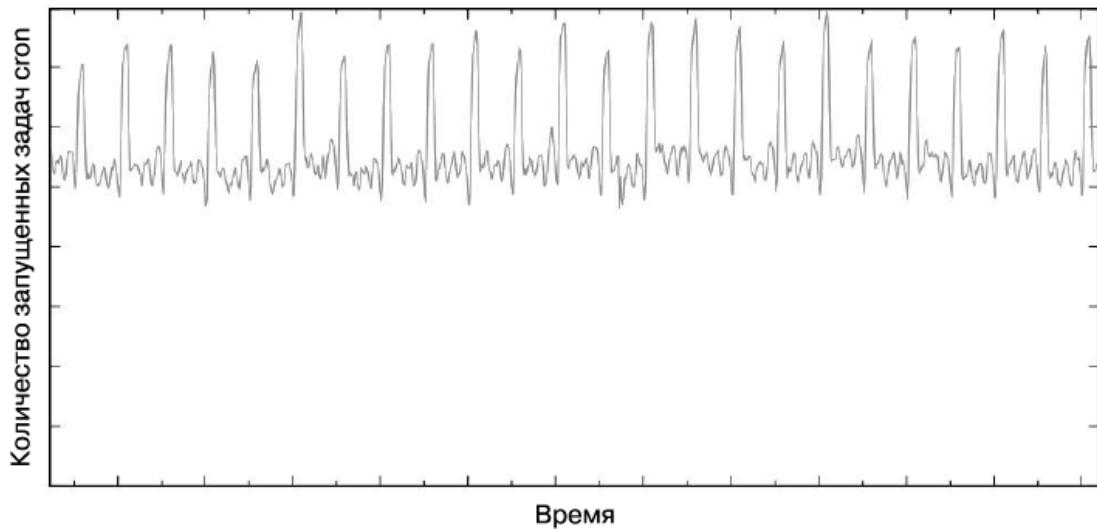


Рис. 24.3. Количество задач cron, запущенных глобально

Итоги главы

Наличие сервиса cron — фундаментальная особенность UNIX-систем на протяжении многих десятилетий.

Однако отрасль перешла на использование крупных распределенных систем, где минимальным рассматриваемым аппаратным блоком может считаться дата-центр, и это требует

изменений в большой части стека. Система cron не стала исключением. В основе нового дизайна Google лежит тщательное изучение необходимых свойств сервиса cron и требований задач cron.

Мы рассмотрели новые ограничения, связанные с распределенной средой, и возможный проект сервиса cron, основанный на решении Google. Это решение требует гарантированной целостности и устойчивости в распределенной среде. Поэтому ядром распределенной реализации cron является Paxos — широко распространенный алгоритм достижения консенсуса в ненадежных средах. Применение Paxos и корректный анализ новых типов сбоев задач cron в крупномасштабных распределенных средах позволили создать устойчивый сервис cron, широко используемый в компании Google.

[145](#) Эта глава была ранее частично опубликована в ACM Queue (март 2015 года, том 13, выпуск 3).

[146](#) Сбои отдельных задач лежат за пределами этого анализа.

25. Конвейеры обработки данных

Автор — Дэн Дэннисон

Под редакцией Тима Харви

В этой главе рассматриваются сложности, с которыми вам придется столкнуться при управлении сложными конвейерами обработки данных. В ней рассказывается, когда нужно использовать циклические конвейеры, которые запускаются очень редко, а когда — непрерывные, которые никогда не останавливаются. Поговорим также о том, какие прерывания могут приводить к заметным операционным проблемам. Свежий взгляд на модель лидера — последователя позволяет представить ее как более надежную и качественно масштабируемую альтернативу циклическим конвейерам для обработки больших данных.

Происхождение шаблона проектирования «Конвейер»

Классический подход к обработке данных заключается в том, чтобы написать программу, которая считывает данные, преобразует их в желаемую форму и выводит новые данные. Обычно такая программа должна работать под контролем циклической программы планирования наподобие cron.

Такой шаблон проектирования называется *конвейером обработки данных*. Конвейерами можно считать сопрограммы [Conway, 1963], коммуникационные файлы DTSS [Bull, 1980], конвейеры UNIX [McIlroy, 1986] ETL^{[147](#)}, но внимание к ним возросло с появлением больших данных, или «наборов данных настолько больших и сложных, что традиционные приложения для обработки данных для них не подходят»^{[148](#)}.

Основной эффект, который большие данные оказывают на шаблон простого конвейера

Программы, которые выполняют периодические или продолжительные преобразования больших данных, обычно называют *простыми однофазными конвейерами*.

Учитывая масштаб и сложность обработки, связанные с большими данными, программы обычно организуют в цепочки, где выходные данные одной программы становятся входными данными для следующей. Такую компоновку можно обосновывать по-разному, но, как правило, она разрабатывается для простоты рассмотрения системы и не предназначена для повышения операционной эффективности. Программы, организованные таким образом, называются *мультифазными конвейерами*, поскольку каждая программа в цепочке действует как отдельный этап обработки данных.

Количество программ, объединенных в серию, — это показатель, который известен как *глубина конвейера*. Неглубокий конвейер может иметь только одну программу, его глубина будет равна единице. Глубокий же конвейер может иметь глубину, равную десяткам или сотням.

Сложности, связанные с шаблоном циклического конвейера

Циклические конвейеры обычно стабильны, когда имеют достаточное количество работников для указанного объема данных и спрос на вычислительную работу не выходит за рамки их возможностей. Вдобавок неустойчивости наподобие узких мест можно избежать, когда количество связанных работ и относительная полоса пропускания между задачами остаются постоянными.

Циклические конвейеры полезны и удобны, и в компании Google их задействуют довольно регулярно. Они написаны с помощью фреймворков вроде MapReduce [Dean, 2004], Flume [Chambers, 2010] и др.

Однако коллективный опыт SR-инженеров показал, что модель циклических конвейеров довольно нестабильна. Мы обнаружили, что, когда для такого конвейера задаются параметры, такие как периодичность, прием разбиения данных и др., производительность поначалу не снижается. Однако естественный рост и изменения неизбежно приводят к напряжению в системе, и появляются проблемы, например истечение дедлайна для задач, истощение ресурсов и зависшая обработка фрагментов данных, которые влекут за собой соответствующую операционную нагрузку.

Проблемы, вызванные неравномерным распределением работы

Основным прорывом модели больших данных было широкое распространение «массово параллельных» [Moler, 1986] алгоритмов, позволяющих разбить большую нагрузку на фрагменты такого размера, которые могут уместиться на отдельных машинах. Иногда эти фрагменты требовали неодинакового объема ресурсов, и далеко не всегда сразу можно было понять, почему некоторые фрагменты требуют большего их количества. Например, при загрузке, фрагментированной по клиентам, фрагменты данных для одних клиентов могут быть гораздо крупнее, чем для других. Поскольку клиент является неделимым объектом, общее время работы равно времени работы самого крупного клиента.

Проблема «висящих фрагментов» может привести к тому, что ресурсы будут распределяться на основе разницы между

машинами в кластере или отводиться для заданий с избытком. Она возникает из-за трудностей, связанных с операциями в реальном времени для потоков наподобие сортировки данных. Типичное поведение клиентского кода заключается в том, чтобы подождать, пока не завершатся все вычисления, и только потом переходить на следующий этап конвейера, в основном потому, что может быть использована сортировка, для выполнения которой потребуются все данные. Это может значительно задержать работу конвейера, поскольку все задачи не будут завершены до тех пор, пока не завершится самая медленная из них, что продиктовано используемой технологией разбиения на фрагменты.

Если эта проблема обнаружена инженерами или инфраструктурой наблюдения за кластером, реакция на нее может еще сильнее ухудшить ситуацию. Например, ответ по умолчанию на эту проблему мгновенно завершит задачу и позволит ей перезапуститься, поскольку блокировка может оказаться результатом влияния недетерминированных факторов. Но поскольку реализация конвейера по умолчанию не предполагает наличия контрольных точек, работа для всех фрагментов будет возобновлена, что приведет к потере времени, циклов процессора и человеческих усилий, приложенных на предыдущем цикле.

Недостатки циклических конвейеров в распределенных средах

Циклические конвейеры, работающие с большими данными, широко используются в Google, поэтому наше решение по управлению кластерами включает в себя альтернативный механизм планирования для таких конвейеров. Этот механизм необходим, поскольку, в отличие от непрерывных конвейеров,

циклические обычно запускают пакетные задачи с низким приоритетом. Назначения с низким приоритетом работают хорошо, так как в этом случае пакетная работа не будет чувствительной к задержкам настолько, насколько чувствительны веб-сервисы. В дополнение к этому, для того чтобы контролировать затраты путем максимизации нагрузки машин, Borg (система управления кластерами Google [Verma, 2015]) дает указание доступным машинам выполнять пакетную работу. Такой приоритет может вызвать увеличение задержки при запуске, поэтому задачи конвейера, запускаясь, могут испытывать постоянные задержки.

Задачи, вызванные с помощью этого механизма, имеют несколько естественных ограничений, из-за чего начинают вести себя по-разному. Например, на задачи, выполнение которых запланировано в промежутках между выполнением задач пользовательских веб-сервисов, могут влиять доступность ресурсов с малой задержкой, цена и стабильность доступа к ресурсам. Стоимость выполнения обратно пропорциональна запрошенной задержке запуска и прямо пропорциональна количеству потребленных ресурсов. Несмотря на то что пакетное планирование на практике может работать гладко, избыточное использование пакетного планировщика (см. главу 24) подвергает задачи риску откачки (см. [Verma, 2015, раздел 2.5]), когда нагрузка кластера высока из-за того, что пользователям не хватает пакетных ресурсов. В свете риска появления компромиссов успешный запуск циклического конвейера представляет собой тонкий баланс между высокой стоимостью ресурсов и риском откачки.

Задержки продолжительностью до нескольких часов могут оказаться приемлемыми для конвейеров, которые запускаются раз в день. Однако по мере увеличения частоты запланированных запусков минимальное время между

выполнениями может быстро достичь минимального среднего времени ожидания. Тогда следует установить нижнюю границу задержки, которую может позволить циклический конвейер. Интервал выполнения задач ниже этой границы приводит к нежелательному поведению, а не помогает продвинуться в выполнении задачи. Конкретный тип сбоя зависит от применяемой политики пакетного планирования. Например, каждый новый запуск может накладываться на другие в планировщике кластера, поскольку предыдущий запуск еще не завершен. Что еще хуже, выполняющийся в данный момент и практически завершенный запуск может быть прерван в момент, когда должна запускаться следующая задача.

На рис. 25.1 показана точка, в которой пересекаются стремящаяся вниз линия интервала простоя и запланированная задержка. В этом сценарии интервал выполнения менее 40 минут для этой примерно 20-минутной работы может привести к появлению параллелизма и, соответственно, к нежелательным последствиям.

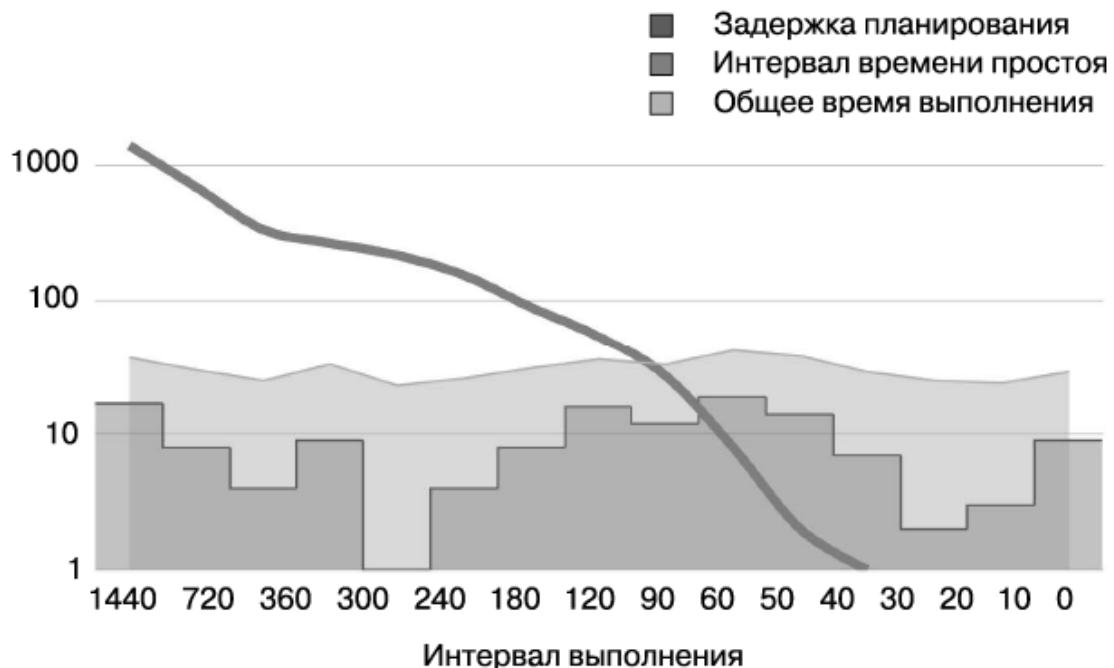


Рис. 25.1. Интервал запуска циклического конвейера и время простоя (логарифмическая шкала)

Для решения этой проблемы нужно зарезервировать достаточную производительность сервера, чтобы иметь возможность выполнять работу. Однако получение ресурсов в разделяемой распределенной среде зависит от предложения и спроса. Ожидается, что команды разработчиков будут отказываться проходить через процесс получения ресурсов, когда ресурсы должны будут оказываться в общем пуле и становиться совместными. Таким образом, нужно разграничить ресурсы для потокового планирования и ресурсы, необходимые для производственной системы, чтобы рационализировать затраты на их получение.

Наблюдение за проблемами в циклических конвейерах

Для конвейеров, довольно долго выполняющих свою работу, наличие поступающей в режиме реального времени информации о показателях производительности среды выполнения может оказаться таким же важным (если не важнее), как и знание общих показателей. Это происходит потому, что данные в реальном времени важны для предоставления операционной поддержки, которая включает в себя и реагирование на чрезвычайные ситуации. На практике стандартная модель наблюдения включает в себя сбор данных по всем показателям во время выполнения задач и отправку этих данных только по завершении работы. Если ни одна задача не даст сбой во время выполнения, статистики мы не получим.

У непрерывных конвейеров нет таких проблем, поскольку их задачи выполняются постоянно и их телеметрия спроектирована таким образом, чтобы показатели в реальном

времени были доступны. Циклические конвейеры не должны иметь характерных проблем с наблюдением.

Проблемы «шумной толпы»

В дополнение к вызовам, создаваемым выполнением и наблюдением, существует проблема «шумной толпы», характерная для распределенных систем, которая также рассматривается в главе 24. Возьмем довольно большой циклический конвейер, в каждом цикле которого тысячи модулей могут начать действовать одновременно. Если их количество слишком велико, так как неверно сконфигурировано или обусловлено некорректной логикой выполнения повторных попыток, то серверы, на которых они работают, будут перегружены, как и лежащие в их основе разделяемые сервисы кластеров и любая использованная сетевая инфраструктура.

Еще сильнее ухудшает эту ситуацию то, что, если логика выполнения повторов не реализована, проблемы с корректностью могут привести к тому, что при сбоях задача не будет выполнена повторно. Если логика выполнения повторных попыток прослеживается, но реализована примитивно или плохо, повторные попытки могут ухудшить проблему.

Вмешательство человека также способно повлиять на этот сценарий. Инженеры с небольшим опытом управления конвейерами, как правило, усугубляют проблему, добавляя в конвейер большее количество модулей, когда задача не выполняется за желаемый промежуток времени.

Независимо от источника проблемы «шумной толпы», для инфраструктуры кластера и SR-инженеров, отвечающих за

расположенные в нем различные сервисы, нет ничего хуже, чем 10 000 задач конвейера, работающих некорректно.

Интерференция нагрузки

Иногда проблема «шумной толпы» обнаруживается не сама по себе. Может возникнуть связанная с ней проблема, которую мы называем «интерференцией нагрузки» (Moire load pattern). Она проявляется в тех случаях, когда два или более конвейера работают одновременно и последовательности выполняемых ими операций случайно накладываются друг на друга, приводя к потреблению одних и тех же разделяемых ресурсов. Эта проблема может возникать даже в непрерывно действующих конвейерах, однако там она менее характерна, поскольку нагрузка поступает более равномерно.

Интерференция нагрузки больше всего проявляется в сценариях, где конвейеры задействуют общие ресурсы. Например, на рис. 25.2 показан уровень использования ресурсов тремя циклическими конвейерами. На рис. 25.3, который представляет собой вариант отображения данных предыдущей диаграммы с наложением (суммированием) значений, пиковое воздействие, за которым последует подъем по тревоге дежурной смены, произойдет при достижении суммарной нагрузки 1,2 миллиона.

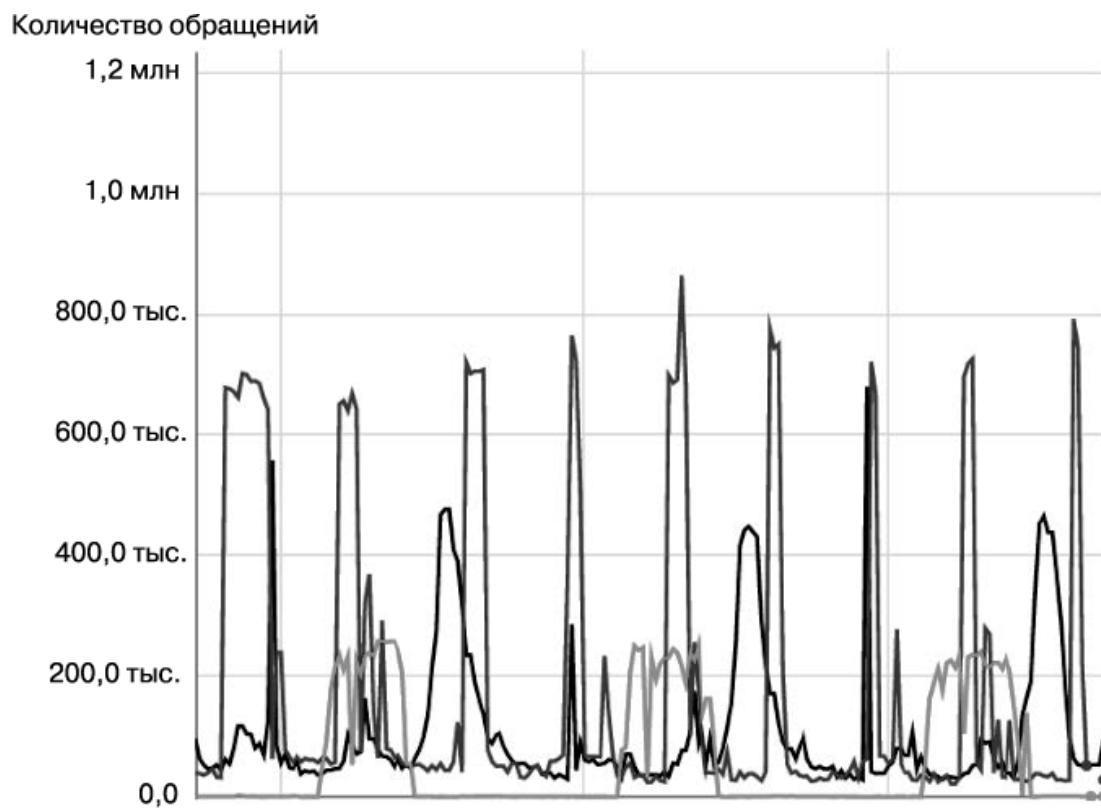


Рис. 25.2. Интерференция нагрузки в инфраструктуре с независимыми ресурсами

Количество обращений

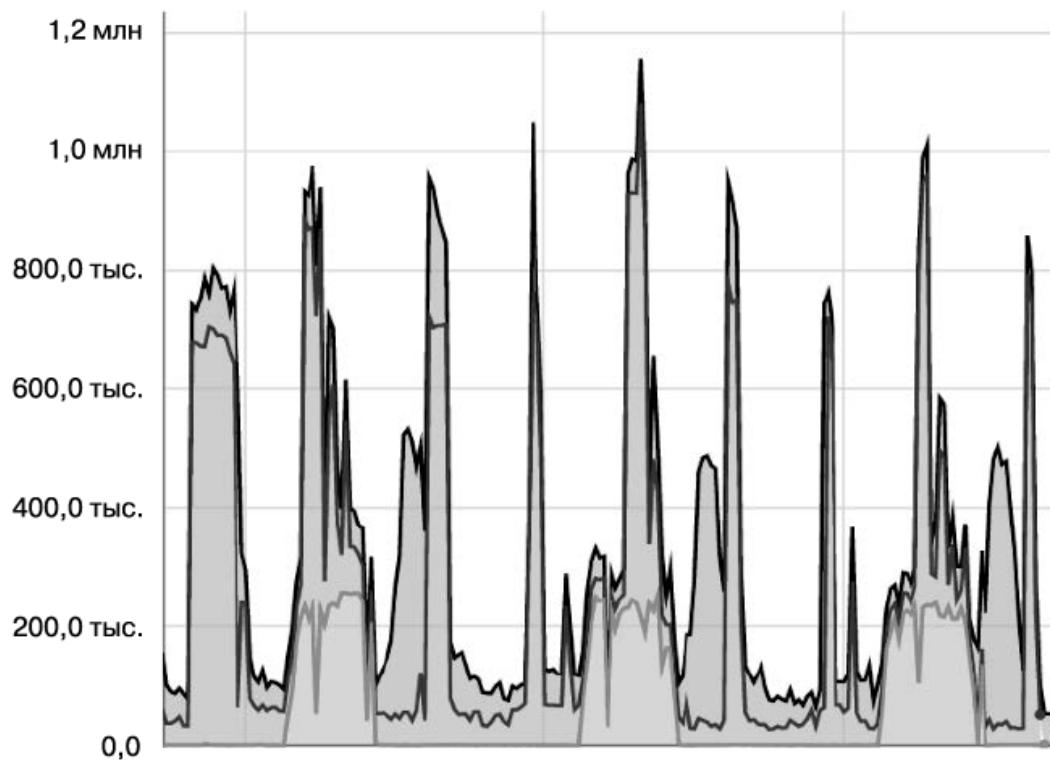


Рис. 25.3. Интерференция нагрузки в инфраструктуре с разделяемыми ресурсами

Знакомство с Google Workflow

Когда по умолчанию одноразовый пакетный конвейер перегружен бизнес-запросами, требующими непрерывно поступающих обновленных результатов, команда разработчиков конвейера обычно рассматривает либо возможность рефакторинга оригинального проекта для удовлетворения текущих потребностей, либо переход на модель непрерывного конвейера.

К несчастью, бизнес-запросы обычно появляются в самое неудобное время для выполнения рефакторинга системы конвейера, переходящего в реализацию продолжительной системы обработки. Новые и более крупные клиенты, которые столкнулись с необходимостью масштабирования, обычно

также хотят присоединения новой функциональности и надеются, что их требования будут выполнены четко в оговоренные сроки. Ожидая такого вызова, важно уточнить некоторые детали в начале проектирования системы, в том числе информацию о предлагаемом конвейере обработки данных. Убедитесь, что вы определили траекторию ожидаемого роста^{[149](#)}, спрос на модификации проекта, ожидаемые дополнительные ресурсы и ожидаемые требования к задержке обработки данных.

Столкнувшись с этими потребностями, компания Google в 2003 году разработала систему Workflow, которая позволила масштабировать продолжительную обработку данных. Workflow использует шаблон «Лидер — последователь» (иногда называют работником), характерный для распределенных систем [Shao, 2000], а также шаблон системного преобладания^{[150](#)}. Эта комбинация позволяет создавать крупномасштабные транзакционные конвейеры, гарантируя корректность благодаря семантике exactly-once («строго однократная доставка»).

Workflow как шаблон «Модель — представление — контроллер»

Из-за способа работы шаблона системного преобладания (System Prevalence) можно считать Workflow эквивалентом шаблона «Модель — представление — контроллер» для распределенных систем, хорошо известного разработчикам интерфейсов^{[151](#)}. Этот шаблон проектирования разделяет заданное приложение на три связанные друг с другом части, чтобы отделить внутреннюю реализацию информации от способов ее представления или получения от пользователя (рис. 25.4).



Рис. 25.4. Шаблон «Модель – представление – контроллер» применяется при проектировании пользовательских интерфейсов

Адаптируя этот шаблон для Workflow, можно сказать, что модель находится на сервере, который называется мастером задач. Он использует шаблон системного преобладания, чтобы удерживать в памяти состояние всех задач для быстрого получения доступа к ним, а синхронные изменения журналов при этом отправляет на диск.

Представление — это работники, которые постоянно обновляют состояние системы транзакционно с помощью мастера в соответствии со своим положением как подкомпонентов конвейера. Несмотря на то что все данные конвейера могут храниться в мастере задач, наилучшая производительность обычно достигается, когда в нем хранятся только указатели на работу, а сами входные и выходные данные располагаются в файловой системе или в другом хранилище. В соответствии с этой аналогией, работники совсем не имеют состояния и могут быть в любой момент удалены.

Контроллер можно опционально добавить как третий компонент системы для эффективного обеспечения влияющих на конвейер второстепенных действий системы, таких как масштабирование конвейера во время работы, получение снимков, управление состоянием рабочего цикла, откат состояния конвейера или даже реализация глобальных ограничений для непрерывности бизнеса. Этот шаблон проектирования показан на рис. 25.5.

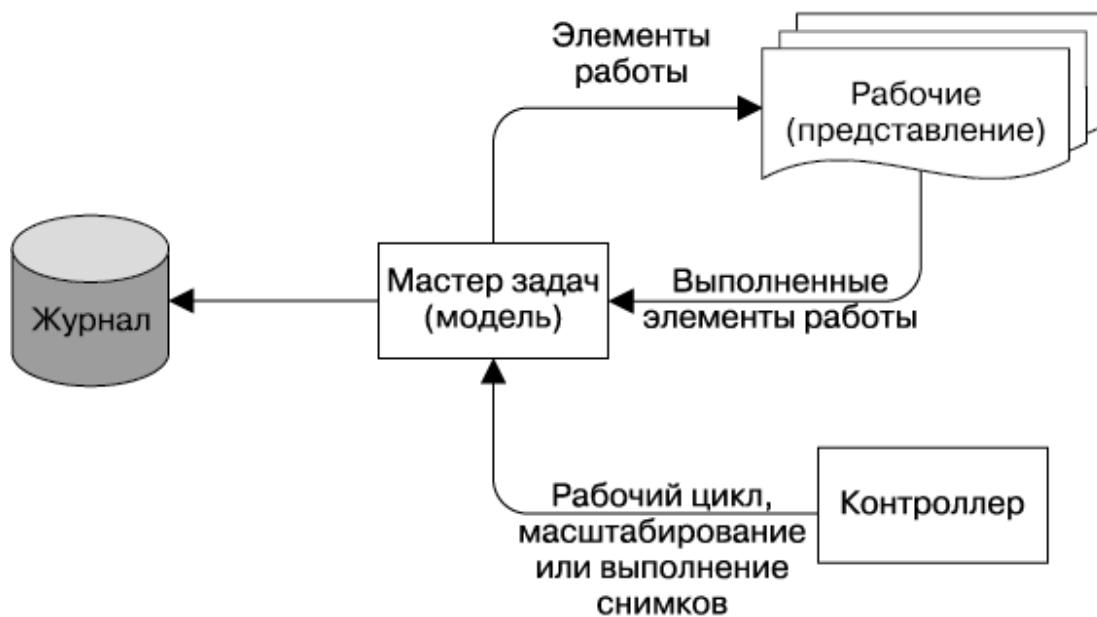


Рис. 25.5. Шаблон проектирования «Модель – представление – контроллер», адаптированный для Google Workflow

Этапы выполнения в Workflow

Внутри Workflow можно увеличить глубину конвейера на любое значение, разбив выполняющиеся задачи на группы, хранимые в мастере задач. Каждая группа задач определяет работу, соответствующую этапу конвейера, который может выполнять произвольные операции для какого-то фрагмента данных. Так можно довольно эффективно выполнять сравнение,

перетасовку, сортировку, разбиение, объединение или любую другую операцию.

С этапом обычно связан некий тип работников. Одновременно могут существовать несколько экземпляров заданного типа работников, и они способны планировать свою деятельность самостоятельно в том смысле, что могут искать разные типы работ и выбирать, что именно будут делать.

Работники используют элементы работы с предыдущих этапов и создают элементы выходных данных. Выходные данные могут быть итоговыми, а могут стать входными данными для другого этапа обработки.

Гарантии корректности Workflow

Сохранять все подробности состояния конвейера внутри мастера задач непрактично, поскольку он ограничен размером оперативной памяти. Однако в этом случае имеется гарантия двойной корректности, так как мастер хранит коллекцию указателей на данные, именованные уникальным образом, и у каждого элемента работы есть уникальный объект аренды. Работники получают работу с помощью объекта аренды и могут отправлять ее только для тех задач, для которых имеют корректный объект аренды.

Для того чтобы избежать ситуации, когда «осиротевший» работник продолжает трудиться над элементом, уничтожая тем самым результаты действующего работника, каждый выходной файл, открытый работником, должен иметь уникальное имя. Таким образом, даже «осиротевшие» работники могут продолжать выполнять запись независимо от мастера до тех пор, пока не попытаются завершить транзакцию. Сделать это они не смогут, поскольку данный элемент работы арендует другой работник. Кроме того, «осиротевшие» работники не

могут уничтожить работу, выполненную действующим работником, поскольку схема уникальных имен файла гарантирует, что каждый работник выполняет запись в отдельный файл. Так реализуется двойная гарантия корректности: выходные файлы всегда уникальны, а состояние конвейера всегда корректно благодаря задачам с арендой.

Если двойной гарантии корректности вам недостаточно, Workflow позволяет присваивать каждой задаче свой номер версии. Если обновляется задача или изменяется аренда, каждая операция получает новую задачу, заменяющую предыдущую, и ей присваивается новый идентификатор. Поскольку все конфигурации Workflow хранятся внутри мастера задач в том же виде, что и сами элементы работы, то для фиксации выполненной работы работник должен владеть активной арендой и сослаться на идентификатор конфигурации, использованной для получения результата. Если во время работы над задачей конфигурация изменилась, все работники этого типа не смогут выполнить фиксацию, несмотря на то что владеют действующими арендами. Поэтому вся работа, выполняемая после смены конфигурации, связана с новой конфигурацией, что определяется работой, сбрасываемой работниками, которым «повезло» обладать устаревшими арендами.

Эти меры предоставляют гарантию тройной корректности: конфигурация, владение арендой и уникальность файловых имен. Однако в некоторых случаях даже этого может оказаться недостаточно. Например, что, если изменился сетевой адрес мастера задач и его по этому адресу заменил другой мастер? А если из-за повреждения памяти изменился IP-адрес или номер порта, что привело к появлению другого мастера задач? Еще чаще встречается ситуация, когда кто-то неверно сконфигурировал своего мастера задач, добавив

балансировщик нагрузки для множества независимых мастеров задач.

Workflow встраивает токен сервера — уникальный идентификатор для конкретного мастера задач — в метаданные каждой задачи, чтобы предотвратить ситуации, когда чужой или неверно сконфигурированный мастер задач портит конвейер. Клиент и сервер проверяют этот токен для каждой операции, что позволяет избежать очень трудноуловимых ошибок конфигурации, и все операции работают гладко до того, как произойдет столкновение модификаторов задач.

Итак, перечислим четыре гарантии корректности для Workflow.

- Результат труда работников с помощью конфигурационных задач создает барьеры, на которых основывается работа.
- Фиксация выполненной работы требует наличия у работника аренды, действительной в данный момент.
- Работники дают выходным файлам уникальные имена.
- Клиент и сервер проверяют правильность мастера задач путем проверки токена сервера в каждой операции.

В этот момент вы можете захотеть отказаться от специализированного мастера задач и использовать Spanner [Corbett, 2012] или другую базу данных. Однако Workflow особенный, поскольку каждая задача уникальна и неизменяема. Эти свойства предотвращают множество потенциальных трудноуловимых проблем, связанных с распределением работы в больших масштабах. Например, аренда, полученная работником, является частью самой

задачи, что требует создания новой задачи даже при изменения аренды. Если база данных используется напрямую и журналы ее транзакций выступают в роли журнала регистрации, то каждая операция чтения должна быть частью долгоиграющей транзакции. Такая конфигурация практически наверняка реализуема, но ужасно неэффективна.

Гарантируем непрерывность бизнеса

Конвейеры, работающие с большими данными, должны продолжать действовать, несмотря на все виды сбоев, включая возникшие из-за перерезанного кабеля, погодных катаклизмов и каскадного отключения питания. Такие сбои могут отключить целые дата-центры. Вдобавок конвейеры, которые не пользуются шаблоном системного преобладания для получения гарантий того, что задача будет выполнена, зачастую отключены и находятся в неопределенном состоянии. Подобный недостаток архитектуры приводит к тому, что нельзя гарантировать непрерывность работы, и приходится удваивать усилия для восстановления конвейеров и данных, а это обходится недешево.

Workflow окончательно решает эту проблему для конвейеров непрерывной обработки. Для достижения глобальной устойчивости мастер задач сохраняет журналы в Spanner, используя повсеместно доступную и чрезвычайно устойчивую файловую систему с низкой полосой пропускания. Чтобы определить мастер задач, который может выполнять операции записи, каждый мастер задач использует распределенный сервис блокировки, который называется Chubby [Burrows, 2006], для выбора программы, выполняющей запись, а затем результат сохраняет в Spanner. Наконец,

клиенты выполняют поиск текущего мастера задач с помощью внутренних сервисов именования.

Поскольку Spanner не может обеспечить работу системы с высокой пропускной способностью, глобально распределенные Workflows используют два или больше локальных Workflows, работающих в разных кластерах, в дополнение к ссылочным задачам, хранящимся в глобальном Workflow. По мере того как конвейер потребляет элементы работы (задачи), эквивалентные справочные задачи внедряются в глобальный Workflow с помощью бинарного файла, имеющего метку «Этап 1» (рис. 25.6). По мере выполнения задач справочные задачи транзакционно удаляются из глобального Workflow, как показано на этапе *n*. Если задачи не могут быть удалены из глобального Workflow, локальный Workflow заблокирует их до тех пор, пока глобальный Workflow не станет доступным, что гарантирует корректность транзакций.

Для того чтобы автоматизировать переход, вспомогательный бинарный файл, помеченный как «Этап 1» (см. рис. 25.6), работает в каждом локальном Workflow. В противном случае локальный Workflow не изменяется, как это описывается в блоке «Выполнение работы» на схеме. Этот вспомогательный бинарный файл с точки зрения MVC действует как контроллер, отвечает за создание справочных задач, а также за обновление специальной задачи, содержащей контрольные сигналы и расположенной внутри глобального Workflow. Если задача, содержащая контрольные сигналы, не обновляется по окончании тайм-аута, удаленный вспомогательный бинарный файл Workflow захватит выполняемую работу, что задокументировано справочными задачами, и работа конвейера, на которую не повлияла среда, продолжится.

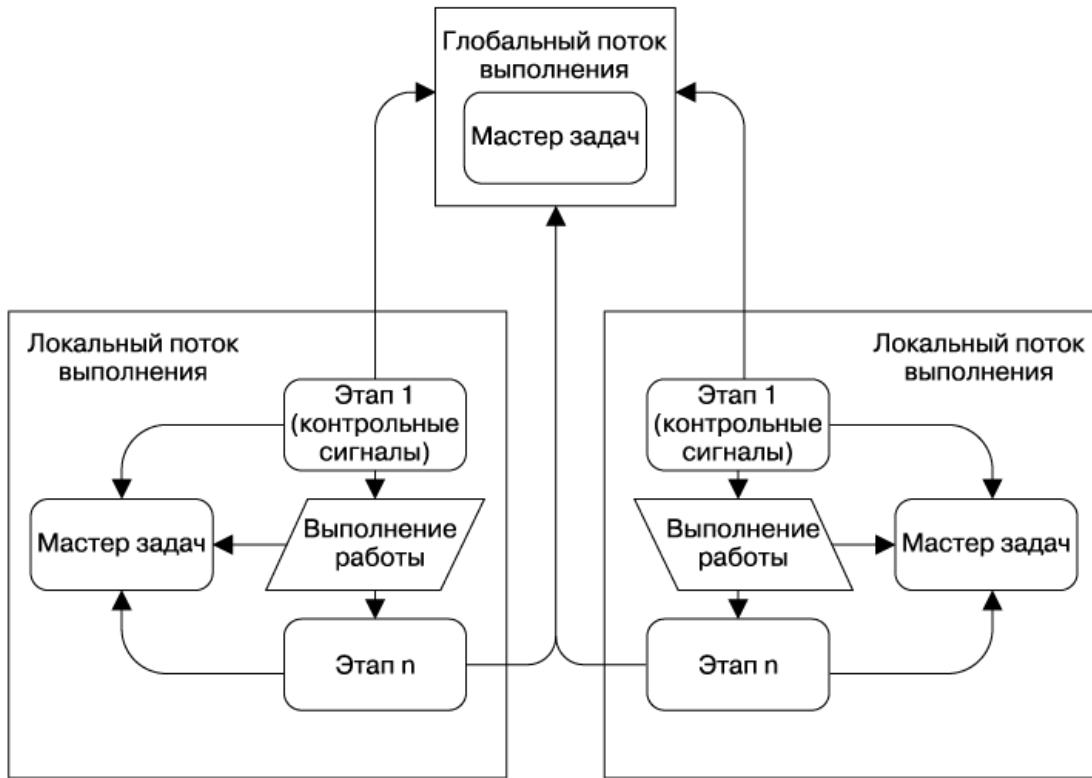


Рис. 25.6. Пример распределенных данных и потока процессов, использующих конвейеры Workflow

Итоги главы и завершающие ремарки

Циклические конвейеры очень полезны. Но не используйте их, если задача обработки данных носит продолжительный характер или стремится к тому. Вместо этого применяйте технологию, напоминающую Workflow.

По нашему опыту, продолжительная обработка данных с гарантированным качеством, обеспечиваемым технологией Workflow, хорошо функционирует и масштабируется в распределенной инфраструктуре кластеров, давая надежные результаты, на которые могут положиться наши пользователи. Эта система является стабильной и надежной, удобной в управлении и обслуживании для SRE-команды.

[148](#) «Википедия». Большие данные // https://ru.wikipedia.org/wiki/Большие_данные.

[149](#) Лекция Джоффа Дина под названием Software Engineering Advice from Building Large-Scale Distributed Systems — это отличный ресурс [Dean, 2007].

[150](#) «Википедия»: System Prevalence, http://en.wikipedia.org/wiki/System_Prevalence.

[151](#) Шаблон «Модель — представление — контроллер» изначально использовался для описания структуры дизайна графических интерфейсов пользователя [Fowler, 2008].

26. Сохранность данных: как пишется, так и читается

Авторы — Рэймонд Блюм и Рандив Синг

Под редакцией Бетси Бейер

Что такое сохранность данных? Когда речь идет о пользователях, сохранность данных — это то, чем она является по их мнению.

Мы можем сказать, что *сохранность данных* — это мера доступности и точности, которую хранилища данных должны обеспечить пользователям, предоставив адекватный уровень обслуживания. Но это неполное определение.

Например, если из-за ошибки пользовательского интерфейса в сервисе Gmail довольно долго отображается, что в почтовом ящике нет писем, пользователи могут решить, что данные потеряны. Поэтому, если на самом деле данные не были потеряны, мир может поставить под сомнение способность компании Google действовать как ответственный распорядитель данных и целесообразность облачных вычислений может оказаться под угрозой. Независимо от того, было отображено сообщение об ошибке или сообщение об обслуживании, пока восстанавливается «лишь небольшой фрагмент метаданных», доверие пользователей Google будет подорвано.

Каково максимальное время, в течение которого данные могут быть недоступны? Как показал реальный сбой, произошедший с сервисом Gmail в 2011 году [Hickins, 2011], четыре дня — это долго, возможно, даже слишком долго. В итоге мы считаем, что 24 часа — это хорошая стартовая точка

для понятия «слишком долго», когда речь идет о приложениях Google.

Аналогичные соображения применяются к приложениям вроде Google Photos, Drive, Cloud Storage и Cloud Datastore, поскольку пользователи не всегда замечают разницу между этими продуктами. (Они обосновывают это так: «Продукт — это все еще часть Google» или «Google, Amazon — какая разница? Этот продукт все еще является частью облака».) Потери данных, их повреждение и продолжительные периоды недоступности обычно неразличимы для пользователей. Поэтому требование сохранности данных должно применяться ко всем типам данных для всех сервисов. При рассмотрении вопроса сохранности данных важно то, что *сервисы в облаке остаются доступными для пользователей. Доступность данных для пользователей особенно важна*.

Прямые требования к сохранности данных

При рассмотрении потребности в надежности заданной системы может показаться, что потребность во времени функционирования (доступность сервиса) должна быть строже, чем потребность в сохранности данных. Например, пользователь может посчитать, что отключение сервиса электронной почты на час — это неприемлемо, при этом он может раздраженно ожидать в течение четырех дней, пока будет восстановливаться почтовый ящик. Однако существует более подходящий способ рассмотреть потребность во времени функционирования и сохранности данных.

Целевое значение для времени функционирования, равное 99,99 %, позволяет сервису находиться в отключенном состоянии всего лишь один час в год. Оно задает довольно

высокую планку, которая, скорее всего, превышает ожидания большинства интернет- и промышленных пользователей.

В противоположность этому целевое значение количества хороших байтов, равное 99,99 %, для двухгигабайтового артефакта обозначает повреждение документов, исполняемых файлов и баз данных (может быть искажено до 200 Кбайт). Такой объем повреждений в большинстве случаев *катастрофичен* — это приводит к появлению исполняемых файлов со случайными кодами операций и совершенно незагружаемых баз данных.

С точки зрения пользователя, к каждому сервису предъявляются независимые требования относительно времени функционирования и сохранности данных, даже если эти требования неявные. Самый худший момент для того, чтобы не согласиться в этом с пользователями, — тот, когда все их данные только что утрачены!



Для того чтобы уточнить приведенное ранее определение сохранности данных, мы можем сказать, что *сохранность данных* означает, что *сервисы, расположенные в облаках, останутся доступными пользователю*. Для него возможность получить доступ к данным особенно важна, поэтому механизмы доступа должны оставаться идеальными.

Теперь предположим, что один артефакт повреждается или теряется ровно один раз в год. Если эта потеря окажется невосполнимой, время функционирования упомянутого артефакта для этого года будет *потеряно*. Скорее всего, чтобы

избежать таких потерь, станут использовать упреждающий поиск, объединенный с быстрым ремонтом.

Предположим, что в альтернативной Вселенной повреждение данных обнаружено мгновенно — еще до того, как оно затронуло пользователя, и этот артефакт удален, исправлен и возвращен сервису в течение получаса. Если кроме этих 30 минут никаких других простоев по техническим причинам не возникнет, такой объект будет доступен 99,99 % времени в год.

Удивительно, по крайней мере с точки зрения пользователя, что в этом сценарии показатель сохранности данных будет равен 100 % или близок к этому значению в течение всего периода доступности объекта. В этом примере было показано, что секрет исключительной сохранности данных состоит в упреждающем обнаружении проблем и быстром восстановлении.

Выбор стратегии для обеспечения отличной сохранности данных

Существует множество стратегий, которые можно применить для быстрого обнаружения, исправления и восстановления потерянных данных. Все эти стратегии обменивают время функционирования на сохранность данных, уважая пользователей, которых коснулась данная ситуация. Одни стратегии работают лучше остальных, другие требуют более сложных инженерных инвестиций. Учитывая, что доступных вариантов очень много, какую стратегию следует использовать? Ответ зависит от вашей парадигмы вычислений. Большая часть приложений для облачных вычислений стремится найти баланс между временем функционирования, задержкой, масштабом, скоростью и приватностью. Дадим определение каждому из этих терминов.

- *Время функционирования.* Количество времени, в течение которого сервис может быть использован. Также называется *доступностью*.
- *Задержка.* Насколько быстро сервис отзыается на обращения пользователей.
- *Скорость.* Как быстро сервис может вводить новшества, чтобы пользователи получили большую пользу по разумной цене.
- *Приватность.* Это концептуально сложное требование. Для простоты в этой главе под приватностью подразумевается то, что данные должны быть уничтожены в разумное время после их удаления пользователем.

Многие облачные приложения непрерывно развиваются на базе API ACID^{[152](#)} и BASE^{[153](#)} для того, чтобы соответствовать требованиям по этим пяти параметрам^{[154](#)}. BASE предоставляет большую доступность, чем ACID, она дается взамен более мягких гарантий распределенной устойчивости. В частности, BASE может гарантировать только то, что значение фрагмента данных, который больше не обновляется, в итоге станет согласованным в потенциально распределенных местах хранилища. В следующем сценарии показано, как работают компромиссы для времени выполнения, задержки, масштаба, скорости и приватности.

Когда остальные требования приносят в жертву ради скорости, получившиеся приложения будут зависеть от произвольной коллекции API, с которыми лучше всего знакомы определенные разработчики этих приложений.

Например, приложение может пользоваться эффективным API хранилища BLOB¹⁵⁵, таким как Blobstore, которое пренебрегает распределенной устойчивостью ради масштабирования при высоких нагрузках с продолжительным функционированием, малой задержкой и низкой стоимостью. Для того чтобы это компенсировать:

- приложение может передавать небольшие объемы метаданных, принадлежащих его блобам, что приведет к созданию приложения, основанного на Paxos, с большей задержкой, низкой доступностью и более высокой стоимостью — наподобие Megastore [Baker, 2011], [Lamport, 1998];
- некоторые клиенты приложения могут кэшировать определенный объем этих метаданных локально и получать доступ к блобам непосредственно, еще больше снижая задержку с точки зрения пользователей;
- другое приложение может хранить метаданные в Bigtable, частично жертвуя устойчивостью, поскольку его разработчики были знакомы с Bigtable.

Такие облачные приложения сталкиваются с разнообразными серьезными трудностями во время выполнения программы, например со ссылочной целостностью между хранилищами данных (в предыдущем примере — Blobstore, Megastore и кэши на стороне клиента). Влияние высокой скорости говорит о том, что изменения схемы, миграция данных, создание новой функциональности на основе старой, переписывание кода и развитие точек взаимодействия с другими приложениями создают окружение, усложненное запутанными отношениями между разными

фрагментами данных, в которых полностью не разбирается ни один инженер.

Для того чтобы данные такого приложения не деградировали до того, как попадут на глаза пользователям, необходимо создать систему внеполосных проверок и баланса внутри хранилищ данных и между ними. Такая система рассматривается в подразделе «Третий уровень: раннее обнаружение» далее в этой главе.

В дополнение к этому, если такое приложение полагается на независимые некоординируемые резервные копии нескольких хранилищ (в предыдущем примере Blobstore и Megastore), то его способность эффективно использовать данные, полученные в ходе восстановления, ухудшается из-за разнообразных отношений между восстановленными и рабочими данными. Приложению из нашего примера пришлось бы выделять необходимые данные из восстановленных блобов и рабочих данных из Megastore, восстановленных данных из Megastore и рабочих данных из блобов, восстановленных блобов и восстановленных данных из Megastore, а также взаимодействовать с кэшами клиентов.

Учитывая все эти зависимости и сложности, сколько ресурсов нужно выделить на сохранность данных и где?

Резервные копии или архивы?

Как правило, компании защищают данные от потерь путем инвестиций в разработку стратегий резервного копирования. Однако эти усилия должны быть сосредоточены не на создании резервных копий, а на восстановлении данных, что позволяет отличать резервные копии от архивов. Как иногда говорится, никто не хочет делать резервные копии, но все хотят восстановления данных.

Является ли ваша резервная копия архивом, подходящим для восстановления данных в случае катастрофы?



Наиболее важным различием между резервными копиями и архивами является то, что резервные копии *могут* быть загружены обратно в приложение, а архивы — нет. Поэтому резервные копии имеют разное применение.

В *архивах* данные безопасно хранятся в течение длительного времени для проведения проверок, обнаружения и соблюдения правил. Восстанавливать данные для этих целей не обязательно в рамках требований к времени функционирования сервиса. Например, вам может понадобиться сохранить данные о финансовых транзакциях за 7 лет. Для этого можете перемещать собранные журналы проверок в долгосрочное архивное хранилище, расположенное за пределами площадки, раз в месяц. Получение и восстановление журналов во время финансовой проверки, длившейся один месяц, может занять целую неделю, и для архива такое продолжительное время восстановления будет приемлемым.

Однако, если случится катастрофа, данные должны быть быстро восстановлены из *резервных копий*, причем желательно, чтобы это произошло в соответствии с потребностями во времени функционирования сервиса. В противном случае пользователи, которых коснется эта ситуация, не смогут работать с приложением с момента появления проблемы с

сохранностью данных до того, как восстановление данных завершится.

Важно также иметь в виду следующее: поскольку большая часть недавно полученных данных находится под угрозой до тех пор, пока не будет сделана их резервная копия, лучше запланировать регулярное создание реальных резервных копий (в противоположность архивам): ежедневно, ежечасно или по подходящему вам графику.

Поэтому при формулировании стратегии создания резервных копий задумайтесь, как быстро вам нужно восстанавливаться после возникновения проблем и какое количество данных вы можете позволить себе потерять.

Требования к облачному окружению на перспективу

При создании облачного окружения наблюдается уникальная комбинация технических сложностей.

- Если окружение использует транзакционные и нетранзакционные резервные копии и решения по восстановлению, то восстановленные данные не обязательно будут корректными.
- Если сервисы должны развиваться, не отключаясь, разные версии бизнес-логики могут работать с данными параллельно.
- Если взаимодействующие сервисы получают версии независимо друг от друга, несовместимые версии разных сервисов могут какое-то время взаимодействовать между собой, повышая вероятность того, что данные будут случайно повреждены или потеряны.

В дополнение к этому, для поддержания принципа экономии на масштабе поставщики услуг должны предоставлять ограниченное количество API. Эти API должны быть простыми, чтобы их можно было использовать в большинстве приложений, или они будут пригодны лишь для нескольких пользователей. В то же время API должны быть достаточно продуманными и иметь следующую функциональность:

- локальность данных и кэширование;
- локальное и глобальное распределение данных;
- высокую устойчивость или устойчивость в конечном счете;
- долговечность данных, резервное копирование и восстановление.

В противном случае взыскательные клиенты не смогут перенести приложение в облако и простые приложения, которые станут сложными и крупными, нужно будет полностью переписать для того, чтобы использовать другие, более сложные API.

Проблемы появляются, когда указанные особенности API задействуются в определенных комбинациях. Если поставщик услуг не может решить эти проблемы, приложение, которое столкнется с такими сложностями, должно определить их и разрешить самостоятельно.

Целевые значения показателей сохранности и доступности данных для SRE

Да, глобальная цель SRE, которая заключается в поддержании целостности устойчивых данных, неплоха, но мы стремимся к конкретным целям с измеряемыми показателями. Служба SRE определяет ключевые показатели, которые мы применяем для формирования ожиданий от наших систем и процессов с помощью тестирования. Эти показатели используются также для того, чтобы отследить производительность систем во время реального события.

Сохранность данных – это средство, доступность данных – цель

Сохранность данных – это точность и устойчивость данных на протяжении периода их существования. Пользователям нужно знать, что информация будет корректной и после ее создания не изменится непредсказуемым образом. Но достаточно ли такой уверенности?

Рассмотрим ситуацию, когда данные поставщика услуг электронной почты были недоступны целую неделю [Kincaid, 2009]. В течение десяти дней пользователям приходилось искать другие, временные средства ведения бизнеса в надежде, что они скоро вернутся к привычной электронной почте с контактами и историей переписки.

Но позже произошло самое плохое: провайдер объявил, что, несмотря на все ожидания, часть электронных писем и контактов пропала, они попросту безвозвратно испарились. Казалось, что серия сбоев при управлении сохранностью данных оставила поставщика услуг без резервных копий. Разъяренные пользователи либо продолжили использовать временные аккаунты, либо создали новые, уйдя от прежнего поставщика услуг электронной почты.

Но погодите! Через несколько дней после сообщения о полной потере данных поставщик объявил, что личная

информация пользователей может быть восстановлена. Данные не утрачены, произошло лишь отключение. Все хорошо!

Кроме того, что *не все было хорошо*. Пользовательские данные сохранились, но люди, которым они были нужны, слишком долго не могли получить к ним доступ.

Мораль такова: с точки зрения пользователя, если данные есть, но к ним нельзя получить доступ, то их, по сути, нет.

Создаем систему восстановления, а не систему резервного копирования

Создание резервных копий — это классическая задача системного администрирования, выполнением которой пренебрегают, которую стремятся переложить на плечи других или задвинуть в долгий ящик. Резервное копирование ни для кого не высокоприоритетно — оно представляет собой постоянную трату времени и ресурсов и не приносит мгновенного видимого эффекта. Кто-то может поспорить, что, как и для большей части мер по защите от низкоуровневых опасностей, такой подход является прагматичным. Основная проблема, связанная с этой стратегией, заключается в том, что возможные опасности могут нести небольшой риск, но при этом оказывать серьезное воздействие. Если данные сервиса недоступны, ваша реакция может определить будущее сервиса, продукта или даже всей компании.

Вместо того чтобы концентрироваться на сложностях резервного копирования, гораздо полезнее, не говоря уже о том, что проще, обосновывать его необходимость, сосредотачиваясь на задачах с видимым результатом, то есть на *восстановлении!* Создание резервных копий — это «налог», который уплачивается на постоянной основе муниципальным

службам, гарантирующим доступность данных. Вместо того чтобы делать акцент на «налоге», обращайте внимание на «налоговые» сервисы — доступность данных. Мы не заставляем команды тренироваться выполнять резервное копирование, вместо этого они:

- определяют целевые значения показателей (SLO) для сохранности данных при разных типах неисправностей;
- тренируются и демонстрируют свою способность соответствовать этим SLO.

Типы сбоев, которые ведут к потере данных

На самом высоком уровне существует 24 вида сбоев, когда три фактора могут соединиться в любой комбинации (рис. 26.1). Вы должны рассмотреть каждый из потенциальных сбоев при разработке программы сохранности данных. Рассмотрим факторы сохранности данных при возможных неполадках.

- *Основная причина.* Невосстановимая потеря данных может быть вызвана многими причинами — действиями пользователя, ошибкой в работе, ошибкой приложения, дефектами инфраструктуры, сбоем оборудования или аварией на площадке.
- *Масштаб.* Одни потери данных распространяются широко, влияя на множество объектов. Другие узконаправленные — иногда оказываются удалены или повреждены данные, необходимые небольшому количеству пользователей.
- *Уровень.* Одни потери данных похожи на «большой взрыв» (например, один миллион рядов был заменен десятью рядами всего за минуту), а другие нарастают постепенно

(например, десять рядов данных удаляются каждую минуту в течение нескольких недель).

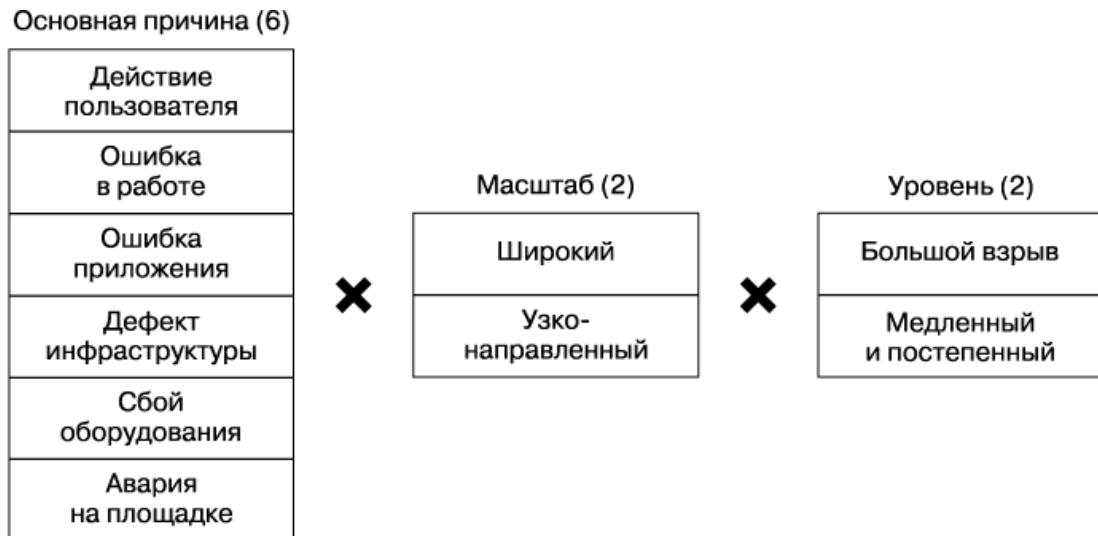


Рис. 26.1. Факторы сохранности данных при возможных неполадках

Эффективный план восстановления должен учитывать все эти типы неполадок, происходящих в самых разнообразных комбинациях. Идеальная эффективная стратегия защиты от потери данных, вызванной нарастающим количеством ошибок приложения, может оказаться абсолютно бесполезной, если ваш дата-центр начнет гореть.

Исследования 19 попыток восстановления данных в компании Google показали, что потери данных, наиболее заметные пользователям, включают в себя удаление данных или нарушение ссылочной целостности, вызванные ошибками программного обеспечения. Наиболее сложные случаи повреждения или удаления данных были обнаружены спустя неделю или даже месяцы после первого появления ошибок в производственной среде. Поэтому меры безопасности, предпринимаемые в компании Google, должны подходить для предотвращения таких потерь или восстановления после них.

Чтобы восстановиться при таком развитии событий, крупное и успешное приложение должно получать данные, сгенерированные за дни, недели или месяцы для миллионов возможных пользователей. Приложению также может понадобиться восстанавливать каждый затронутый объект к определенному лишь для него моменту времени. Такой сценарий восстановления данных за пределами компании Google называется использование точки восстановления, а внутри нее — путешествие во времени.

Решение о создании резервных копий и восстановлении данных, которое обеспечивает возможность восстанавливать данные с помощью точки восстановления для хранилищ данных, основанных на семантиках ACID и BASE, соответствующим образом целевым значениям времени функционирования, задержки, масштабируемости, скорости и стоимости, на сегодняшний день является несбыточной мечтой!

Решение этой проблемы усилиями собственных инженеров приведет к тому, что вам придется пожертвовать скоростью. Многие проекты идут на компромисс, используя многоуровневую стратегию резервного копирования без функциональности точек восстановления. Например, API, лежащий в основе вашего приложения, может поддерживать множество механизмов восстановления данных. Дорогие локальные снимки способны дать частичную защиту от ошибок приложения, поэтому вы можете хранить такие снимки, выполняющиеся каждые несколько часов, на протяжении нескольких дней. Эффективные с точки зрения стоимости полные и инкрементные копии, создаваемые каждые два дня, могут храниться дольше. Точка восстановления — это полезная функциональность, если ее поддерживают одна или несколько стратегий.

Рассмотрим варианты восстановления данных, обеспечиваемые облачными API, которые вы скоро начнете применять. Смените возможность их выполнения с помощью точек восстановления на многоуровневую стратегию, если это необходимо, но обязательно используйте хотя бы один из этих способов! Если вы можете задействовать обе функции, сделайте это. Каждая из этих функций или даже обе в какой-то момент окажутся полезными.

Сложности поддержания высокой сохранности данных

При разработке программы поддержания целостности данных важно иметь в виду, что *репликация и избыточность — это не восстанавливаемость*.

Проблемы масштабирования: возможности резервных копий и восстановления

Классическим неверным ответом на вопрос: «У вас есть резервная копия?» является: «У нас есть кое-что получше — репликация!» Репликация имеет множество преимуществ, включая локальность данных и защиту от катастроф в местах расположения, но она не может защитить вас от множества способов потерять данные. В хранилищах данных, которые автоматически синхронизируют несколько реплик, повреждение столбцов базы данных или случайное удаление данных произойдет во всех копиях, скорее всего, еще до того, как вы сможете устраниТЬ проблему.

Для того чтобы решить эту проблему, вы можете создавать необслуживающие копии ваших данных в другом формате, например часто выполнять экспорт из базы данных в исходный файл. Эта дополнительная мера дает защиту от ошибок, от которых репликация спасти не может, — пользовательских

ошибок и ошибок на уровне приложения, но она не способна ничего сделать с потерями данных на более низком уровне. Эта мера также создает риск появления ошибок во время конвертирования данных (в обоих направлениях) и во время сохранения исходного файла в дополнение к возможным несоответствиям между двумя форматами. Представьте атаку нулевого дня^{[156](#)}, которой подвергся некий низкий уровень вашего стека, например файловая система или драйвер устройства. Все копии, которые относились к атакованному программному компоненту, включая экспорт из базы данных, записанный для файловой системы, поддерживающей вашу базу данных, становятся уязвимыми.

Поэтому мы видим, что очень важно обеспечить разнообразие: защита от сбоя на слое X требует сохранения данных на разных компонентах этого слоя. Изоляция носителей защищает от их недостатков: ошибка или атака на драйвер дискового устройства, скорее всего, не повлияют на накопители на магнитной ленте. Если бы мы могли, то создавали бы резервные копии ценных данных на глиняных табличках^{[157](#)}.

Что важнее: поддерживать данные в актуальном состоянии и восстанавливать их или обеспечить полную их защиту? Чем ниже в стеке находится снимок данных, тем больше времени потребуется на то, чтобы сделать их копию, что означает снижение частоты выполнения резервного копирования. На уровне баз данных репликация транзакции может занять несколько секунд, экспортование снимка базы данных в файловую систему — 40 минут. Создание полной резервной копии файловой системы может длиться часы.

Вы можете потерять данные, которые появились за эти 40 минут, если выполняете восстановление из последнего снимка. Восстановление из резервной копии файловой системы может

повлечь за собой утрату многих часов транзакций. Наверняка вы хотели бы получить самую свежую копию максимально быстро, но в зависимости от типа сбоя эта самая свежая и мгновенно доступная копия может оказаться плохим вариантом.

Сохранность

Сохранность — показатель того, как долго вы храните копии ваших данных, — это еще один фактор, который следует принимать во внимание при создании планов восстановления данных.

Скорее всего, вы или ваши клиенты быстро обнаружите исчезновение всей базы данных, но на то чтобы постепенная потеря данных привлекла внимание нужных людей, может потребоваться несколько дней. Восстановление утерянных данных в последнем сценарии потребует использования сделанных ранее снимков. При получении настолько старых данных вы, вероятно, захотите объединить восстановленные данные с данными в текущем состоянии. Это значительно усложнит процесс восстановления.

Как служба SRE справляется с трудностями обеспечения сохранности данных

Мы думаем, что системы, лежащие в основе наших сервисов, ненадежны, и точно так же предполагаем, что все механизмы защиты могут дать сбой в самый неподходящий момент. Поддержание гарантии сохранности данных в больших масштабах — это вызов, который еще больше усложняется высокой степенью изменений связанных программных систем и требует выполнения вспомогательных, но несвязанных

приемов, каждый из которых сам по себе предлагает высокую степень защиты.

Двадцать четыре комбинации типов сбоев с точки зрения сохранности данных

Способов, которыми можно потерять данные, множество (как описано ранее), поэтому не существует панацеи, которая защитила бы нас от всех комбинаций типов сбоев. Вместо этого вам нужна глубокая защита. Она объединяет несколько слоев, где каждый последующий уровень защищает от все менее распространенных сценариев потери данных. На рис. 26.2 показаны путь объекта от мягкого удаления до разрушения, а также перечислены стратегии восстановления данных, которые следует применять на протяжении этого пути для того, чтобы гарантировать глубокую защиту.

Первый уровень — это *мягкое* (или *ленивое*) *удаление*, которое показало себя эффективным способом защиты от сценариев непреднамеренного удаления данных. Вторая линия обороны — это *резервные копии и связанные с ними методы восстановления*. Третий, последний уровень — это *обычная проверка данных*, она рассматривается в подразделе «Третий уровень: раннее обнаружение» далее. Для всех этих уровней наличие *репликации* полезно лишь иногда и при определенных сценариях (планы восстановления данных не должны полагаться на репликацию).



Рис. 26.2. Путь объекта от мягкого удаления до разрушения

Первый уровень: мягкое удаление

Когда скорость разработки высока и имеет значение приватность, ошибки в приложениях ответственны за большую часть событий, связанных с потерей и повреждением данных. Фактически ошибки, связанные с удалением данных, могут стать настолько распространеными, что способность отменять удаление данных в рамках ограниченного промежутка времени станет основным способом защиты от потери данных, которые в противном случае станут постоянными.

Любой продукт, который гарантирует конфиденциальность данных своих пользователей, должен позволять им удалять выбранные фрагменты данных и/или их все. Такие продукты трудно поддерживать из-за риска случайного удаления. Возможность отменить удаление данных (например, с помощью корзины) снижает, но не может полностью избавить вас от этих трудностей, особенно если ваш сервис

поддерживает и сторонние надстройки, которые также могут удалять данные.

Мягкое удаление может значительно снизить количество трудностей поддержки или даже полностью избавить от них. Мягкое удаление означает, что удаленные данные мгновенно помечаются как удаляемые, что делает их недоступными для использования для большей части кода приложения, исключая административные его ветви. С административными ветвями кода могут быть связаны предоставление информации по запросу, восстановление похищенных учетных записей, промышленное администрирование, поддержка пользователей, поиск проблем и соответствующая этим аспектам функциональность. Выполняйте мягкое удаление, когда пользователь очищает свою корзину, и предоставьте ему инструмент поддержки, который позволяет авторизованным администраторам отменить случайное удаление данных. Компания Google реализует эту стратегию для того, чтобы большинство наших популярных приложений были продуктивными, в противном случае трудности, связанные с поддержкой пользователей, были бы невыносимыми.

Вы можете расширить стратегию мягкого удаления, предоставив пользователям способ восстановить удаленные данные. Например, корзина в приложении Gmail дает пользователям возможность получить доступ к удаленным сообщениям в течение 30 дней.

Часто нежелательное удаление данных происходит из-за хищения учетных записей. В этом случае похититель, как правило, удаляет данные владельца учетной записи перед тем, как использовать ее для рассылки спама и других незаконных целей. Когда вы связываете случайное удаление данных пользователем с риском их уничтожения похитителем,

необходимость программного интерфейса мягкого удаления и его отмены в вашем приложении становится очевидной.

Мягкое удаление данных подразумевает, что данные, помеченные как удаляемые, будут уничтожены по прошествии определенного времени. Длина этого промежутка зависит от политики организации и применяемых законов, доступных ресурсов хранилища и их стоимости, цены продукта и позиционирования рынка, особенно при наличии большого количества недолговечных данных. Распространенными вариантами таких промежутков являются 15, 30, 45 или 60 дней. Опыт компании Google свидетельствует, что большая часть случаев хищения учетных записей и проблем с целостностью данных обнаруживаются в течение 60 дней. Поэтому хранить данные после мягкого удаления больше 60 дней не имеет смысла. Мы также обнаружили, что наиболее ощутимые разрушения данных вызваны разработчиками, незнакомыми с существующим кодом, которые работают над кодом, связанным с удалением данных, особенно если это конвейеры с пакетной обработкой (например, онлайн-конвейер MapReduce или Hadoop). Разрабатывать интерфейсы так, чтобы помешать разработчикам, незнакомым с вашим кодом, обмануть механизм мягкого удаления с помощью нового кода, довольно полезно.

Одним из эффективных способов сделать это является реализация программы облачных вычислений, которые включают в себя встроенные API мягкого удаления и его отмены, позволяющие гарантировать, что у нас будет *необходимая функциональность*¹⁵⁸. Даже наилучшая броня окажется бесполезной, если вы ее не наденете.

Стратегии мягкого удаления охватывают функциональность по удалению данных в потребительских продуктах вроде Gmail

или Google Drive, но что, если вы хотите вместо этого воспользоваться программой облачных вычислений?

Если предположить, что программа облачных вычислений уже поддерживает программное мягкое удаление и его отмену с разумными параметрами по умолчанию, то прочие сценарии случайного удаления данных будут проистекать из ошибок, допущенных либо внутренними разработчиками, либо разработчиками — потребителями вашего сервиса.

В таких случаях может быть полезно создать дополнительный уровень мягкого удаления, который мы называем «ленивым» удалением. Вы можете подумать, что мягкое удаление — это закулисное очищение, которым управляет система хранения (мягкое удаление управляет и реализуется в клиентском приложении или сервисе). В случае использования «ленивого» удаления данные, удаленные облачным приложением, мгновенно становятся недоступными, но поставщик облачных услуг сохраняет их в течение нескольких недель до полного удаления. «Ленивое» удаление рекомендуется применять не для всех стратегий глубокой защиты. Длительный период «ленивого» удаления может оказаться дорогим для систем, работающих с большим количеством недолгосрочных данных. Оно также может быть неэффективным в системах, которые должны гарантировать уничтожение удаленных данных за разумный промежуток времени, то есть в системах, предлагающих гарантированную приватность.

Подытоживая разговор о первом уровне глубокой защиты, отметим следующее.

- Создание корзины, которая позволяет пользователям отменить удаление данных, — это основной способ защиты от пользовательских ошибок.

- Мягкое удаление — основной способ защиты от ошибок внешних разработчиков и вторичный — от пользовательских ошибок.
- В программах для разработчиков ленивое удаление — это основной способ защиты от ошибок внутренних разработчиков и вторичный — от ошибок внешних разработчиков.

Как насчет *истории исправлений*? Некоторые продукты предоставляют возможность откатить элементы к предыдущему состоянию. Когда такая функциональность становится доступной пользователям, она представляет собой некую форму корзины. Если она доступна разработчикам, ее можно использовать вместо мягкого удаления в зависимости от ее реализации.

В Google история исправлений оказалась полезной для восстановления после того, как реализовались некоторые сценарии повреждения данных, но она не годится для восстановления в большинстве случаев потери данных, включающих случайное их удаление. Это происходит потому, что некоторые реализации истории исправлений считают удаление особым случаем, при котором предыдущие состояния должны быть удалены, в противоположность изменению элемента, чья история может быть сохранена на определенный период времени. Для того чтобы предоставить адекватную защиту от нежелательного удаления, следует применять мягкое удаление и для истории исправлений.

Второй уровень: резервные копии и связанные с ними методы восстановления

Резервные копии и восстановление данных — это вторая линия защиты после мягкого удаления. Наиболее важным принципом для этого слоя является то, что сами резервные копии ничего не значат — важно только восстановление. Факторы, поддерживающие успешное восстановление, должны управлять вашими решениями при создании резервных копий, но не наоборот.

Другими словами, прежде чем определять политику резервного копирования, задайте себе следующие вопросы.

- Какие методы создания резервных копий и восстановления данных нужно использовать?
- Насколько часто вы будете создавать точки восстановления, создавая полные или инкрементные резервные копии своих данных?
- Где вы будете хранить резервные копии?
- Как долго станете сохранять резервные копии?

Сколько недавно полученных данных вы готовы потерять во время восстановления? Чем меньше данных вы можете позволить себе потерять, тем серьезнее должны отнестись к стратегии создания инкрементных резервных копий. В одном из самых запоминающихся случаев, произошедших в компании Google, мы создавали резервные копии для более старой версии сервиса Gmail практически в реальном времени.

Даже если деньги не являются для вас ограничивающим фактором, частое создание резервных копий может оказаться накладным в другом смысле. Наиболее примечательно то, что этот процесс создает вычислительные сложности для работающих хранилищ данных вашего сервиса во время

обслуживания пользователей, что заставляет сервис приближаться к границам масштабируемости и производительности. Для того чтобы облегчить ситуацию, вы можете в непиковые часы создавать полные резервные копии, а в то время, когда сервис занят сильнее, — инкрементные резервные копии.

Как быстро нужно восстанавливаться? Чем быстрее ваши пользователи должны быть спасены, тем более локальными должны быть резервные копии. Зачастую компания Google сохраняет более дорогие, но и более быстрые для восстановления снимки¹⁵⁹ на короткое время внутри экземпляра хранилища и чуть дольше хранит более поздние резервные копии в распределенном хранилище с произвольным доступом внутри того же или близлежащего data-центра. Такая стратегия сама по себе не защитит вас от катастроф в местах расположения, поэтому зачастую эти резервные копии отправляют в ближайшие (или офлайн-) пункты на более продолжительное время, перед тем как они уступят место новым резервным копиям.

Насколько старыми могут быть резервные копии? Стратегия создания резервных копий становится более дорогой по мере увеличения возраста самых старых копий, но при этом увеличивается количество сценариев, при которых можно выполнить восстановление (однако это увеличение влечет за собой снижение эффективности).

Опыт компании Google показывает, что ошибки, связанные с низкоуровневой мутацией данных или их удалением, находящиеся в коде приложения, требуют возврата назад на наибольший промежуток времени, поскольку некоторые из них обнаруживаются лишь спустя несколько месяцев после того, как вы начнете терять данные. Такие ситуации предполагают,

ЧТО ВЫ МОЖЕТЕ ВЕРНУТЬСЯ В ПРОШЛОЕ ТАК ДАЛЕКО, КАК ТОЛЬКО ВОЗМОЖНО.

В то же время в высокоскоростном окружении разработки изменения кода и структуры могут сделать старые резервные копии дорогими или их и вовсе станет невозможно использовать. Помимо этого, будет трудно восстановить разные наборы данных из разных точек восстановления, поскольку это потребует применения нескольких резервных копий. Однако именно таким образом вы сможете восстановиться после низкоуровневого повреждения данных или их удаления.

Стратегии, описанные в подразделе «Третий уровень: раннее обнаружение» далее, должны ускорить обнаружение низкоуровневых мутаций данных или багов в коде, связанных с удалением данных, хотя бы частично снижая необходимость выполнения такого сложного процесса восстановления. Как же вы можете обеспечить разумную защиту до того, как поймете, какого рода проблемы вам придется искать?

Наше решение — выбрать срок от 30 до 90 дней хранения резервных копий для большинства сервисов. Попадание сервиса в это окно зависит от его терпимости к потере данных и инвестиций в раннее обнаружение.

Подытоживая совет по защите от 24 комбинаций типов отказа с точки зрения сохранности данных: устранение последствий развития большого количества сценариев за разумную стоимость требует применения многоуровневой стратегии создания резервных копий. Первый уровень состоит в создании большого количества резервных копий, с помощью которых можно быстро выполнить восстановление и которые хранятся максимально близко к работающим хранилищам данных, возможно, использующим ту же или похожую технологию хранения, что и источники данных. Это дает

защиту от большинства проблем, включая ошибки в программном обеспечении и ошибки разработчиков. Из-за относительной дороговизны резервные копии сохраняются на этом уровне от нескольких часов до десяти дней, и для восстановления с их помощью потребуется всего несколько минут.

Второй уровень состоит из меньшего количества резервных копий, которые могут храниться на протяжении нескольких недель в распределенных файловых системах с произвольным порядком выборки. Для того чтобы выполнить восстановление данных с помощью этих резервных копий, может потребоваться несколько часов, они дают дополнительную защиту от сбоев, влияющих на определенные технологии хранения обслуживающего стека, но не на технологии, использованные для хранения резервных копий. Этот уровень защищает также от ошибок приложения, которые были обнаружены слишком поздно, когда первый уровень защиты уже не может помочь. Если вы создаете новые версии кода два раза в неделю, имеет смысл сохранить эти резервные копии минимум на неделю или две.

Последующие уровни пользуются полуоперативными хранилищами, такими как специализированные библиотеки работы с лентами и расположенные вне площадок хранилища запасных носителей, например на накопителях на магнитной ленте или на дисковых накопителях. Резервные копии на этих уровнях предоставляют защиту от проблем, возникающих на площадке, таких как отключения питания дата-центра или повреждения распределенной системы из-за ошибки.

Перемещать большие объемы данных между уровнями дорого. В то же время производительность хранилища на более высоких уровнях не конкурирует с ростом количества экземпляров обслуживающих производственных хранилищ

сервиса. В результате резервное копирование на этих уровнях выполняется реже, но копии сохраняются дольше.

Основной уровень: репликация

В идеальном мире каждый экземпляр хранилища, включая те, что содержат резервные копии, должен быть реплицирован. Во время восстановления данных последнее, что вы хотите обнаружить, — это то, что резервные копии сами потеряли необходимые данные или что дата-центр, содержащий наиболее полезные резервные копии, находится на профилактическом обслуживании.

По мере увеличения объема данных становится не всегда возможно выполнить репликацию каждого экземпляра хранилища. В таких случаях имеет смысл разместить успешные резервные копии на разных площадках, каждая из которых может дать сбой независимо от других, и записать резервные копии с помощью избыточного метода, например RAID, кодов Рида — Соломона или репликации GFS¹⁶⁰.

При выборе избыточной системы не полагайтесь на нечасто применяемую схему, единственные «тесты» эффективности которой представляют собой лишь ваши нечастые попытки восстановления данных. Вместо этого выберите популярную схему, которую часто применяют многие пользователи.

От терабайта к экзабайту — это не просто «больше»

Процессы и практики, применяемые к объемам данных, измеряемых в терабайтах (T), плохо масштабируются для данных, измеряемых в экзабайтах (E). Проверка, копирование и тестирование нескольких гигабайт структурированных данных — это интересная задача. Однако, если предположить,

что вы хорошо знаете свою схему и модель транзакций, выполнить это упражнение не составит особого труда. Как правило, вам будет достаточно лишь получить ресурсы машины для того, чтобы пройтись по данным, реализовать логику проверки и выделить достаточно места для сохранения нескольких копий данных.

Теперь поднимем ставки: вместо нескольких гигабайт попробуем проверить 700 Гбайт структурированных данных и обеспечить их безопасность. Если предположить, что идеальная производительность SATA 2.0 — 300 Мбайт/с, то выполнение одной задачи, которая проходит по всем данным и проделывает примитивные проверки, займет восемь десятилетий. Создание нескольких полных резервных копий при условии, что имеется достаточно места, займет как минимум столько же времени. Время восстановления, учитывая постобработку, будет еще больше. Теперь потребуется почти 100 лет для того, чтобы восстановить данные из резервной копии, которой было 80 лет на момент, когда началось восстановление. Очевидно, такую стратегию стоит пересмотреть.

Наиболее распространенным и довольно эффективным приемом резервного копирования больших объемов данных является создание точек доверия в данных — порции сохранных данных, которые были проверены после того, как их пометили как неизменяемые, обычно по прошествии какого-то времени. Как только мы узнаем, что заданный профиль пользователя или транзакция уже зафиксированы и не изменятся, мы можем проверить их внутреннее состояние и создать подходящие для восстановления копии этих данных. Далее вы можете создавать инкрементные резервные копии, которые включают в себя только данные, измененные или добавленные с момента создания последней резервной копии.

Такой прием позволяет сократить время на создание резервных копий, сделав его пропорциональным времени обработки основной ветви. Это означает, что частое создание инкрементных копий может спасти вас от проверки и копирования данных, которые будут длиться 80 лет.

Однако помните, что нас интересует *восстановление*, а не сами резервные копии. Предположим, что мы создали полную резервную копию некоторых данных три года назад и с тех пор делали инкрементные резервные копии. При полном восстановлении данных будут последовательно обработаны более 1000 зависимых друг от друга инкрементных резервных копий. Каждая такая обработка повышает риск сбоя, не говоря уже о логистических трудностях планирования и стоимости выполнения этих задач.

Еще одним способом снижения фактического времени, которое требуется для копирования и проверки, является распределение нагрузки. Если вы хорошо сегментировали данные, у вас будет возможность запустить параллельно N задач, каждая из которых отвечает за копирование и проверку $1/N$ части данных. Это потребует предварительного продумывания и планирования дизайна схемы и физического развертывания данных, чтобы:

- корректно сбалансировать данные;
- гарантировать независимость каждого сегмента;
- избежать возникновения разногласий среди конкурирующих задач одного уровня.

Сочетая «горизонтальное» распараллеливание нагрузки и «вертикальное» разграничение содержимого резервных копий по времени, мы можем сократить длительность

восстановления на несколько порядков, получив вместо восьми десятилетий вполне приемлемую оперативность.

Третий уровень: раннее обнаружение

«Плохие» данные не бездействуют, они распространяются. Ссылки на отсутствующие или поврежденные данные копируются и разветвляются, и с каждым обновлением общее качество хранилища данных снижается. Последующие зависимые транзакции и потенциальные изменения формата данных с течением времени все больше усложняют процесс восстановления данных. Чем раньше вы узнаете о потере данных, тем проще будет восстановить их и тем полнее будет результат.

Сложности, с которыми сталкиваются разработчики облачных систем

В высокоскоростных окружениях облачные приложения и сервисы инфраструктуры сталкиваются с множеством трудностей, связанных с сохранностью данных. К ним относятся, например, следующие:

- ссылочная целостность между хранилищами данных;
- изменения схемы;
- старение кода;
- миграция данных с нулевым временем работы вхолостую;
- изменение точек взаимодействия с другими сервисами.

Если не прилагать никаких усилий для отслеживания возникающих связей данных, их качество со временем будет ухудшаться.

Зачастую новые разработчики облачных сервисов, выбирающие распределенный устойчивый API хранилища, например Megastore, делегируют заботу о сохранности данных приложения алгоритму достижения распределенного консенсуса, реализованному под API, например Paxos (см. главу 23). Разработчики заключают, что выбранный API сам по себе сможет поддерживать данные приложения в хорошей форме. В результате они объединяют все данные приложения в единое решение для хранения, которое гарантирует распределенную устойчивость, и избегают проблем, связанных со ссылочной целостностью, в обмен на сниженную производительность и/или возможность масштабирования.

Несмотря на то что такие алгоритмы в теории безупречны, их реализация зачастую усложняется «костылями» — оптимизациями и обоснованными предположениями. Например, в теории алгоритм Paxos игнорирует давшие сбой вычислительные узлы и продолжает работать до тех пор, пока имеется кворум работающих узлов. Однако на практике игнорирование давшего сбой узла может привести к тайм-аутам, повторным попыткам и применению других способов обработки сбоев, характерных для конкретной реализации Paxos [Chandra, 2007]. Как долго Paxos должен пытаться связаться с неотзывающимся узлом до его отключения по тайм-ауту? То, что конкретная машина дает сбой (возможно, периодически) определенным способом, в определенный момент и в определенном дата-центре, может вызвать непредсказуемое поведение. Чем больше масштаб приложения, тем чаще на него влияют такие противоречия. Если эта логика останется верной, даже будучи примененной к реализации

Paxos (так было у компании Google), она должна быть еще более верной для реализаций, устойчивых в конечном счете, наподобие Bigtable (что также было верно). Затронутые приложения не могут знать, что 100 % их данных в порядке, пока не выполнят проверку: доверяйте системам хранения, но проверяйте!

Эту проблему усугубляет то, что для восстановления после низкоуровневого повреждения данных или сценариев удаления мы должны восстанавливать разные наборы данных из разных точек восстановления с помощью разных резервных копий, при этом изменения кода и схемы могут сделать старые резервные копии неэффективными в высокоскоростном окружении.

Внеполосная проверка данных

Для того чтобы предотвратить ухудшение качества данных до того, как пользователь это увидит, а также чтобы обнаружить сценарии повреждения данных и их потери, прежде чем они станут невосстановимыми, нужна система проверок и уравновешивания внеполосных проверок как внутри каждого хранилища данных приложения, так и между ними.

Зачастую конвейеры для подтверждения данных реализуются как коллекции задач MapReduce или Hadoop. Такие конвейеры с опозданием добавляются в уже популярные и успешные сервисы. Иногда конвейеры применяются, когда сервис достигает предела масштабируемости и перестраивается с нуля. Мы создали валидаторы для каждой из этих ситуаций.

Переключение некоторых разработчиков на работу над конвейером по проверке данных может на короткое время снизить скорость разработки. Однако выделение инженерных

ресурсов для проверки данных позволяет другим разработчикам двигаться быстрее в долгосрочной перспективе, поскольку инженеры знают, что в этом случае меньше вероятность того, что баги, вызывающие повреждение данных, останутся незамеченными и попадут в производственные системы. Как и применение юнит-тестов на ранних этапах жизненного цикла приложения, наличие конвейера проверки данных приведет к общему ускорению разработки проектов.

Рассмотрим конкретный пример: сервис Gmail sports имеет некоторое количество валидаторов, каждый из которых обнаружил проблемы, связанные с сохранностью данных, в системе, находящейся в промышленной эксплуатации. Разработчиков сервиса успокаивает то, что эти ошибки, вносящие несоответствия в данные работающей системы, будут обнаружены в течение 24 часов, и они вздрагивают при мысли о том, чтобы запускать валидаторы данных реже чем один раз в день. Эти валидаторы наряду с модульным и регрессионным тестированием позволили разработчикам Gmail вносить изменения в реализацию производственного хранилища чаще чем раз в неделю.

Корректно реализовать внеполосную проверку данных непросто. Если проверка слишком строгая, то даже простые надлежащим образом внесенные изменения заставят ее дать сбой. В результате инженеры прекратят попытки проверять данные. Если проверка данных недостаточно строгая, то данные могут быть повреждены и пользователи заметят это. Для того чтобы найти верный баланс, проверяйте только постоянные величины, которые вызывают проблемы у пользователей.

Например, сервис Google Drive периодически проверяет, соответствует ли содержимое файла листингам в каталогах Drive. Если эти два элемента не соответствуют друг другу,

данные некоторых файлов будут утрачены — наступят катастрофические последствия. Разработчики инфраструктуры сервиса Drive приложили столько усилий к обеспечению целостности данных, что вдобавок к этому улучшили свои валидаторы так, чтобы они автоматически исправляли подобные несоответствия.

Эта мера безопасности в 2013 году превратила требующую срочного вмешательства ситуацию потери данных: «Свистать всех наверх! О боже мой, файлы пропадают!» в обычную: «Пойдемте домой, причину исправим в понедельник». Преобразуя неотложные ситуации в обычные, валидаторы помогают повысить моральный дух инженеров, качество жизни и предсказуемость.

Внеполосные валидаторы могут дорого обойтись в случае масштабирования. Для проверки значительной части вычислительных ресурсов сервиса Gmail потребуется целая группа ежедневно запускаемых валидаторов. Повышают эти расходы валидаторы, которые одновременно снижают частоту попаданий кэша на стороне сервера, уменьшая скорость ответа сервера пользователям. Для того чтобы сгладить последствия подобной потери обратной связи, сервис Gmail предоставляет набор рычагов управления для ограничения скорости своих валидаторов и периодически проводит их рефакторинг, чтобы снизить количество состязаний за диск. В ходе выполнения такого рефакторинга мы уменьшили количество состязаний за дисководы на 60 %. Несмотря на то что большая часть валидаторов для сервиса Gmail запускаются каждый день, нагрузка самого крупного из них разделена на 10–14 сегментов и ежедневно запускается проверка только одного сегмента.

Google Compute Storage — это еще один пример сложности обеспечения сохранности данных, которую привносит масштабирование. Когда внеполосные валидаторы не могут

завершить свою работу в течение дня, инженеры, отвечающие за сервис Compute Storage, должны придумать более эффективный способ проверять метаданные вместо того, чтобы использовать исключительно метод ручного перебора. По аналогии с восстановлением данных многоуровневая стратегия также может оказаться полезной при внеполосной проверке данных. По мере масштабирования сервиса пожертвуйте строгостью ежедневных валидаторов. Убедитесь, что валидаторы, работающие ежедневно, продолжают отлавливать наиболее аварийные сценарии в течение 24 часов, но выполняйте более строгие проверки с меньшей частотой, для того чтобы сдержать рост стоимости и уменьшить задержку.

Поиск проблем в давших сбой проверках может потребовать значительных усилий. Причины прерывания таких проверок могут исчезнуть через минуты, часы или дни. Поэтому способность быстро разобраться в журналах проверки очень важна. Продуманные и завершенные сервисы компании Google предоставляют дежурным инженерам полную документацию и инструменты для поиска проблем. Например, инженеры, обслуживающие сервис Gmail, имеют в своем распоряжении:

- набор рекомендаций, описывающих, как нужно реагировать на оповещение о сбое проверки данных;
- инструмент исследования проблемы наподобие BigQuery;
- информационную панель проверки данных.

Для эффективной внеполосной проверки данных требуется все нижеперечисленное:

- управление задачами проверки;

- мониторинг, оповещения и информационные панели;
- функциональность для ограничения уровня;
- инструменты для поиска проблем;
- производственные сценарии;
- API для проверки данных, которые позволяют легко добавлять валидаторы и выполнять их рефакторинг.

Бо'льшая часть маленьких команд инженеров, с высокой скоростью разрабатывающих функциональность, не может позволить себе одновременно разрабатывать, создавать и поддерживать все эти системы. Если им придется делать это, результатом окажется нестабильное, ограниченное и затратное одноразовое решение, которое быстро станет неремонтируемым. Поэтому вам следует структурировать свои команды инженеров так, чтобы основная команда, отвечающая за инфраструктуру, предоставила фреймворк проверки данных для остальных команд, занимающихся разработкой продукта. Основная команда, занимающаяся инфраструктурой, поддерживает фреймворк внеполосной проверки данных, а команды разработчиков продуктов поддерживают свою бизнес-логику, лежащую в основе валидатора, чтобы идти в ногу с развивающимися продуктами.

Знаем, что восстановление данных сработает

Когда лампочка перегорает? Тогда, когда щелчок на выключателе не приводит к включению света? Не всегда — зачастую лампочка уже перегорела, и вы замечаете это из-за

того, что щелчок на выключателе не дает результатов. К этому моменту в комнате темно, и вы ударяетесь мизинцем.

Аналогично зависимости для восстановления данных (в большинстве случаев имеются в виду резервные копии) могут быть неисправными, о чем вы не будете знать до того, как попробуете восстановить с их помощью данные.

Если вы обнаружите, что процесс восстановления данных не работает, до того, как понадобится выполнить восстановление, то сможете справиться с уязвимостью, прежде чем станете ее жертвой: использовать другую резервную копию, предоставить дополнительные ресурсы и изменить целевые значения SLO. Но для того чтобы предпринимать превентивные действия, следует понять, что их вообще нужно предпринимать. Для обнаружения этих уязвимостей необходимо:

- постоянно тестировать процесс восстановления данных наряду с другими повседневными операциями;
- настроить оповещения, которые срабатывают, когда процесс восстановления не может выдать контрольные сигналы, информирующие о его успехе.

Что может пойти не так с процессом восстановления? Все что угодно. И именно поэтому единственный тест, который позволит вам ночью спокойно спать, — это полный сквозной тест. Но лучше один раз увидеть, чем сто раз услышать. Даже если вы недавно успешно выполнили восстановление данных, некоторые части процесса восстановления все равно могут оказаться недействующими. Если из этой главы вы вынесете всего один урок, то пусть он будет следующим: *вы сможете узнать, что данные подлежат восстановлению, только когда восстановите их.*

Если тест восстановления данных — это действие, выполняемое вручную в несколько этапов, тестирование становится очень неприятной работой, которая выполняется либо недостаточно глубоко, либо недостаточно часто. Поэтому следует автоматизировать эти тесты, когда возможно, и постоянно выполнять их.

Существует множество аспектов плана восстановления, которые нуждаются в подтверждении.

- Являются ли резервные копии полными и корректными или же они пусты?
- Имеется ли у вас достаточное количество вычислительных ресурсов для того, чтобы выполнить всю настройку, восстановление и постобработку задач, из которых состоит процесс восстановления?
- Выполняется ли процесс восстановления данных за разумное время?
- Можно ли наблюдать за состоянием процесса восстановления данных по мере его выполнения?
- Свободны ли вы от критической зависимости от ресурсов, неподконтрольных вам, таких как доступ к хранилищу накопителей, которое находится за пределами площадки и к которому нельзя получать доступ все 24 часа 7 дней в неделю?

Тестирование выявило упомянутые ранее сбои, а также сбои многих других компонентов успешного восстановления данных. Если бы мы не обнаружили эти сбои в обычных тестах, то есть если бы столкнулись с ними в момент, когда

понадобилось бы восстановить данные в критический ситуации, вполне возможно, что наиболее успешные продукты компании Google не прошли бы проверку временем.

Сбои избежать нельзя. Если вы оттягиваете их обнаружение до момента, когда окажетесь перед лицом реальной потери данных, то вы играете с огнем. Если в результате тестирования сбои происходят до того, как случится реальная катастрофа, вы сможете исправить проблемы, прежде чем они вам навредят.

Примеры

Как вы и предполагали, жизнь обеспечила нам неприятные и неизбежные возможности протестировать процессы восстановления данных под давлением реального мира. В этой книге рассматриваются два наиболее заметных и интересных примера такого тестирования.

Gmail – февраль 2011 года: восстановление с помощью GTape

Первый пример, который мы рассмотрим, уникален в нескольких смыслах: потерю данных вызвало множество причин и нам пришлось много работать с последней линией обороны — системой GTape, предназначеннной для выполнения резервного копирования в режиме онлайн.

Воскресенье, 27 февраля 2011 года, поздний вечер

На пейджер инженеров, обслуживающих систему резервного копирования сервиса Gmail, поступило сообщение, которое содержало телефонный номер для участия в конференции. Произошло то, чего мы так долго опасались и что являлось основной причиной существования системы резервного

копирования, — сервис Gmail потерял значительный объем данных пользователей. Несмотря на множество мер безопасности, внутренних проверок и избыточных элементов, данные Gmail исчезли.

Это было первое крупномасштабное применение GTape — глобального сервиса резервного копирования для сервиса Gmail — для восстановления пользовательских данных. К счастью, мы выполняли такое восстановление не впервые, поскольку моделировали подобные ситуации много раз. Так что мы смогли сделать следующее:

- оценить, сколько времени потребуется для восстановления бо́льшей части пострадавших учетных записей;
- в течение нескольких часов восстановить все учетные записи, исходя из первоначальной оценки;
- восстановить свыше 99 % данных до того, как истечет запланированное время.

Была ли удачей способность сформулировать такую оценку? Нет — наш успех стал результатом планирования, применения оптимальных приемов и кооперации, и мы были рады увидеть, что наш вклад в каждый из этих элементов оправдал себя. Мы смогли своевременно восстановить данные, выполнив план, разработанный в соответствии с лучшими методами глубокой защиты и подготовки к неотложным ситуациям.

Когда компания Google публично заявила, что мы восстановили данные с помощью не афициированной ранее системы резервного копирования на носителе на магнитной ленте [Sloss, 2011], люди были удивлены. Накопители на ленте? Разве у компании Google нет множества дисков и быстрой сети для репликации таких важных данных? Конечно, у компании

Google есть такие ресурсы, но принцип глубокой защиты гласит, что мы должны обеспечить несколько слоев защиты для того, чтобы застраховаться от сбоя любого единственного механизма защиты.

Создание резервных копий онлайн для систем вроде Gmail обеспечивает глубокую защиту из двух слоев:

- при сбое внутренней избыточности сервиса Gmail и систем выполнения резервного копирования;
- при широкомасштабных сбоях или уязвимости нулевого дня в драйвере устройства или в файловой системе, которая влияет на лежащие в его основе медиаустройства (диск).

Этот сбой появился на основе первого сценария — несмотря на то что для сервиса Gmail имелись внутренние средства восстановления утраченных данных, эту потерю восстановить с их помощью не удалось.

Одними из наиболее почитаемых в нашей компании факторов восстановления данных сервиса Gmail были степень скооперированности и четкая координация процесса восстановления. Многие команды, часть из которых не были связаны с сервисом Gmail или восстановлением данных, взялись нам помочь. Восстановление не прошло бы успешно и гладко, если бы не существовало основного плана руководства такой сложной операцией. Этот план был продуктом регулярных репетиций и прогонов вхолостую.

Готовясь к ситуациям, требующим неотложного вмешательства, мы начали считать такие сбои неизбежными. Принимая эту неизбежность, мы не надеемся, что нам удастся избежать этих катастроф, а также можем прогнозировать их появление. Поэтому нам нужен план того, как справляться не

только с ожидаемыми сбоями, но и с каким-то количеством случайных универсальных сбоев.

Короче говоря, мы всегда знали, что следует использовать лишь наилучшие методы, и было приятно увидеть, что это правило оказалось верным.

Google Music – март 2012 года: определение бесконтрольного удаления данных

Второй сбой, который мы рассмотрим, повлек за собой трудности в логистике, уникальные для восстанавливаемого хранилища данных: где вы храните более 5000 лент и как эффективно (или даже реалистично) считываете такое количество данных с онлайн-носителей в разумный промежуток времени?

Вторник, 6 марта 2012 года, вторая половина дня

Обнаружение проблемы. Пользователь сервиса Google Music сообщает о том, что треки, которые ранее были проблемными, теперь оказываются пропущенными. Команда, ответственная за взаимодействие с пользователями сервиса Google Music, оповещает инженеров. Проблему рассматривают как возможный сбой потоковой передачи мультимедиа.

Седьмого марта инженер, который исследовал проблему, обнаружил, что у метаданных невоспроизводимых треков отсутствуют ссылки, которые должны указывать на данные самой песни. Он удивлен. Очевидным решением является поиск аудиоданных и восстановление ссылок на них.

Однако инженеры компании Google гордятся своей культурой устранения основных причин проблемы, поэтому этот инженер копнул глубже. Когда он обнаружил причину

нарушения сохранности данных, у него чуть не случился сердечный приступ: ссылка на аудиоданные была удалена защищающим приватность конвейером по удалению данных. Эта часть сервиса Google Music была разработана для того, чтобы удалять большие количества аудиозаписей в рекордные сроки.

Оценка ущерба. Политика конфиденциальности компании Google защищает личные данные пользователя. Для сервиса Google Music это означает, что файлы музыки и релевантные метаданные удаляются в течение разумного промежутка времени после того, как их удаляет пользователь. По мере взлета популярности сервиса Google Music объем данных быстро увеличивался, поэтому в 2012 году оригинальная реализация должна была быть перепроектирована для повышения эффективности. Шестого февраля обновленный конвейер удаления данных был запущен в первый раз и удалил релевантные метаданные.

В тот момент показалось, что все в порядке, поэтому мы позволили на следующем шаге удалить также связанные с этими метаданными аудиоданные.

Мог ли оказаться реальностью худший кошмар инженера? Работник немедленно забил тревогу, повысил приоритет этого случая до максимально возможного и сообщил о проблеме менеджерам и службе SRE. Небольшая команда разработчиков Google Music и SR-инженеры собрались для того, чтобы решить проблему, и конвейер-нарушитель был временно отключен для того, чтобы не увеличить количество пострадавших пользователей.

Проверить вручную метаданные миллиардов файлов, расположенных в нескольких data-центрах, было бы невозможно. Поэтому команда создала быструю задачу

MapReduce для оценки урона и в отчаянии ожидала ее завершения.

Восьмого марта инженеры получили результат и обмерли: процесс рефакторинга конвейер удаления данных «убил» примерно 600 000 ссылок на аудиоданные, которые не нужно было трогать, а это повлияло на аудиофайлы 21 000 пользователей. Поскольку быстрый конвейер диагностики сделал несколько упрощений, реальный масштаб урона оказался намного меньше.

Прошло больше месяца с тех пор, как конвейер удаления данных был запущен в первый раз и в итоге удалил сотни тысяч аудиозаписей, которые не должен был трогать. Можно ли было надеяться на то, что мы восстановим данные? Если бы треки не были восстановлены или были восстановлены недостаточно быстро, компании Google пришлось бы скрыть музыку от пользователей. Как мы могли не заметить такой сбоя?

Решение проблемы. Одновременные идентификация ошибки и попытки восстановления.

Первый шаг решения проблемы заключался в том, чтобы идентифицировать сбой и определить, почему это произошло. До тех пор пока основная причина не определена и не исправлена, любые попытки восстановления данных были бы тщетными. Нам пришлось бы снова запустить конвейер, чтобы удовлетворить запросы пользователей, которые удалили свои аудиотреки, но это навредило бы ни в чем не повинным пользователям, которые продолжали бы терять купленную музыку или, что еще хуже, собственные кропотливо записанные аудиофайлы. Единственным способом избежать уловки-22¹⁶¹ было быстрое исправление основной причины сбоя.

Мы не могли терять время до запуска процесса восстановления. Сами аудиотреки были сохранены на носители на магнитной ленте, но в отличие от примера с сервисом Gmail, носители с резервными копиями для сервиса Google Music были вывезены в онлайн-хранилища, поскольку это позволяло отвести больше места для объемных резервных копий пользовательских аудиоданных. Для того чтобы быстро восстановить возможность работать с сервисом для пострадавших пользователей, команда решила найти основную причину проблемы и параллельно получить расположенные за пределами площадки накопители (довольно затратный по времени вариант восстановления).

Инженеры разбились на две группы. Наиболее опытные SR-инженеры работали над восстановлением данных, а разработчики анализировали код удаления данных и пытались исправить ошибку, связанную с потерей данных. Из-за того что полного понимания основной проблемы не было, восстановление пришлось проводить в несколько этапов. Был определен первый пакет, состоящий из примерно полумиллиона аудиотреков, и в 4:54 пополудни по тихоокеанскому времени 8 марта команда, которая отвечала за систему восстановления данных с помощью носителей на магнитной ленте, была оповещена о попытке восстановления данных.

На команду восстановления работал один фактор: процесс восстановления проходил спустя всего лишь несколько недель после проведения внутри компании ежегодной тренировки по восстановлению данных (см. [Krishan, 2012]). Команда восстановления с носителей на магнитной ленте уже знала о возможностях и ограничениях их подсистем, которые были субъектами тестирования Dirt, и начала сдувать пыль с нового инструмента, который протестировала во время тренировки.

Используя этот инструмент, объединенная команда восстановления начала кропотливый процесс проверки сотен тысяч аудиозаписей.

Таким образом команда определила, что процесс начального восстановления потребует доставки более чем 5000 носителей на грузовике. После этого техники дата-центра должны будут освободить место для лент в соответствующих библиотеках. Затем начнется долгий и сложный процесс регистрации лент и извлечения данных с них, что потребует, в частности, использования обходных путей и сглаживания последствий на случай, если ленты или диски окажутся плохими.

К сожалению, в резервных копиях были найдены только 436 223 из примерно 600 000 потерянных аудиозаписей, что означало, что около 161 000 аудиозаписей были «съедены» еще до того, как могла быть сделана их резервная копия. Команда восстановления решила определить, как можно было бы восстановить 161 000 отсутствующих аудиозаписей, после того как был запущен процесс восстановления аудиозаписей, для которых имелись резервные копии.

В то же время команда, которая искала основную причину, сделала ложное заключение, а затем отказалась от него: сначала они думали, что сервис хранения, от которого зависел сервис Google Music, предоставил данные, содержащие ошибку, что заставило конвейер удалить неверные данные. Более тщательное расследование показало, что эта теория неверна. Члены команды, искавшие основную причину, почесали голову и продолжили искать неуловимую ошибку.

Первый этап восстановления. Как только команда, отвечавшая за восстановление, определила нужные резервные копии, а это произошло 8 марта, была предпринята первая попытка восстановления. Запрос 1,5 Пбайт данных,

распределенных между тысячами лент из хранилища, — это одна проблема, но извлечение данных с лент — совершенно другая. Наш стек для работы с резервными копиями, записанными на ленты, не был спроектирован так, чтобы обработать одну операцию по восстановлению такого масштаба, поэтому пришлось разбить эту задачу на 5475 задач поменьше. Для выполнения такого количества операций по восстановлению от человека-оператора потребовалось бы вводить по одной команде каждую минуту на протяжении трех дней, и, несомненно, он допустил бы множество ошибок. Только для того чтобы запросить восстановление с помощью системы резервного копирования, SR-инженерам пришлось разрабатывать программное решение^{[162](#)}.

К полуночи 9 марта SR-инженеры закончили запрашивать выполнение всех 5475 операций восстановления. Система резервного копирования начала колдовать над восстановлением. Четыре часа спустя она выдала список из 5337 резервных копий, которые нужно было вернуть из мест их размещения вне дата-центра. Еще через 8 часов ленты прибыли в дата-центр на грузовиках.

Пока машины были в дороге, техники дата-центра отключили от обслуживания несколько библиотек, работающих с лентами, и удалили тысячи лент, чтобы освободить место для масштабной операции по восстановлению данных. Когда ранним утром тысячи лент прибыли, техники начали кропотливо загружать их вручную. Проведенные ранее тренировки с помощью DiRT показали, что этот ручной процесс при выполнении крупных операций в сотни раз быстрее, чем основанные на использовании роботов методы, которые приняты у поставщиков библиотек работы с лентами. В течение 3 часов библиотеки снова были включены и начали сканировать ленты и выполнять тысячи операций

восстановления для распределенного вычислительного хранилища.

Несмотря на то что команда уже имела опыт работы с DiRT, масштабное восстановление 1,5 Пбайт данных продлилось больше двух дней, которые мы на него отводили. К утру 10 марта только 74 % от 436 223 аудиофайлов были успешно переведены из 3475 полученных резервных копий в распределенное хранилище файловой системы в ближайшем компьютерном кластере. Еще 1862 резервные копии были пропущены поставщиком. Вдобавок процесс восстановления приостановился из-за 17 плохих лент. В ожидании сбоя, связанного с плохими лентами, для создания резервных копий файлов была использована избыточная кодировка. В ходе дополнительной доставки на грузовиках были переданы лишние ленты вместе с 1862 лентами, которые были пропущены в первой партии.

К утру 11 марта операция восстановления была завершена более чем на 99,95 % и выполнялся возврат дополнительных лент для оставшихся файлов. Несмотря на то что данные уже находились в распределенных файловых системах, необходимо было предпринять еще несколько шагов для того, чтобы сделать их доступными пользователям. Команда Google Music начала осуществлять эти шаги восстановления данных параллельно для небольшого фрагмента восстановленных аудиофайлов, чтобы убедиться, что процесс выполняется в соответствии с ожиданиями.

В этот момент пейджеры команды Google Music сообщили о не связанном с предыдущими событиями критическом сбое, который влиял на пользователей, — команда потратила еще на два дня на его устранение. Процесс восстановления данных продолжился 13 марта, когда все 436 223 аудиозаписи снова стали доступными для пользователей. Почти за семь дней 1,5

Пбайт данных были восстановлены с помощью резервных копий, пять дней из них занял сам процесс восстановления данных.

Второй этап восстановления. Когда первая волна процесса восстановления данных прошла, команда переключилась на 161 000 отсутствующих аудиофайлов, которые были удалены из-за ошибки еще до того, как мы смогли сделать их резервные копии. Бо́льшая часть этих файлов была куплена в магазинах и являлась рекламными треками, а оригинальные копии в магазинах не были затронуты ошибкой.

Эти треки были быстро восстановлены, и пользователи снова могли наслаждаться музыкой.

Однако небольшую часть 161 000 аудиофайлов пользователи загрузили самостоятельно. Команда Google Music дала задание своим серверам попросить клиентов перезагрузить файлы 14 марта и позже. Данный процесс занял больше недели. Этим завершился полный процесс восстановления после инцидента.

Решение основной проблемы. В итоге команда разработчиков Google Music определила, в чем заключается недостаток конвейера удаления данных. Для того чтобы это понять, вы сперва должны узнать, как офлайн-системы обработки данных развиваются в больших масштабах.

Для больших и сложных сервисов, состоящих из нескольких подсистем и сервисов хранения, даже такая простая задача, как уничтожение удаленных данных, должна выполняться в несколько этапов, каждый из которых задействует разные хранилища данных.

Для того чтобы обработка данных завершалась быстро, ее распараллеливают между десятками тысяч машин, которые распределяют большую нагрузку на разные подсистемы. Это может замедлить работу сервиса для пользователей или

спровоцировать сбой при большой нагрузке. Чтобы избежать этих нежелательных сценариев, инженеры, занимающиеся облачными вычислениями, зачастую делают недолговременную копию данных во вторичном хранилище, где затем выполняется обработка данных. Если относительный возраст вторичных копий не координируется, такая практика приводит к состоянию гонки.

Например, два этапа конвейера могут быть спроектированы так, чтобы работать строго друг за другом с промежутком 3 часа, чтобы второй этап мог сделать упрощающее допущение о корректности своих входных данных. Без этого допущения логику второго этапа может быть трудно распараллелить. Но по мере роста объемов данных выполнение этапов может занимать больше времени. В итоге предположения, сделанные для оригинального проекта, могут оказаться неверными для некоторых фрагментов данных, необходимых на втором этапе.

Поначалу гонки могут происходить для небольших фрагментов данных. Но по мере увеличения объема данных они могут затрагивать все большие и большие фрагменты данных. Такой сценарий является вероятным — конвейер работает корректно для большей части данных большую часть времени. Когда случаются гонки в конвейере удаления данных, случайно могут быть удалены неверные данные.

Конвейер удаления данных для Google Music был разработан с учетом возможных ошибок. Но когда по мере роста сервиса предшествующие этапы конвейера начали требовать большего количества времени, мы выполнили оптимизацию производительности для того, чтобы сервис Google Music мог продолжить соответствовать требованиям к приватности. В результате вероятность непреднамеренно удаляющего данные состояния гонки в этом конвейере начала повышаться. Когда конвейер прошел рефакторинг, она снова

значительно возросла до точки, когда гонки стали происходить более регулярно.

По результатам процесса восстановления команда разработчиков Google Music перепроектировала конвейер обработки данных и избавилась от гонок. К тому же мы улучшили наблюдение за производственной средой и настроили системы оповещения, чтобы обнаруживать аналогичные крупномасштабные бесконтрольные ошибки, связанные с удалением данных, и исправлять такие проблемы до того, как пользователи их заметят^{[163](#)}.

Общие принципы SRE, применяемые для сохранности данных

Общие принципы SRE могут быть применены в области сохранности данных и облачных вычислений, как показано в этом разделе.

Мышление новичка

В крупномасштабных сложных системах имеются заложенные изначально ошибки, которые невозможно исследовать полностью. Никогда не думайте, что вы разбираетесь в работе крупной системы достаточно хорошо для того, чтобы сказать, что она не даст определенный сбой. Доверяйте, но проверяйте и применяйте глубокую защиту.

Доверяй, но проверяй

Любой API, от которого вы зависите, не будет *все время* идеально работать. Учитывайте, что независимо от качества вашей инженерной работы или тестирования API будет иметь

дефекты. Проверяйте корректность наиболее критических элементов данных, применяя внеполосные валидаторы, даже если семантика API предполагает, что этого делать не нужно. Идеальные алгоритмы могут не иметь идеальной реализации.

Надежда – плохая стратегия

Системные компоненты, которые используются непостоянно, дадут сбой в тот момент, когда они нужнее всего. Докажите, что процесс восстановления данных работает, регулярного применения его, или же он просто не будет функционировать. Людям не хватает терпения постоянно тестировать системные компоненты, поэтому в данном вопросе автоматизация — ваш друг. Однако, когда вы нанимаете для написания решения по автоматизации инженера, имеющего конкурирующие приоритеты, вам может понадобиться найти ему временную замену.

Глубокая защита

Даже наиболее устойчивая система подвержена воздействию багов и ошибок операторов. Для того чтобы можно было исправить проблемы, связанные с сохранностью данных, сервисы должны быстро определять такие проблемы. Любая стратегия непременно даст сбой в изменяющихся средах. Наилучшими стратегиями сохранения данных являются многоуровневые стратегии — несколько стратегий, которые используются совместно, решают широкий диапазон сценариев и имеют разумную стоимость.

Возвращайтесь и перепроверяйте

То, что данные были в порядке вчера, никак не поможет вам завтра и даже сегодня. Системы и инфраструктуры меняются, и нужно доказывать, что ваши предположения и процессы остаются релевантными перед лицом прогресса. Подумайте о следующем.

Сервис Shakespeare получил высокие оценки в прессе, и число его пользователей постепенно увеличивается. Но при сборке сервиса не было уделено достаточно внимания сохранности данных. Конечно, мы не хотим, чтобы сервис поставлял *плохие* биты, но если индекс для Bigtable будет потерян, мы легко сможем восстановить его с помощью оригинальных текстов Шекспира и MapReduce. Это займет совсем немного времени, поэтому мы никогда не делаем резервных копий индекса.

Теперь рассмотрим ситуацию, когда новая функциональность позволит пользователям создавать текстовые аннотации. Оказывается, наше множество данных не может быть с легкостью восстановлено, при этом ценность пользовательских данных будет постоянно повышаться. Поэтому нужно пересмотреть варианты репликации — мы выполняем ее не только для того, чтобы улучшить показатели задержки и полосы пропускания, но и чтобы обеспечить сохранность данных. Эта процедура также периодически тестируется на практических занятиях по использованию DiRT для того, чтобы гарантировать, что мы можем восстановить пользовательские аннотации из резервных копий ко времени, указанному в SLO.

Итоги главы

Обеспечение доступности данных — важнейшая задача любой системы, основанной на работе с данными. Не зацикливаясь на одних лишь инструментах, SR-инженеры считают полезным позаимствовать также идеи разработки, управляемой тестированием, и убедиться, что наши системы действительно смогут обеспечить доступность данных с прогнозируемым значением времени отключения не хуже заданного. Методы и средства, которые мы используем для достижения этой цели, являются необходимым злом. Сосредоточившись на этой цели, мы избегаем ловушки, когда «операция была выполнена успешно, но система погибла». Понимание того, что «не так» может пойти не «что-то», а сразу все, — это большой шаг навстречу подготовке к реальной чрезвычайной ситуации. Сценарии всех возможных комбинаций катастроф и планы реагирования на них позволяют вам крепко спать как минимум одну ночь, поддерживаемые в актуальном состоянии планы восстановления позволяют вам спать и остальные 364 ночи в году.

По мере того как вы научитесь восстанавливаться после любого сбоя за разумное время N , найдите способы сократить его за счет более быстрого и качественного выявления потерь, поставив перед собой цель приблизиться к значению $N = 0$. Далее можете переключиться с планирования восстановления на планирование предотвращения проблем, стремясь достичь Святого Грааля всех данных всех времен. Добейтесь этой цели — и сможете спать на пляже во время заслуженного отпуска.

[152](#) Atomicity, Consistency, Isolation, Durability — атомарность, согласованность, изолированность, долговечность, см. <https://en.wikipedia.org/wiki/ACID>. Базы данных SQL, такие как MySQL и PostgreSQL, стремятся соответствовать этим свойствам.

[153](#) Basically Available, Soft state, Eventual consistency — доступность в большинстве случаев, неустойчивое состояние, согласованность в конечном счете, см.

https://en.wikipedia.org/wiki/Eventual_consistency. Системы, основанные на семантике BASE, наподобие Bigtable или Megastore, зачастую описываются как NoSQL.

[154](#) Для получения более полной информации о семантиках ACID и BASE обратитесь к [Golab, 2014] и [Bailis, 2013].

[155](#) Binary Large Object — двоичный большой объект, см.
https://en.wikipedia.org/wiki/Binary_large_object.

[156](#) См. [https://en.wikipedia.org/wiki/Zero-day_\(computing\)](https://en.wikipedia.org/wiki/Zero-day_(computing)).

[157](#) Глиняные таблички — это старейший из известных примеров письма. Для того чтобы получить более подробную информацию о сохранении данных на длительный срок, см. [Conway, 1996].

[158](#) Во время чтения этого совета кто-то может задаться вопросом: если вам нужно предоставить API на базе хранилища данных для того, чтобы реализовать мягкое удаление, зачем ограничивать себя только им, если можно обеспечить обширную функциональность, которая поможет защитить данные от случайного удаления пользователями? В качестве примера приведем реальную ситуацию, произошедшую в компании Google, и рассмотрим Blobstore: вместо того чтобы позволить клиентам непосредственно удалять данные и метаданные, API для Blob реализовал широкую функциональность для безопасности, включая политику создания резервных копий (оффлайн-реплики), сквозные контрольные суммы и периоды существования по умолчанию (мягкое удаление). Оказалось, что во многих случаях мягкое удаление спасало клиентов Blobstore от потери данных, которая могла все усложнить. Конечно, обширную функциональность для защиты от удаления стоит реализовать, но для компаний, имеющих конкретные сроки удаления данных, мягкое удаление — это наиболее подходящая защита от ошибок и случайного удаления, как и в случае Blobstore.

[159](#) Снимок в этом контексте является статическим экземпляром хранилища, предназначенным только для чтения, например снимок базы данных SQL. Снимки зачастую реализуются с помощью семантики «копирования при записи» для эффективности хранения. Они могут быть дорогими по двум причинам: во-первых, они соперничают за работу с той же производительностью, что и работающие хранилища данных, и во-вторых, чем быстрее изменяются данные, тем менее эффективным будет копирование при записи.

[160](#) Чтобы получить более подробную информацию о репликации GFS, см. [Ghemawat, 2003]. Для получения более подробной информации о кодах Рида — Соломона см. https://en.wikipedia.org/wiki/Reed–Solomon_error_correction.

[161](#) См. [http://en.wikipedia.org/wiki/Catch-22_\(logic\)](http://en.wikipedia.org/wiki/Catch-22_(logic)).

[162](#) На практике создать программное решение было нетрудно, поскольку большая часть SR-инженеров имеет опыт разработки ПО. Необходимость такого опыта значительно усложняет процесс поиска и найма SR-инженеров, и теперь вы наконец понимаете, почему в службу SRE нанимают практикующих инженеров-программистов; см. [Jones, 2015].

[163](#) Из нашего опыта, инженеры, занимающиеся облачными вычислениями, зачастую не хотят настраивать производственные оповещения для удаления данных из-за естественных отклонений в количестве удаляемых пользователями данных. Но поскольку такие оповещения предназначены для определения глобального, а не локального удаления данных, полезнее будет оповещать о том, что выполняется удаление большого количества данных, собранных для всех пользователей, которое превышает определенную границу (например, 10x от наблюдаемого 95-го процентиля), чем рассылать менее полезные уведомления об удалении данных каждому пользователю индивидуально.

27. Надежный масштабируемый выпуск продукта

Авторы — Рандив Синг и Себастьян Кирш при участии Вивека Рая

Под редакцией Бетси Байер

Интернет-компании наподобие Google могут запускать новые продукты и функциональность гораздо быстрее, чем традиционные компании. Роль SR-инженеров заключается в том, чтобы гарантировать быстрый темп внедрения изменений, не нарушая стабильности сайта. Мы создали особую команду инженеров, координирующих запуск, для того чтобы ее члены консультировали другие команды по вопросам технических аспектов успешного запуска.

Эти специалисты создали «список для запуска», в который включили распространенные вопросы, связанные с запуском, а также способы решения часто возникающих проблем. Этот список оказался полезным инструментом, позволившим нам гарантировать, что надежные запуски будут происходить регулярно.

Рассмотрим обычный сервис компании Google, например Keyhole, который доставляет изображения со спутника для Google Maps и Google Earth. В обычный день Keyhole выдает до нескольких тысяч изображений в секунду. Но в канун Рождества 2011 года трафик увеличился в 25 раз — до миллиона запросов в секунду. Что же вызвало такой всплеск?

Прибытие Санта-Клауса.

Несколько лет назад компания Google работала вместе с NORAD (North American Aerospace Defense Command — Командование воздушно-космической обороны Северной Америки) с целью размещения сайта на рождественскую тематику, который отслеживал бы перемещение Санта-Клауса по миру, позволяя пользователям смотреть, как он разносит подарки в реальном времени. Частью задумки был «виртуальный воздушный парад», который включал в себя изображения со спутника, позволявшие отследить перемещение Санты по симулированному миру.

Несмотря на то что проект NORAD Tracks Santa мог показаться экстравагантным, он имел все характеристики сложного и рискованного запуска: жесткий дедлайн (компания не могла попросить Санту прибыть на неделю позже, если бы сайт был не готов), большая публичность, миллионная аудитория и очень резкое увеличение трафика (все хотели посетить сайт в канун Рождства). Никогда не недооценивайте силу миллионов детей, жаждущих подарков, — этот проект вполне мог поставить серверы компании Google на колени.

Команда SRE-инженеров хорошо поработала для того, чтобы подготовить нашу инфраструктуру к этому запуску, гарантируя, что Санта доставит все свои подарки вовремя под пристальным взглядом аудитории. Последнее, чего мы хотели, — это заставить детей плакать из-за того, что они не смогли посмотреть, как Санта разносит подарки. Фактически мы продублировали различные аварийные выключатели, встроенные в сайт, чтобы защитить наши серверы, — «выключатели, заставляющие детей плакать». Предвидя, что запуск проекта может пойти не так множеством способов и координируя разные группы инженеров, вовлеченные работу над ним, мы создали внутри службы SRE особую команду

инженеров — координаторов запуска (Launch Coordination Engineers, LCE).

Запуск новых продукта или функциональности — это момент истины для каждой компании, точка, когда месяцы или годы труда будут представлены миру. Традиционные компании довольно редко запускают новые продукты. Цикл запуска продуктов интернет-компаниями совершенно иной. Запуски и быстрые итерации им выполнять гораздо проще, поскольку новая функциональность может появиться на стороне сервера, а не на рабочих станциях отдельных клиентов.

Компания Google определяет как запуск создание любого кода, который вносит в приложение заметные извне изменения. В зависимости от характеристик — комбинации атрибутов, времени, количества шагов и сложности — процессы запуска могут значительно отличаться друг от друга. Так, компания Google иногда выполняет до 70 запусков в неделю.

Такое количество изменений является как обоснованием, так и причиной создания упрощенного процесса запуска. Компания, которая запускает новый продукт раз в три года, не нуждается в детально проработанном алгоритме запуска. Ко времени нового запуска большинство компонентов ранее разработанной процедуры устареют. Традиционные компании даже не имеют возможности разработать детализированный процесс запуска, поскольку им не хватает для этого опыта.

Управление запусками

Хорошие инженеры-программисты имеют большой опыт разработки и проектирования и очень хорошо понимают технологию собственных продуктов. Однако те же инженеры

могут быть не знакомы с вызовами и ловушками запуска продукта для миллиона пользователей, когда требуется одновременно минимизировать отключения и максимизировать производительность.

Компания Google отвечает на вызовы, связанные с запусками, созданием внутри службы SRE специальной консультационной команды, которая отвечает за техническую сторону запуска нового продукта или функциональности. Эта команда состоит из программных и системных инженеров, имеющих опыт работы в других командах SRE, и специализируется на помощи разработчикам в создании надежных и быстрых продуктов, соответствующих стандартам компании Google по устойчивости, масштабированию и надежности. Эта консультационная команда, состоящая из инженеров – координаторов запуска (Launch Coordination Engineering, LCE), придает процессу запуска плавность несколькими способами. Входящие в нее специалисты:

- проверяют продукты и сервисы на соответствие стандартам надежности компании Google и практическим рекомендациям, а также советуют, как увеличить надежность;
- действуют как посредники между несколькими командами, участвующими в запуске;
- управляют техническими аспектами запуска, гарантируя, что задачи будут выполняться с нужной скоростью;
- действуют как шлюз, выполняя запуски, определенные как безопасные;
- обучают разработчиков лучшим приемам и способам интеграции с другими сервисами компании Google,

обеспечивая их внутренней документацией и практическими упражнениями для ускорения обучения.

Члены команды LCE проверяют сервисы несколько раз на протяжении их жизненного цикла. Большая часть проверок нового продукта или сервиса выполняется до запуска. Если команда разработчиков продукта выполняет запуск без поддержки SR-инженеров, LCE делятся своими знаниями для того, чтобы гарантировать спокойный запуск. Но даже команды запуска продуктов, уже имеющие сильную поддержку SR-инженеров, зачастую привлекают команду LCE во время критических запусков.

Вызовы, с которыми сталкиваются команды во время запуска нового продукта, значительно отличаются от повседневного обслуживания надежного сервиса (задача, в выполнении которой SR-инженеры уже преуспели), и команда LCE может воспользоваться опытом, приобретенным во время сотен других запусков. Команда LCE также упрощает проверки сервиса, когда новый сервис впервые попадает в руки службы SRE.

Роль инженера – координатора запуска

Наша команда координации запусков состоит из инженеров – координаторов запуска (LCE), которые либо были наняты непосредственно на эту позицию, либо перешли из службы SRE, так как имели опыт запуска сервисов Google. К LCE предъявляют те же требования, что и к SR-инженерам, они также должны иметь значительные навыки общения, взаимодействия и лидерства – LCE объединяют разрозненные группы для того, чтобы работать над общей целью, слаживают

случайные конфликты, а также направляют и обучают коллег-инженеров.

Наличие команды, координирующей процесс запуска, дает такие преимущества.

- *Широта опыта.* Поскольку команда работает с большим количеством продуктов, ее члены действуют во всех областях деятельности Google. Обширное знание продуктов и поддержание отношений с другими командами компании позволяют LCE легко распространять знания.
- *Кросс-функциональная перспектива.* LCE имеют целостное представление о запуске, что позволяет им координировать разрозненные команды в подразделениях SRE, разработки и управления продуктом. Такой подход особенно важен для сложных запусков, в которых могут быть задействованы полдюжины команд, работающих в разных часовых поясах.
- *Объективность.* Как беспристрастный советник, LCE играет роль балансировщика и посредника между заинтересованными лицами, включая SR-инженеров, разработчиков продуктов и маркетологов.

Поскольку роль инженера — координатора запуска — это роль SR-инженера, LCE также отдают приоритет надежности. Компания, которая не разделяет целей компании Google в области надежности, но стремится к быстрым изменениям, может выбрать другой набор стимулов.

Настройка процесса запуска

Компания Google улучшала процесс запуска более 10 лет. Со временем мы определили несколько критериев, которые

характеризуют удачный запуск.

- *Упрощенность.* Процесс прост для разработчиков.
- *Продуманность.* Во время запуска обнаруживаются очевидные ошибки.
- *Доскональность.* При запуске стабильно решаются наиболее важные проблемы.
- *Масштабируемость.* Выполняется как большое количество простых запусков, так и небольшое количество сложных.
- *Адаптируемость.* Процесс наложен как для распространенных видов запусков (например, добавление нового языка интерфейса для продукта), так и для новых (например, первый запуск Google Chrome или Google Fiber).

Как видите, некоторые из этих требований вступают в очевидный конфликт друг с другом. Например, очень сложно разработать процесс, который одновременно является упрощенным и доскональным. Балансировать этих требования относительно друг друга приходится долго. Компания Google успешно использует несколько тактик, чтобы соответствовать этим критериям.

- *Простота.* Усвойте основы. Не планируйте свои действия для каждого возможного случая.
- *Персонализированный подход.* Опытные инженеры настраивают процесс так, чтобы он подходил для каждого запуска.

- *Быстрые распространенные методы.* Идентифицируйте классы запусков, которые всегда выполняются по конкретной схеме (например, запуск продукта в новой стране), и обеспечьте для них упрощенный процесс запуска.

Практика показывает, что инженеры стремятся избегать процессов, которые считают слишком обременительными или не несущими пользы, особенно когда команда уже работает в кризисном режиме. По этой причине LCE должны постоянно оптимизировать процесс запуска, чтобы найти верный баланс между стоимостью и пользой.

Чек-лист для запуска

Чек-листы применяются для того, чтобы уменьшить количество сбоев и гарантировать стабильность и завершенность процессов в различных сферах деятельности. Распространенные примеры — карты контрольных проверок перед полетом и хирургические чек-листы [Gawande, 2009]. Аналогично LCE используют чек-лист (пример приведен в приложении Д) для оценки запуска. Этот документ помогает LCE оценить процесс запуска и предлагает запускающей команде варианты действий и ссылки на более подробную информацию. Рассмотрим примеры элементов, которые могут входить в чек-лист.

- **Вопрос:** нужно ли вам новое доменное имя?

Действие: поговорите с представителями службы маркетинга о том, каким должно быть желаемое доменное имя, и запросите регистрацию домена. Вот ссылка на форму маркетинга.

- **Вопрос:** сохраняете ли вы устойчивые данные?

Действия: реализуйте ограничение скорости и квоты. Используйте следующий разделяемый сервис.

На практике о любой системе можно задать практически бесконечное количество вопросов, и чек-лист запросто может разрастись до неконтролируемых размеров. Для того чтобы нагрузка на разработчиков оставалась управляемой, потребуется тщательно вести чек-лист. Сдержать его рост помогает то, что с какого-то момента добавление в него новых вопросов требует подтверждения вице-президента. Служба LCE при этом руководствуется следующими соображениями.

- Важность каждого вопроса нужно ставить под сомнение, особенно если предыдущий запуск оказался провальным.
- Каждая инструкция должна быть конкретной, практической и разумной, чтобы разработчики могли ее выполнить.

Вы должны постоянно работать с чек-листом, чтобы он оставался релевантным и актуальным, так как рекомендации с течением времени меняются, одни внутренние системы заменяются другими, а проблемы, актуальные для предыдущих запусков, устаревают из-за введения новых политик и процессов. LCE постоянно курируют чек-лист и вносят в него небольшие обновления, когда члены команды замечают элементы, которые следует изменить. Один-два раза в год член команды анализирует чек-лист в целом, чтобы найти устаревшие области, а затем вместе с владельцами сервиса и экспертами в предметной области проводит модернизацию пунктов чек-листа.

Сходимость и простота

В крупных организациях инженеры могут не представлять себе всей инфраструктуры, участвующей в выполнении распространенных задач, например ограничении скорости. Из-за этого они, скорее всего, будут повторно реализовывать существующие решения. Формирование набора общих инфраструктурных библиотек помогает избежать этой ситуации, а также приносит пользу компании: не затрачивается время на ненужную работу, знания становятся лучше переносимыми между сервисами, и это обеспечивает более высокий уровень разработки и качества сервиса благодаря тому, что усилия концентрируются на инфраструктуре.

Почти все группы в компании Google участвуют в распространенных процессах запуска, что делает чек-лист для запуска отличным инструментом улучшения сходимости для распространенной инфраструктуры. LCE могут порекомендовать вместо реализации собственного решения использовать существующую инфраструктуру как кирпичики — инфраструктуру, которая закалена годами применения и может снизить риски, связанные с производительностью или масштабируемостью. Примером является широко распространенная инфраструктура для ограничения скорости или пользовательских квот, отправки новых данных на серверы или выпуска новых версий бинарного файла. Такой тип стандартизации помогает радикально упростить чек-лист для запуска: например, обширные разделы чек-листа, связанные с выполнением требований для ограничения скорости, могут быть заменены одной строкой «Реализовать ограничение скорости с помощью системы X».

Благодаря хорошему знанию всех продуктов компании Google LCE находятся в уникальной позиции, которая

позволяет им находить возможности для упрощения. Во время работы над запуском они первыми замечают камни преткновения: какие части процесса запуска даются труднее всего, какие шаги занимают неприемлемое время, а также выявляют участки, где не хватает широко применяемой инфраструктуры. У LCE есть несколько способов упростить процесс запуска, так что они выступают в роли защитников запускающих команд. Например, LCE могут работать с владельцами особенно сложного процесса подтверждения и реализовать автоматические подтверждения для распространенных случаев. Они также могут отправлять сообщения о болевых точках владельцам широко распространенной инфраструктуры и создать диалог со своими потребителями. Используя опыт, полученный в ходе множества предыдущих запусков, LCE могут уделить больше внимания индивидуальным вопросам и предложениям.

Запускаем неожиданное

Когда проект выходит на новый уровень, службе LCE может понадобиться создать новый чек-лист с нуля. Этот процесс зачастую включает в себя синтезирование опыта, полученного специалистами в требуемой области. При создании черновика чек-листа может быть полезно структурировать его так, чтобы выделить в отдельные разделы наиболее распространенные темы, такие как надежность, типы сбоев и процессы.

Например, до запуска Android компания Google редко сталкивалась с устройствами широкого потребления, имеющими логику клиентской стороны, которую мы не можем контролировать. Мы способны в течение нескольких часов довольно легко исправить ошибку в сервисе Gmail, отправляя новую версию кода JavaScript в браузеры, но такой способ не подходит для мобильных устройств. Поэтому LCE, работающие

над запусками, связанными с мобильными устройствами, подключают к работе экспертов в этой области, чтобы определить, какие разделы чек-листа можно и нельзя создавать и когда нужно добавлять в него новые вопросы. В таких беседах важно учитывать цель каждого вопроса, чтобы избежать бездумного применения действия, которое не важно для разработки уникального продукта. LCE, столкнувшись с необычным запуском, должны вернуться к первым абстрактным принципам выполнения безопасного запуска, а затем индивидуализировать их, чтобы сделать чек-лист конкретным и полезным для разработчиков.

Разрабатываем чек-лист для запуска

Чек-лист — это инструмент для запуска новых сервисов и продуктов с ожидаемо высокой надежностью. Наш чек-лист для запуска рос с течением времени и периодически корректировался членами команды координаторов запуска. Детали чек-листа для запуска у каждой компании свои, поскольку специфика запуска должна настраиваться для внутренних сервисов и инфраструктуры этих компаний. В этом разделе мы рассмотрим несколько тем из чек-листов LCE и представим примеры того, как эти чек-листы можно реализовать.

Архитектура и зависимости

Обзор архитектуры позволяет определить, корректно ли сервис применяет разделяемую инфраструктуру и идентифицирует ее владельцев как дополнительных заинтересованных лиц. Компания Google имеет множество внутренних сервисов, которые служат кирпичиками для новых продуктов. На более поздних стадиях планирования производительности (см.

[Hixson, 2015a]) список зависимостей, описанный в этом разделе чек-листа, может быть использован для того, чтобы убедиться, что мы правильно подготовили каждую зависимость.

Примеры вопросов чек-листа

- Как выглядит поток запросов от пользователя к фронтенду и бэкенду?
- Существуют ли запросы, для которых имеются разные требования к задержке?

Примеры действий

- Изолируйте запросы, с которыми работают пользователи, от запросов, с которыми не работают.
- Проверяйте предположения об объеме запросов. Один просмотр страницы может превратиться в большое количество запросов.

Интеграция

Сервисы многих компаний запускаются во внутренней экосистеме, которая обуславливает способы настройки этих машин, конфигурирования новых сервисов, настройки системы мониторинга, интегрирования с системой балансировки нагрузки, настройки DNS-адресов и т.д. Эти внутренние экосистемы обычно растут с течением времени и зачастую имеют индивидуальные особенности и ловушки, с которыми нужно справиться. Поэтому этот раздел чек-листа в разных компаниях будет значительно различаться.

Примеры действий

- Настройте новое имя DNS для вашего сервиса.
- Настройте балансировщики нагрузки, которые будут общаться с вашим сервисом.
- Настройте систему наблюдения для вашего сервиса.

Планирование производительности

После запуска новой функциональности может произойти временное увеличение частоты использования сервиса, но в течение нескольких дней она снизится. Вид нагрузки или структуры трафика во время пика, возникающего при запуске, может значительно отличаться от этих же показателей во время устойчивой работы, что обесценит результаты нагрузочных тестов. Интерес общественности, как правило, трудно предугадать, и некоторые продукты компании Google должны были справиться с возникающими при запуске пиками, которые в 15 раз превышали оценочные значения. Первый удачный запуск в регионе или стране помогает придать уверенности в том, что вы сможете справиться с более крупными запусками.

Производительность, избыточность и доступность влияют друг на друга. Например, если вам нужно выполнить три реплицированных развертывания для обслуживания 100 % трафика во время пиковых нагрузок, то нужно поддерживать четыре или пять развертываний, одно или два из которых будут избыточными, для того чтобы защитить пользователей от отключений или неожиданных сбоев. Дата-центры и сетевые ресурсы зачастую создаются долго, и нужно делать запрос заранее, чтобы ваша компания могла их получить.

Примеры вопросов чек-листа

- Связан ли запуск с пресс-релизом, рекламой, статьей в блоге или другой формой рекламной кампании?
- Какого трафика и уровня роста вы ожидаете во время и после запуска?
- Получили ли вы все необходимые вычислительные ресурсы для поддержки своего трафика?

Типы сбоев

Систематическое рассмотрение возможных типов сбоев для нового сервиса гарантирует его высокую надежность с самого начала. В этой части чек-листа проверяйте все компоненты и зависимости и определяйте влияние их сбоев. Может ли сервис справиться со сбоями отдельных машин, с отключением данных центра, с сетевыми сбоями? Как вы справляетесь с неверными входными данными? Готовы ли вы к потенциальной DoS-атаке? Может ли сервис продолжать работу в деградированном режиме, если один из его зависимостей даст сбой? Как вы справляетесь с недоступностью зависимости при запуске сервиса? Во время его работы?

Примеры вопросов чек-листа

- Имеются ли в проекте единые точки сбоев?
- Как вы справляетесь с недоступностью зависимостей?

Примеры действий

- Реализуйте дедлайны запросов для того, чтобы избежать ситуации, когда у сервиса заканчиваются ресурсы во время работы с долго выполняющимися запросами.

- Реализуйте сегментирование нагрузки, чтобы отклонять новые запросы при перегрузках.

Поведение клиента

Для традиционных сайтов зачастую не нужно принимать во внимание злоупотребления зарегистрированных пользователей. Когда каждый запрос вызывается действием пользователя, например нажатием на ссылку, количество запросов ограничивается лишь тем, как быстро он можно выполнять эти нажатия. Для того чтобы нагрузка удвоилась, количество пользователей тоже должно удвоиться.

Эта аксиома перестает работать, когда мы рассматриваем системы, которые выполняют действия без участия пользователя, например приложение для сотового телефона, периодически синхронизирующее свои данные с облачным хранилищем, или сайт, который временами обновляется. В таких ситуациях злоупотребление клиентов может угрожать стабильности сервиса. (Существует также проблема защиты от злоупотреблений трафиком наподобие скраперов или DoS-атак, принципы которой отличаются от принципов проектирования безопасного поведения для основных клиентов.)

Пример вопроса

Предусмотрена ли у вас функциональность автосохранения/автозаполнения/контрольных сигналов?

Примеры действий

- Убедитесь, что ваш клиент имеет экспоненциальную задержку при сбое.
- Убедитесь, что вы распределяете во времени автоматические запросы.

Процессы и автоматизация

Компания Google поощряет использование инженерами стандартных инструментов для автоматизации распространенных процессов. Однако автоматизация никогда не работает идеально, и для каждого сервиса предусмотрены процессы, которые должен выполнять человек: создание нового релиза, перенос сервиса в другой дата-центр, восстановление данных из резервных копий и т.д. Для повышения надежности мы стремимся минимизировать количество единых точек сбоя, включающих в себя и людей.

Эти процессы должны быть задокументированы перед запуском, чтобы гарантировать, что информация будет доступна в критических ситуациях. Процессы должны быть задокументированы так, чтобы в критической ситуации любой член команды мог их выполнить.

Пример вопроса чек-листа

Существуют ли ручные процессы, которые необходимы для того, чтобы сервис работал?

Примеры действий

- Задокументируйте все ручные процессы.
- Задокументируйте все процессы, необходимые для перемещения сервиса в новый дата-центр.
- Автоматизируйте процессы сборки и выпуска новой версии.

Процесс разработки

Компания Google широко применяет системы контроля версий, с ними глубоко интегрированы почти все процессы разработки. Многие наши приемы связаны с эффективным использованием системы контроля версий. Например, мы

выполняем бо́льшую часть разработки в основной ветви, но сборка релизов происходит в отдельных ветвях. Такая настройка позволяет легко исправлять ошибки в релизах, не требуя получения несвязанных изменений из основной ветви.

Мы используем систему контроля версий и для других целей, например для хранения файлов конфигурации. Многие преимущества системы контроля версий — отслеживание истории, поручение внесения изменений отдельным людям и выполнение обзоров кода — применимы и к конфигурационным файлам. В некоторых случаях мы также передаем изменения из системы контроля версий на работающие серверы автоматически, и инженеру нужно только отправить изменение, чтобы оно попало в работающие системы.

Примеры действий

- Отправьте все файлы, содержащие код и конфигурацию, в систему контроля версий.
- Выполняйте каждый релиз в новой ветви.

Внешние зависимости

Иногда запуск зависит от внешних факторов, которые компания не может контролировать. Их определение позволяет вам справиться со связанной с ними непредсказуемостью. Например, зависимость может быть библиотекой кода, которую поддерживают сторонние разработчики, или сервисом, или данными, предоставляемыми другими компаниями. При сбое у поставщика, баге, систематической ошибке, проблеме с безопасностью или неожиданном ограничении масштабируемости своевременное планирование поможет вам избежать урона, который будет

причинен вашим пользователям, или уменьшить его. В ходе запусков мы использовали фильтрацию и/или переписывающие прокси, конвейеры перекодирования данных и кэши для того, чтобы снизить влияние этих рисков.

Примеры вопросов чек-листа

- От каких кода, данных, сервисов или событий, предоставляемых сторонними разработчиками, зависит сервис или запуск?
- Зависят ли от вашего сервиса какие-либо партнеры? Если да, то нужно ли их оповещать о вашем запуске?
- Что случится, если вы или поставщик не сможете соблюсти жесткий дедлайн запуска?

Планирование отправок

В крупных распределенных системах несколько событий могут произойти одновременно. По соображениям надежности это не всегда хорошо. Усложненный запуск может потребовать включения отдельной функциональности для нескольких различных подсистем, и на каждое изменение конфигурации может уйти несколько часов. Наличие работающей конфигурации в тестовом экземпляре не гарантирует, что та же конфигурация может быть выпущена для производственного экземпляра. Иногда сложные манипуляции или специальная функциональность могут потребоваться для того, чтобы запуск всех компонентов производился чисто и в правильном порядке.

Внешние требования, предъявляемые командами маркетинга и PR, могут еще больше усложнить задачу. Например, команде нужно, чтобы какая-то функциональность

была доступна к моменту основного доклада на конференции, но оставалась скрытой до этого времени. Действия на случай непредвиденных обстоятельств — еще одна часть планирования отправок. Что если вы не сможете вовремя выпустить функциональность для доклада? Иногда эти действия сводятся к простой подготовке запасного набора слайдов, которые гласят: «Мы запустим эту функциональность в течение нескольких дней» — вместо: «Мы запустили эту функциональность».

Примеры действий

- Создайте план запуска, который определяет шаги, которые нужно предпринять для запуска сервиса. Укажите ответственного за каждый шаг.
- Определите риск для отдельных этапов запуска и реализуйте действия на случай непредвиденных обстоятельств.

Избранные приемы выполнения надежных запусков

Как описано в других частях этой книги, компания Google разработала множество приемов для запуска надежных систем. Некоторые из них особенно хорошо подходят для безопасного запуска продуктов. Они дают преимущества и при повседневной работе сервиса, но особенно важно их правильно выполнить во время этапа запуска.

Постепенные отправки и отправки в несколько этапов

Поговорка из области системного администрирования гласит: «Никогда не изменяйте работающую систему». Любое изменение несет за собой риск, а он должен быть минимизирован для того, чтобы гарантировать надежность

системы. То, что работает для маленьких систем, вдвойне верно для высокореплицированных глобально распределенных систем наподобие тех, что запускает компания Google.

Лишь немногие запуски в нашей компании выполняются, так сказать, нажатием одной кнопки, когда мы запускаем новый продукт в заданное время для того, чтобы его мог использовать сразу весь мир. С течением времени компания Google разработала несколько шаблонов, которые позволили нам запускать продукты и функциональность плавно и тем самым минимизировать риск (см. приложение Б).

Почти все обновления сервисов Google происходят постепенно, в соответствии с определенным процессом и с выполнением проверочных шагов. Новый сервер может быть установлен на нескольких машинах одного дата-центра, и в течение заданного времени за ним будут наблюдать. Если все хорошо, сервер будет установлен на все машины одного дата-центра и мы снова станем наблюдать за ним, и только потом он будет глобально установлен на все машины. Первые этапы отправки обычно называют канареевыми тестами (см. главу 17). Наши серверы, которые в данном случае играют роль канареек, могут обнаруживать опасные последствия поведения нового ПО в ходе его работы с реальным пользовательским трафиком.

Канареевые тесты — это концепция, которая встроена в большое количество внутренних инструментов компании Google, используемых для выполнения автоматических изменений, а также в системах, изменяющих конфигурационные файлы. Инструменты, которые управляют установкой нового ПО, обычно какое-то время наблюдают за поведением только что запущенного сервера, убеждаясь, что он не «падает» и ведет себя как следует. Если изменение не проходит проверку, оно автоматически откатывается.

Концепция постепенных отправок применима и к тому ПО, которое не работает на серверах компании Google. Новые версии приложения для Android также могут быть внедрены постепенно — некоторым клиентам будет предложено обновить свои устройства до обновленной версии. Процент обновленных экземпляров приложения постепенно будет увеличиваться, пока не достигнет 100 %. Такой тип отправок особенно полезен, если внедрение новой версии приводит к тому, что на серверы бэкенда в data-центры компании Google поступает дополнительный трафик. Таким образом, мы можем наблюдать, как эта процедура воздействует на наши серверы, по мере того как мы постепенно отправляем новую версию и обнаруживаем проблемы.

Система приглашений — это еще одна разновидность постепенной отправки. Зачастую вместо того, чтобы позволить всем пользователям свободно регистрироваться, мы позволяем делать это в течение дня ограниченному количеству пользователей. Такой подход нередко связан с системой приглашений, при которой пользователь может отправить ограниченное количество приглашений своим друзьям.

Фреймворки флагов функциональности

Компания Google зачастую повышает качество тестирования перед запуском с помощью стратегий, которые снижают риски отключения. Механизм, позволяющий медленно выполнить отправку изменений и наблюдать за общим поведением системы под реальными нагрузками, может отплатить за эти инвестиции повышением надежности, скорости разработки и сокращением срока внедрения. Эти механизмы показали свою полезность в случаях, когда реалистичные тестовые среды непрактичны, а также при особенно сложных запусках, результат которых сложно предугадать.

Более того, не все изменения одинаковы. Иногда вы всего лишь хотите проверить, улучшает ли небольшое изменение пользовательского интерфейса навыки пользователей. Такие небольшие изменения не требуют написания тысяч строк кода или тяжеловесного процесса запуска. Вы можете захотеть протестировать тысячи таких изменений одновременно.

Наконец, иногда бывает нужно проверить на небольшой группе пользователей ранний прототип новой функциональности, которую трудно реализовать. Вы не хотите тратить месяцы работы инженеров на создание этой функциональности для миллионов пользователей только для того, чтобы понять, что она им не нужна.

Для того чтобы реализовать предыдущие сценарии, для некоторых наших продуктов были разработаны фреймворки флагов функциональности. Цель разработки некоторых из этих фреймворков — постепенно отправлять новую функциональность пользователям. Когда продукт предлагается любому подобному фреймворку, этот фреймворк максимально защищен, поэтому большей части приложений непосредственное участие LCE не требуется. Такие фреймворки обычно соответствуют следующим требованиям.

- Отправляют множество изменений одновременно на несколько серверов, пользователей, объектов или данных центров.
- Постепенно увеличивают группу пользователей, обычно на 1–10 %.
- Направляют трафик через разные серверы в зависимости от пользователей, сессий, объектов и/или локаций.

- Автоматически обрабатывают сбои новых путей кода согласно проекту без участия пользователей.
- Выполняют независимый мгновенный откат каждого из этих изменений в случае обнаружения серьезной ошибки или побочного эффекта.
- Измеряют степень влияния изменений на навыки пользователя.

Фреймворки флагов функциональности компании Google делятся на две общие категории.

- Фреймворки, которые в основном поддерживают улучшения пользовательского интерфейса.
- Фреймворки, которые поддерживают произвольные изменения серверной и бизнес-логики.

Самый простой фреймворк флагов функциональности для изменений пользовательского интерфейса в сервисе, не сохраняющем состояние, — это инструмент, переписывающий полезную нагрузку HTTP для фронтенд-серверов приложения, ограниченный подмножеством cookies или другого аналогичного атрибута запросов/ответов HTTP. Механизм конфигурации может указывать идентификатор, связанный с новыми путями кода, масштабом изменений (например, диапазон режима хеширования для cookies), белыми и черными списками.

Сервисы, запоминающие состояние, стремятся ограничить флаги функциональности подмножеством уникальных идентификаторов авторизовавшихся пользователей или

сущностями продуктов, к которым выполняется доступ, например идентификаторами документов, электронных таблиц или объектов хранилища. Вместо того чтобы переписывать полезную нагрузку HTTP, сервисы, сохраняющие состояние, скорее всего, будут проксировать или перенаправлять запросы на другие серверы в зависимости от изменений, связанных со способностью протестировать улучшенную бизнес-логику и более сложную функциональность.

Справляемся со злоупотреблениями клиента

Самым простым примером злоупотреблений клиента является неверная оценка уровня обновлений. Новый клиент, который синхронизируется каждые 60 секунд, а не каждые 600 секунд, как прежде, вызывает десятикратное увеличение нагрузки на сервис. Поведение сервиса при повторных попытках имеет несколько ловушек, которые влияют на запросы, инициированные как пользователем, так и клиентом. Рассмотрим пример сервиса, который перегружен, из-за чего некоторые запросы к нему дают сбой. Если клиенты выполняют повторные попытки запросов, давших сбой, они повышают нагрузку на уже перегруженный сервис, что еще больше увеличивает количество повторных попыток и повторных запросов. Вместо этого клиенты должны снизить частоту выполнения запросов, обычно добавляя экспоненциально увеличивающуюся задержку между повторными попытками в дополнение к тщательному рассмотрению типов ошибок, которые требуют выполнения повторных попыток. Например, ошибка сети обычно требует сделать повторную попытку, тогда как ошибка HTTP 4xx (которая показывает ошибку на стороне клиента) обычно этого не делает.

Намеренная или непреднамеренная синхронизация автоматических запросов в «шумной толпе» (например, описанная в главах 24 и 25) — это еще один распространенный пример злоупотреблений клиента. Разработчик приложения для телефона может решить, что два часа ночи — это хорошее время для загрузки обновлений, поскольку пользователи в это время, скорее всего, будут спать и она им не помешает. Однако этот план приводит к появлению шквала запросов к серверу загрузки в два часа ночи каждый день, и почти никаких запросов не будет в остальное время. Вместо этого клиент должен выбирать время для подобных запросов случайно.

Случайность нужно внедрить и в другие периодические процессы. Говоря об упомянутых ранее повторных попытках: рассмотрим клиент, который отправляет запрос и, когда сталкивается со сбоем, выполняет повторные попытки через 1, 2, 4 секунды и т.д. Без случайности короткий пик запросов, приводящий к повышению количества ошибок, может повторяться через 1, 2, 4 секунды и т.д. Для того чтобы стабилизировать эти синхронизированные события, каждую задержку нужно «встряхнуть», то есть сдвинуть на случайный промежуток времени.

Способность контролировать поведение клиента на стороне сервера зарекомендовала себя как важный инструмент. Для приложения на устройстве такой контроль может заключаться в том, чтобы предлагать клиенту периодически проверять обновления на сервере и загружать файл конфигурации. Файл может включать и отключать определенную функциональность или набор параметров, например, указывать, как часто клиент будет синхронизироваться и выполнять повторные попытки.

Конфигурация клиента может даже включить совершенно новую функциональность, заметную пользователям. Разместив код, который поддерживает новую функциональность, в

клиентском приложении до того, как активизировать ее, мы значительно снижаем риск, связанный с этим запуском. Выпуск новой версии становится гораздо проще, если не нужно поддерживать параллельные релизы для версий с новой функциональностью и без нее. Это особенно верно, если мы работаем не с одним фрагментом новой функциональности, а с набором отдельных особенностей, которые могут быть выпущены в разное время.

Кроме того, наличие скрытой функциональности упрощает отмену запусков, когда во время отправки обнаруживаются негативные эффекты. В таких случаях мы можем просто отключить функциональность, переделать работу и выпустить обновленную версию приложения. Без такой конфигурации на стороне клиента нам пришлось бы предоставить новую версию приложения, не содержащую этой функциональности, и обновить приложение на телефонах всех пользователей.

Поведение при перегрузке и нагрузочные тесты

Перегрузки — это особенно сложный вид сбоя, поэтому он заслуживает особого внимания. Ошеломительный успех — это зачастую наиболее приятная причина перегрузок при запуске нового сервиса, но существует и множество других причин, в том числе сбои при балансировке нагрузки, отключение машин, синхронизация клиентского поведения и внешние атаки.

Упрощенная модель подразумевает, что степень использования ЦП машины, предоставляющей возможность применения определенного сервиса, увеличивается линейно с ростом нагрузки, например количества запросов или объема обработанных данных, и как только доступные ресурсы процессора истощаются, обработка попросту замедляется. К сожалению, в реальном мире сервисы редко ведут себя

настолько идеально. Многие из них работают гораздо медленнее, когда не находятся под нагрузкой, обычно это происходит из-за разнообразных кэшей: кэшей процессора, JIT и кэшей, характерных для данного сервиса. По мере увеличения нагрузки, как правило, образуется окно, в пределах которого степень использования процессора и нагрузка на сервис увеличиваются линейно и время ответов остается постоянным.

В какой-то момент по мере приближения к перегрузке многие сервисы могут достигнуть точки нелинейности. В самых безобидных случаях время ответа просто начинает увеличиваться, что приводит к неудобству для пользователей, но неизбежно вызывает сбой (однако медленная зависимость может вызвать видимые пользователям ошибки из-за превышенных дедлайнов RPC). В самых крайних случаях сервис полностью зависает из-за перегрузки.

Рассмотрим конкретный пример поведения при перегрузке: сервис записывал в журнал информацию для отладки в ответ на ошибки бэкенда. Оказалось, что делать это было гораздо дороже, чем обрабатывать ответ бэкенда при нормальном состоянии. Поэтому, как только сервис обнаруживал перегрузку и прерывал по тайм-ауту ответы бэкенда внутри собственного стека RPC, он тратил еще больше времени процессора на журналирование этих ответов, прерывая по тайм-ауту еще большее количество запросов, и это продолжалось до тех пор, пока сервис не отключался полностью. В сервисах, запущенных на виртуальной машине Java (Java Virtual Machine, JVM), похожий эффект иногда называется GC-перегрузкой (засорением сборщика мусора). В этом сценарии управление внутренней памятью виртуальной машины происходит в постоянно уменьшающихся циклах. Память освобождается до

тех пор, пока все время процессора не затрачивается только на управление памятью.

К сожалению, исходя из базовых принципов, трудно предугадать, как сервис будет вести себя в случае перегрузки. Поэтому нагрузочные тесты являются бесценным инструментом как с точки зрения надежности, так и с точки зрения планирования производительности, и нагрузочное тестирование необходимо выполнять для большинства запусков.

Развитие службы LCE

В годы становления компании Google команда инженеров увеличивалась каждый год несколько лет подряд, а службу разработки разбивали на несколько мелких команд, работающих над большим количеством новых продуктов и функций. В такой атмосфере инженеры-новички рисковали повторить ошибки своих предшественников, особенно когда речь шла об успешном запуске новой функциональности и новых продуктов.

Для того чтобы извлечь уроки из предыдущих запусков и снизить вероятность повторения ошибок, небольшая группа опытных инженеров, которых называли инженерами запуска, добровольно выступила в роли команды-консультанта. Для запусков новых продуктов инженеры запуска разработали чек-листы на следующие темы.

- Когда следует проконсультироваться с юристами.
- Как выбирать доменные имена.
- Как зарегистрировать новые домены, правильно сконфигурировав DNS.

- Какие ловушки часто встречаются при проектировании и развертывании на производстве.

Обзоры запусков, как стали называться сессии консультаций инженеров запуска за несколько дней или недель перед запуском, для многих новых продуктов стали распространенной практикой.

В течение двух лет чек-лист требований к развертыванию продуктов при запуске стал большим и сложным. Из-за этого, а также из-за увеличивающейся сложности среды разработки компании Google разработчикам становилось все труднее оставаться в курсе того, как безопасно вносить изменения. В то же время служба SRE росла быстрыми темпами, а неопытные SR-инженеры иногда были слишком осторожными и не хотели меняться. В компании Google опасались, что переговоры между этими двумя сторонами снизят скорость разработки новых продуктов и функциональности.

Для того чтобы сгладить этот сценарий с точки зрения инженеров, в рамках службы SRE в 2004 году была создана небольшая команда LCE, работавших на полную ставку. Они отвечали за ускорение запуска новых продуктов и функциональности, применяя опыт SRE для того, чтобы гарантировать, что компания Google поставляет надежные продукты, имеющие высокую доступность и низкую задержку.

LCE должны были гарантировать, что запуски проходят быстро, а сервисы не дают сбоев, а также отвечать за то, что при сбое одного запуска не дадут сбой другие продукты. LCE также должны были информировать заинтересованных лиц о природе и вероятности возникновения таких сбоев при сглаживании углов, для того чтобы ускорить сроки внедрения. Их сессии консультаций были названы производственными обзорами.

Эволюция чек-листа LCE

По мере того как окружение компании Google становилось более сложным, усложнялся чек-лист инженеров — координаторов запуска (см. приложение Д) и рос объем запусков. В течение трех с половиной лет один LCE проводил 350 запусков с помощью чек-листа. Поскольку в команду входили в среднем пять человек, то мы выполнили более 1500 запусков за 3,5 года!

Несмотря на то что каждый вопрос чек-листа LCE прост, сложность в основном обусловлена тем, что вызвало этот вопрос, а также выводами из ответа. Для того чтобы полностью понять эту сложность, новому LCE требуется примерно шесть месяцев тренировок.

По мере увеличения количества запусков, а также ежегодного увеличения штата инженеров, LCE искали способ упростить свои обзоры. Они определили категории запусков с низкими рисками, которые, скорее всего, не вызовут сбоев. Например, запуск функциональности, не включающий в себя новых исполняемых файлов для сервера и повышающий трафик менее чем на 10 %, почти не несет риска. Для таких запусков применялся более простой чек-лист, а запуски, связанные с высокими рисками, проходили через весь диапазон проверок и балансировок. К 2008 году 30 % обзоров считались не несущими рисков.

Одновременно окружение компании Google масштабировалось, избавляясь от ограничений для многих запусков. Например, приобретение YouTube заставило компанию Google построить его сеть и более эффективно использовать полосу пропускания. Это означало, что множество более мелких продуктов могли «влезть в образовавшиеся трещины», избегая сложных процессов планирования производительности сети и снабжения, что

ускоряло их запуски. Компания Google также начала строить очень крупные дата-центры, в которых несколько зависимых сервисов могли находиться под одной крышей. Такое развитие упростило запуск новых продуктов, нуждавшихся в значительной производительности для нескольких уже существующих сервисов, от которых они зависели.

Проблемы, которые инженеры LCE не решили

Несмотря на то что LCE старались свести к минимуму бюрократию для своих обзоров, этих усилий было недостаточно. К 2009 году сложность запуска маленького нового сервиса в компании Google стала легендой. Сервисы, чей масштаб увеличивался, сталкивались с собственным набором проблем, которые инженеры — координаторы запуска решить не могли.

Изменения масштабируемости

Когда успешность продуктов начинает значительно превышать все первоначальные оценки и степень их использования увеличивается больше чем на два порядка, для того чтобы поспеть за их нагрузкой, требуется значительно изменить проект. Такие изменения масштабируемости в сочетании с постоянным добавлением функциональности зачастую делают продукт более нестабильным и сложным для работы. В какой-то момент архитектура оригинального продукта становится неуправляемой, и ее нужно создавать заново. Разработка новой архитектуры продукта, а затем перевод всех пользователей со старой архитектуры на новую требует от разработчиков и SR-инженеров значительных инвестиций времени и ресурсов, снижая скорость создания новой функциональности в этот период.

Рост операционной нагрузки

Когда сервис начинает работать после запуска, операционная нагрузка — количество ручной и повторяющейся работы, необходимой для функционирования системы, как правило, с течением времени увеличивается, если не предпринять целенаправленных усилий по управлению ею. Шумность автоматизированных оповещений, сложность процедур развертывания и нагрузка, вызываемая ручной работой по обслуживанию, с течением времени обычно растут, увеличивая полосу пропускания для владельцев сервиса и оставляя командам меньше времени на разработку функциональности. SR-инженеры должны поддерживать уровень операционной работы ниже 50 % времени (см. главу 5). Поддержание заданного уровня операционной работы, не превышающего этот максимум, требует постоянного отслеживания источников операционной работы, а также приложения усилий по их ликвидации.

Текущка инфраструктуры

Если инфраструктура, лежащая в основе сервиса, например системы управления кластерами, хранения данных, наблюдения, балансировки нагрузки и обмена данными, изменяется из-за активной разработки, которую ведет команда разработчиков инфраструктуры, то владельцы сервисов, работающих с этой инфраструктурой, должны выполнить большую работу только для того, чтобы поспевать за изменениями инфраструктуры. Как только функциональность инфраструктуры, от которой зависит сервис, устаревает и заменяется новой, владельцы сервисов должны модифицировать свою конфигурацию и перестроить свои исполняемые файлы, для чего должны «бежать со всех ног, чтобы только оставаться на месте». Решение для этого

сценария состоит в применении политики снижения определенной текучки, которая мешает инженерам, работающим с инфраструктурой, выпускать изменения, не имеющие обратной совместимости, до тех пор, пока они не автоматизируют переход клиентов на использование этой новой функциональности. Создание автоматизированных инструментов перехода для поддержки новой функциональности снижает объем работы, которую должны выполнить владельцы сервиса для того, чтобы справиться с текучкой инфраструктуры.

Решение этих проблем требует значительных усилий в масштабах компании, находящихся вне компетенции LCE: комбинации более качественных API платформы и фреймворков (см. главу 32), продолжительной сборки и автоматизации тестов, а также бо'льших стандартизации и автоматизации производственных сервисов компании Google.

Итоги главы

Компаниям, переживающим быстрый рост и значительные изменения их продуктов и сервисов, может пойти на пользу наличие инженеров — координаторов запуска. Команды из таких инженеров особенно полезны, если компания планирует удваивать количество разработчиков продуктов каждые год или два, если она должна масштабировать свои сервисы так, чтобы работать с сотнями миллионов пользователей, а также если для пользователей важна надежность, несмотря на быстрые темпы внедрения изменений.

Команда LCE в компании Google стала решением проблемы достижения безопасности без остановки процесса внедрения изменений. В этой главе были показаны решения, к которым приходили члены команды LCE на протяжении более чем 10

лет именно при таких обстоятельствах. Мы надеемся, что наш подход поможет вдохновить других людей, которые сталкиваются с аналогичными задачами в своих организациях.

Часть IV. Управление

В последней части этой книги рассматриваются темы работы в команде и работы в качестве единой команды. Ни одно подразделение SRE не похоже на изолированную группу, и у нас сформировались оригинальные способы организации нашей работы.

Когда речь идет о создании службы SRE, любая солидная компания должна задуматься об обучении SR-инженеров тому, как реагировать в сложных и быстро изменяющихся ситуациях. Тщательно продуманная и успешно проведенная программа обучения может дать новому сотруднику хороший практический опыт в течение первых нескольких недель или месяцев, что в противном случае потребовало бы нескольких месяцев или лет. О стратегиях обучения мы поговорим в главе 28 «Ускоренное обучение SR-инженеров для работы на дежурствах и не только».

Любой, кто работал в службе эксплуатации, знает, что ответственность за все важные сервисы сопровождается большим количеством отвлекающих факторов: работающая система приходит в нестабильное состояние, люди требуют обновлений своих любимых библиотек, очередь запросов на консультации увеличивается... Управление отвлекающими факторами в стрессовых условиях — это обязательный навык, мы рассмотрим его в главе 29 «Справляемся с отвлекающими факторами и прерываниями».

Если такой бешеный режим сохраняется достаточно долго, команда SR-инженеров должна начать восстанавливаться от операционной перегрузки. Мы продемонстрируем вам свой план восстановления в главе 30 «Добавляем в команду нового

SR-инженера, чтобы предотвратить операционную перегрузку».

В главе 31 «Общение и взаимодействие в службе SRE» мы расскажем о разных ролях в SRE; о межкомандном, межплощадочном и межконтинентальном взаимодействии; о проведении производственных совещаний; а также рассмотрим примеры того, как сотрудники SRE успешно взаимодействуют друг с другом.

Наконец, в главе 32 «Развитие модели вовлеченности SR-инженеров» мы рассмотрим фундаментальный принцип работы SR-инженеров, который состоит в проверке готовности продукта (production readiness review, PRR) и является критическим шагом при внедрении нового сервиса. Мы поговорим о том, как проводить PRR и как сделать следующий шаг от этой успешной, но ограниченной модели.

Рекомендуемая литература от Google SRE

Сборка надежных систем требует тщательно откалиброванного набора навыков, которые ранжируются от разработки ПО до, возможно, менее известных инженерных направлений и методик анализа систем. О последних можно прочитать в статье *The Systems Engineering Side of Site Reliability Engineering* [Hixson, 2015b].

Процесс найма SR-инженеров критически важен для того, чтобы у вас была высокофункциональная служба по обеспечению надежности информационных систем, что рассматривается в статье *Hiring Site Reliability Engineers* [Jones, 2015]. Приемы найма компании Google детально

рассматриваются в текстах вроде *Work Rules!* [Bock, 2015]¹, но наем SR-инженеров имеет свои особенности. Даже по общим стандартам компании Google кандидатов для SRE трудно найти и еще труднее проинтервьюировать.

[164](#)

[164](#) Написана Лазло Боком, Google's Senior VP of People Operations.

28. Ускоренное обучение SR-инженеров для работы на дежурствах и не только

Автор — Эндрю Уиддоусан.

Под редакцией Шилайи Нукалы

Как я могу прицепить реактивный ранец на своих новичков и при этом держать в курсе более опытных SR-инженеров?

Вы наняли новых SR-инженеров, что теперь?

Вы наняли новых служащих для своей организации, и они начинают свой путь как SR-инженеры. Теперь вам нужно научить их выполнять свою работу. Инвестирование в образование новых SR-инженеров поможет им стать отличными инженерами. Обучение улучшит их профессиональные навыки, сделает их более целостными и сбалансированными.

Успешные команды SR-инженеров строятся на доверии — для того чтобы поддерживать сервис в стабильном состоянии, вам нужно верить, что ваши коллеги по дежурству знают, как работает¹⁶⁵ ваша система, могут диагностировать ее атипичное поведение, не стесняются попросить помощи и способны реагировать в стрессовых ситуациях, чтобы спасти ситуацию. Очень важно, хоть этого и недостаточно, рассматривать обучение SR-инженеров с точки зрения «Что должен знать новичок для того, чтобы начать работать на

дежурстве». Учитывая требования к доверию, вам также нужно задать следующие вопросы.

- Как имеющиеся дежурные инженеры могут оценить готовность новичка к работе на дежурстве?
- Как мы можем воспользоваться энтузиазмом и любопытством наших новичков, чтобы принести пользу для SRE?
- Как я могу внести свой вклад в обучение новых сотрудников?

Различные люди предпочитают разные методики обучения. Естественно, в вашей команде будут абсолютно разные люди и пользоваться только одной методикой недальновидно. Таким образом, не существует методики, которая лучше всего подойдет для обучения новых SR-инженеров, и определенно ни одна магическая формула не сработает для всех подряд команд SR-инженеров. В табл. 28.1 перечислены рекомендованные методы обучения (и соответствующие антиметоды), которые хорошо известны SR-инженерам компании Google. Эти приемы представляют собой широкий диапазон возможностей, доступных для того, чтобы качественно обучить вашу команду основным концепциям SRE как в текущий момент, так и на постоянной основе.

Таблица 28.1. Приемы обучения SR-инженеров

Рекомендованные методы	Нежелательные методы
Разработка конкретных последовательных упражнений для учеников	Поручение ученикам низкоквалифицированной работы (например, с оповещениями или тикетами) для их обучения; боевые крещения

Стимулирование к обратному проектированию (реверс-инжинирингу), статистическому анализу и изучению основных принципов	Обучение с помощью операционных процедур, чек-листов и сценариев
Мотивация к анализу сбоев путем чтения студентами постмортемов	Отношение к сбоям как к секретной информации, которая должна быть погребена, чтобы можно было избежать ответственности
Создание локализованных, но реалистичных сбоев для учеников, чтобы они осваивали системы мониторинга и инструменты	Предоставление возможности что-то исправить только после того, как ученик начал работать на дежурстве
Имитация в группе катастроф для объединения подходов команды к решению проблем	Создание в команде экспертов, чьи знания и приемы относятся к разным категориям
Периодический допуск учеников к дежурствам в качестве практикантов (теневое дежурство) и сравнение их заметок с заметками дежурного инженера	Назначение ученика основным дежурным инженером при отсутствии у него полного понимания работы сервиса
Объединение учеников с инженерами-экспертами для повторения определенных разделов плана обучения инженеров	Рассмотрение планов обучения работе на дежурстве как неизменных и неприкословенных для всех, кроме экспертов
Обосновление нестандартной работы над проектом для выполнения учениками с предоставлением им возможности частично вступить во владение стеком технологий	Предоставление всей работы над новым проектом самым опытным сотрудникам; поручение новичкам лишь мелкой работы

В остальной части этой главы приведены основные подходы, которые мы считаем эффективными при ускоренном обучении SR-инженеров для работы на дежурствах и не только. Эти методы можно представить в виде схемы самообучения SR-инженеров (рис. 28.1).

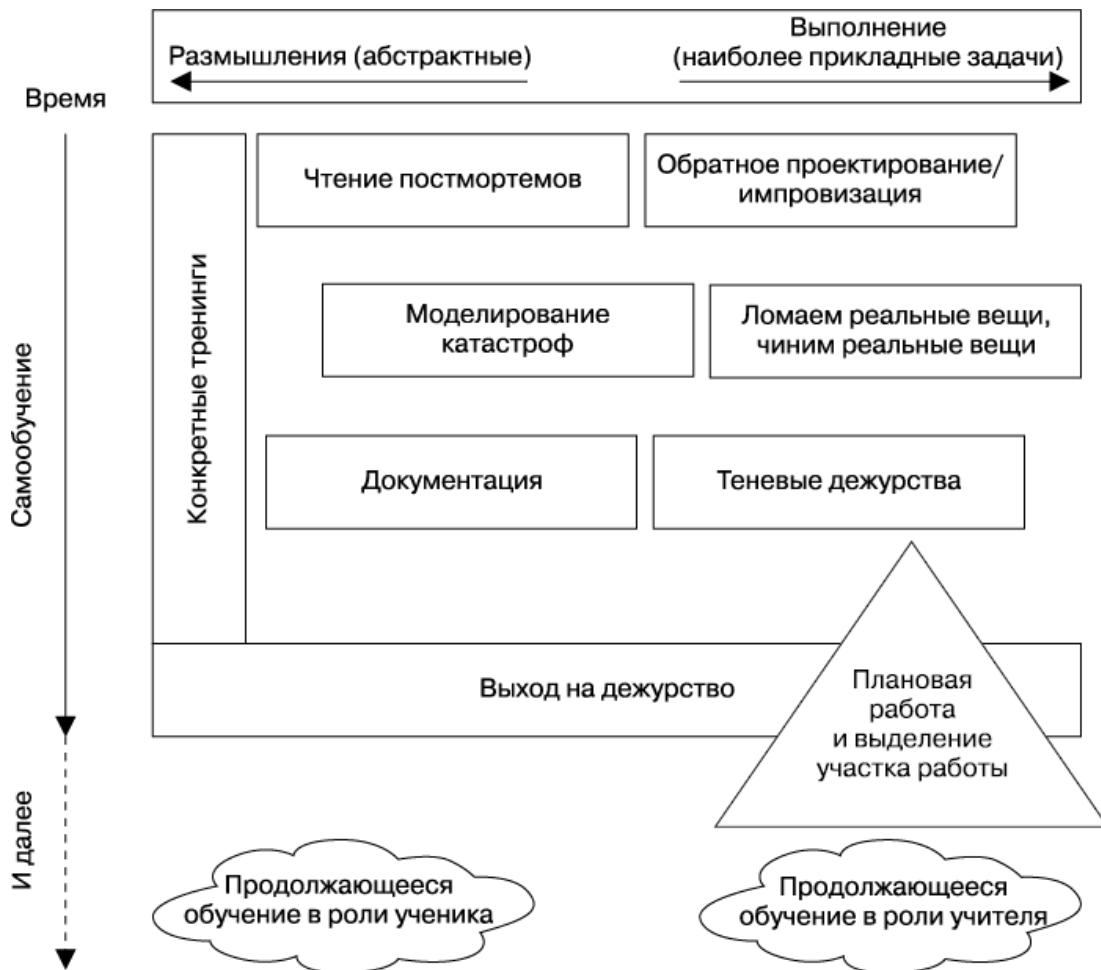


Рис. 28.1. План по выходу на самообучение SR-инженера для работы на дежурствах и не только

На этом рисунке показаны лучшие приемы, которые команды SR-инженеров могут выбрать для того, чтобы помочь выйти на самообучение своих новых сотрудников. Из множества представленных здесь инструментов вы можете выбрать те, которые наилучшим образом подходят вашей команде.

На этом рисунке показаны две оси.

- Горизонтальная ось представляет диапазон разных видов работы, который ранжируется от абстрактных до прикладных процессов.

- Вертикальная ось представляет *время*. Поначалу новые SR-инженеры почти не владеют информацией о системах и сервисах, за которые они будут нести ответственность, поэтому изучение отчетов о происшествиях (постмортемов), в которых детально описано, как эти системы давали сбой, станут отличным отправным пунктом. Новые SR-инженеры также могут выполнить обратное проектирование этих систем с самых основ, поскольку они сами начинают с нуля. Как только они станут лучше понимать эти системы и перейдут к выполнению конкретных операций, они будут готовы выходить на теневые дежурства и начать исправлять неполную или устаревшую документацию.

Ниже даны советы по интерпретации этого рисунка.

- Участие в дежурствах — это важный этап в карьере нового SR-инженера, после чего обучение становится менее прозрачным, неопределенным и самоуправляемым — именно поэтому вы видите пунктирные линии вокруг действий, которые становятся доступны после того, как SR-инженер начинает дежурить.
- Треугольная форма *работы над проектами и выделения участка работы* указывает на то, что работа над проектом начинается с малого и со временем увеличивается, становясь более сложной, и, скорее всего, продолжается и после дежурства.
- Одни из этих действий и практик очень абстрактные/пассивные, а другие — прикладные/активные. Некоторые действия являются комбинацией обоих типов. Желательно использовать разнообразные методы обучения.

- Для максимального эффекта практическая деятельность в процессе обучения должна подчиняться соответствующим правилам: одни операции можно выполнить сразу, другие должны происходить сразу после того, как SR-инженер официально начинает дежурить, а третья необходиомо выполнять на постоянной основе даже опытным SR-инженерам. *Конкретные тренинги* необходимо продолжать проводить на протяжении всего времени от вступления инженера в должность и вплоть до начала его работы на дежурствах.

Первый тренинг: структурировать хаос

Как говорится в этой книге, команды SR-инженеров совмещают проактивную¹⁶⁶ и реактивную¹⁶⁷ работу. Перед каждой командой SR-инженеров должна стоять цель ограничивать и снижать количество реактивной работы путем увеличения проактивности, и подход, с помощью которого выанимаете новичков, не должен стать исключением. Рассмотрим следующий очень распространенный, но, к сожалению, неоптимальный процесс найма.

Джон – новый член команды SR-инженеров, обслуживающих сервис FooServer. Более опытные SR-инженеры делают большое количество грязной работы – отвечают на тикеты, обрабатывают оповещения и выполняют утомительные отправки бинарных файлов. В первый день работы Джону поручают отслеживать все входящие тикеты. Ему говорят, что он может попросить любого члена команды SR-инженеров предоставить ему информацию, необходимую для расшифровки тикета. «Конечно, поначалу тебе придется нелегко, – говорит Джону менеджер, – но в конце концов ты начнешь

разбираться с этими тикетами гораздо быстрее. Когда-нибудь ты узнаешь все об используемых нами инструментах и поддерживаемых системах». Опытный инженер комментирует: «А сейчас мы бросим тебя в самое глубокое место озера».

Этот метод «боевого крещения» для ориентирования новичков зачастую инициирован самими членами команды. Управляемые операционной работой реактивные команды SR-инженеров «тренируют» своих новичков, заставляя их... реагировать! Раз за разом. Однако существует вероятность, что такая стратегия охладит пыл некоторых способных инженеров. Несмотря на то что такой подход может в итоге привести к созданию хороших операционистов, его результаты будут неудовлетворительны. Описанный подход также предполагает, что многие (или даже большинство) сложности, с которыми сталкивается команда, могут быть изучены в процессе работы, а не рассуждений. Если какой-то работе, с которой сталкивается человек, занимающийся тикетами, можно обучиться, выполняя простые упражнения, то она не относится к обязанностям SR-инженера.

У новичков в службе SRE возникают следующие вопросы.

- Над чем я работаю?
- Какого прогресса я добиваюсь?
- Когда эти действия позволят мне получить достаточно опыта, чтобы я мог начать дежурить?

Если человек только окончил университет или сменил место работы, изменив при этом служебную роль (с традиционного разработчика ПО или системного администратора) на

таинственную роль SR-инженера, то этого зачастую достаточно, чтобы пошатнуть его самоуверенность. Для более замкнутых личностей (особенно в отношении второго и третьего вопросов) неопределенность, вызываемая запутанными или непрозрачными ответами, может привести к замедлению обучения или проблемам с запоминанием. Вместо этого рассмотрим подходы, описанные в следующих разделах. Эти предложения настолько же конкретны, насколько может быть конкретным любой тикет или оповещение, но они также последовательны и поэтому более полезны.

Кумулятивные и методичные пути обучения

Постарайтесь как-то упорядочить систему обучения, чтобы ваши новые SR-инженеры понимали, куда они движутся. Любой вид обучения подойдет лучше применения случайных тикетов и прерываний, но вам придется немного потрудиться, чтобы правильно преподнести теорию и практику. Абстрактные концепции, которые появятся несколько раз на протяжении обучения новичка, должны быть описаны заранее. Кроме того, ученики должны получить опыт из первых рук настолько быстро, насколько это возможно на практике.

Для изучения вашей платформы и подсистем требуется начальная точка. Задумайтесь о том, что будет лучше — групповые тренировки с использованием сходных задач или обычный порядок выполнения. Например, если ваша команда отвечает за действующий в реальном времени обслуживающий стек, который работает с пользователями, рассмотрим следующий порядок обучения.

1. *Как запрос входит в систему.* Основные принципы работы с сетями и дата-центрами, балансировка нагрузки на фронтенде, прокси и т.д.

2. *Обслуживание фронтенда.* Фронтенды приложений, журналирование запросов, целевые показатели уровня обслуживания и т.д.
3. *Сервисы среднего уровня.* Кэши, балансировка нагрузки на бэкенде.
4. *Инфраструктура.* Бэкенды, инфраструктура и вычислительные ресурсы.
5. *Объединяем все вместе.* Приемы отладки, процедуры эскалации и сценарии для неотложных ситуаций.

Только от вас и от SR-инженеров, помогающих вам структурировать, разрабатывать и подавать курс обучения, зависит то, как вы будете предоставлять возможности для обучения (неформальные беседы, сухая теория или практические упражнения). Команда SR-инженеров, работающая с системой поиска, структурировала процесс обучения с помощью документа, который называется «чек-лист обучения работе на дежурстве». Упрощенный раздел этого чек-листа может выглядеть так.

Сервер агрегации результатов (Mixer)	
<p>Фронтенд: сервер фронтенда.</p> <p>Вызываемые бэкенды: сервер получения результатов, сервер геолокации, база данных персонализации.</p> <p>SRE-эксперты: Салли У., Дэйв К., Джен П.</p> <p>Контакты разработчиков: Джим Т., results-team@</p>	<p>Изучите перед тем, как двигаться дальше: на каких кластерах развернут сервер; как откатить релиз сервера; какие бэкенды считаются «лежащими на критическом пути» и почему</p>
<p>Прочтите и проанализируйте следующие документы: обзор агрегированных результатов: раздел «Выполнение запросов»;</p>	<p>Вопросы для самопроверки: Как меняется расписание релизов, если на привычный для релиза день выпадает праздник компании?</p>

обзор агрегированных результатов: раздел «Производство»;
план: как развернуть новый сервер агрегации результатов;
анализ производительности сервера

Как вы можете обнаружить плохую отправку набора данных, связанного с геолокацией?

Обратите внимание на то, что в этом разделе никак не закодированы процедуры, этапы диагностики или сценарии. Вместо этого он представляет собой относительно устойчивое описание, сконцентрированное непосредственно на перечислении контактов экспертов, подчеркивании наиболее полезной документации, указании того, какие базовые знания вы должны собрать и усвоить, а также задании наводящих вопросов, на которые можно ответить только после получения базовых знаний. В нем также показаны конкретные результаты, чтобы ученик знал, какого рода знания и навыки он получит после завершения этого раздела чек-листа.

Всем заинтересованным сторонам нужно понять, сколько информации запомнит обучаемый. Несмотря на то что такой механизм обратной связи не обязательно должен быть формальным (например, иметь форму теста), лучше всего добавить домашние задания, где будут задаваться вопросы о том, как работает ваш сервис. Удовлетворительные ответы, проверенные ментором, являются признаком того, что обучение можно переводить на следующий уровень. Вопросы, связанные с внутренней кухней вашего сервиса, могут звучать так.

- Какие бэкенды этого сервера считаются «лежащими на критическом пути» и почему?
- Какие части этого сервера должны быть упрощены или автоматизированы?

- Как вы думаете, где находится первое узкое место этой архитектуры? Если бы вам нужно было устраниить это узкое место, как бы вы это сделали?

В зависимости от того, как сконфигурированы права доступа для вашего сервиса, вы также можете реализовать многоуровневую модель доступа. Первый уровень доступа даст вашим ученикам лишь возможность рассмотреть внутреннее устройство ваших компонентов в режиме «только чтение», а более высокие уровни позволят им изменять состояние производственной системы. Удовлетворительное выполнение сценариев чек-листа, предназначенного для обучения работе на дежурстве, позволит вашему ученику получить более обстоятельный доступ к системе. Команда SR-инженеров, работающая над сервисом Google Search, называет эти уровни бонусами¹⁶⁸ на пути к дежурствам, поскольку ученики в конечном счете получают высшие уровни доступа к системе.

Целевая, а не черновая работа над проектом

SR-инженеры решают проблемы, дайте же им решить серьезную проблему! На первых порах даже незначительное чувство ответственности за сервис, которым управляет команда, может творить чудеса при обучении. Кроме того, чувство вовлеченности позволит наладить доверительные отношения с более опытными инженерами, поскольку они будут учить своих младших товарищей управлению новыми компонентами или процессами. В компании Google стараются сразу поручить каждому инженеру определенный участок работы: все SR-специалисты получают стартовый проект. В результате новые сотрудники максимально быстро знакомятся с инфраструктурой и как можно раньше начинают вносить

небольшой, но полезный вклад в общее дело. Поскольку SR-инженер параллельно обучается и работает над проектом, это позволяет ему быстрее понять функциональное назначение сервиса, чего не произошло бы, если бы он тратил время только на работу или только на обучение. Рассмотрим несколько неплохих схем работы.

- Создание небольшого изменения функциональности, заметного пользователям, в обслуживающем стеке, и последовательное сопровождение релиза этой функциональности вплоть до его промышленной эксплуатации. Понимание набора инструментов разработки и процесса релиза бинарного файла способствует развитию эмпатии среди разработчиков.
- Добавление функциональности для наблюдения за сервисом там, где в данный момент есть «белые пятна». Новичкам придется аргументировать логику наблюдения, согласовав их понимание системы с тем, как она себя (неправильно) ведет на самом деле.
- Автоматизация проблемных мест, которые были недостаточно проблемны для того, чтобы их автоматизировали ранее. Это позволит новому инженеру оценить работу службы SRE, связанную с избавлением от рутины, возникающей при выполнении повседневных операций.

Подготовка выдающихся реверс-инженеров, умеющих импровизировать

Мы можем дать рекомендации, как обучать новых SR-инженеров, но чему мы должны их обучать? Учебный материал

будет зависеть от технологий, которые используются для выполнения задачи, но более важный вопрос заключается в следующем: какого рода инженеров мы пытаемся подготовить? Поскольку SR-инженеры работают над проектами разного масштаба и разной сложности, они не могут себе позволить быть сконцентрированными только на операционной работе, быть традиционными системными администраторами. В дополнение к тому, что они имеют склад ума, необходимый для решения крупномасштабных задач, SR-инженеры должны иметь следующие характерные особенности.

- По мере выполнения своих задач они столкнутся с системами, которых никогда не видели, поэтому они должны знать, как *выполнять обратное проектирование*.
- При работе в больших масштабах будут появляться аномалии, которые трудно обнаружить, поэтому SR-инженерам нужна способность *мыслить статистически*, а не процедурно, чтобы решать такие задачи.
- Когда стандартные процедуры решения задач не работают, они должны уметь *импровизировать*.

Подробнее рассмотрим эти характеристики, чтобы понять, как можно дать нашим SR-инженерам требуемые знания и навыки.

Обратное проектирование: узнаем, как работают системы

Инженерам интересно узнать, как работают системы, которых они никогда не видели ранее — или, если быть точнее, как работают текущие версии систем, с которыми они взаимодействовали ранее. Имея базовое понимание того, как

работают системы компании, а также подробно изучая процесс отладки, RPC-процедуры и журналы бинарных файлов, SR-инженеры начинают лучше справляться с неожиданными проблемами в системах с неизвестной архитектурой. Обучите своих инженеров основам диагностики и отладки ваших приложений и позвольте им практиковаться в выводе заключений на основе поверхностных данных, чтобы такое поведение вошло в привычку при работе с будущими сбоями.

Статистика и сравнение: последователи научного метода под давлением

Можно рассматривать подход SR-инженеров к реагированию на инциденты в крупномасштабных системах как перемещение по массивному дереву решений. За короткое время, которое выделяется на реагирование, SR-инженеры могут предпринять лишь несколько действий из доступного множества, ставя перед собой цель справиться со сбоем либо в краткосрочной, либо в долгосрочной перспективе. Поскольку времени зачастую мало, SR-инженеры должны эффективно отсечь лишние ветви этого дерева решений. Способность сделать это частично приходит с опытом, который нарабатывается со временем при работе с производственными системами. Параллельно нужно учиться осторожно выдвигать гипотезы, которые по мере их доказательства или неподтверждения еще больше сузят пространство решений.

Говоря другими словами, отслеживание сбоев системы зачастую похоже на игру «Какой предмет не похож на другие?», где в роли «предметов» выступают версии ядра, архитектура ЦП, бинарные версии вашего стека, региональная структура трафика или сотни других факторов. С точки зрения архитектуры команда отвечает за то, чтобы все эти факторы можно было проконтролировать, индивидуально

проанализировать и сравнить. Однако мы также должны обучать наших новых SR-инженеров так, чтобы они стали хорошими аналитиками с самого начала их работы.

Импровизация: когда случается неожиданное

Вы пробуете применить свое решение для того, чтобы исправить сбой, но оно не работает. Найти разработчиков системы, выдавшей сбой, не представляется возможным. Что делать? Импровизировать! Изучите принципы работы нескольких инструментов, которые могут хотя бы частично решить вашу проблему, — это позволит вам попрактиковаться в применении глубокой защиты при создании ваших собственных вариантов поведения для решения подобных проблем. Строгое следование процедурам во время сбоя и, как результат, игнорирование аналитических способностей, может серьезно усложнить поиск основной причины. Ситуация, когда процесс поиска ошибок приостанавливается, может еще больше ухудшиться, если SR-инженер делает слишком много непроверенных предположений о причине сбоя. Это ценный урок, который SR-инженеры должны усвоить как можно раньше.

Какой курс обучения и практические занятия мы можем предложить новым SR-инженерам для того, чтобы направить их по верному пути, учитывая эти три важные характеристики? Вам нужно продумать собственное информационное наполнение учебного курса с учетом всех описанных особенностей в дополнение к другим требованиям, характерным для вашей культуры SRE. Рассмотрим один урок, который, по нашему мнению, учитывает все указанные ранее моменты.

Связываем все воедино: обратное проектирование сервиса, находящегося в промышленной эксплуатации

Когда пришло время изучить (часть стека технологий сервиса Google Maps), (новый SR-инженер) спросила, может ли она вместо пассивного прослушивания лекций сделать это самостоятельно — изучить все с помощью методик обратного проектирования, а остальные потом подскажут ей, что она пропустила. Каков был результат? Ее знания оказались более уместными и полезными, чем они могли бы быть после прослушивания моей лекции, а я обслуживал эту часть сервиса более пяти лет!

Пол Кован, SR-инженер компании Google

Один из первых уроков для новичков должен быть на тему обратного проектирования сервиса, находящегося в промышленной эксплуатации (без помощи его владельцев). Поставленная задача на первый взгляд кажется простой. Все сотрудники сервиса Google News: SR-инженеры, программные инженеры, менеджеры и т.д. — уехали в круиз по Бермудскому треугольнику. Мы ничего не слышали о них уже 30 дней, как и наши ученики, определенные в команду Google News. Новичкам нужно понять, как работает стек сервиса от начала до конца, чтобы управлять им и поддерживать его работу.

Получив эту задачу, ученики выполняют интерактивные упражнения, где отслеживают путь, который проходит запрос

браузера через нашу инфраструктуру. На каждом этапе мы делаем акцент на том, что очень важно разобрать несколько способов исследования взаимодействия между производственными серверами, чтобы соединение не было потеряно. В середине урока мы даем ученикам задание найти другую конечную точку для входящего трафика, показывая, что наше изначальное предположение было слишком узким. Далее мы предлагаем им найти другие пути через стек. Мы используем производственные бинарные файлы, которые самостоятельно сообщают о своей подключаемости к RPC, а также доступные нам системы мониторинга методом черного и белого ящика, позволяющие определить направление запросов пользователей^{[169](#)}. В процессе выполнения этого задания мы строим схему системы и рассматриваем ее компоненты — общую инфраструктуру, которую наши ученики, скорее всего, еще встретят в работе.

В конце урока ученики получают еще одно задание. Каждый ученик возвращается в свою команду и просит более опытного коллегу помочь ему выбрать стек или участок стека, который он будет обслуживать, находясь на дежурстве. Используя навыки, полученные в ходе урока, ученик самостоятельно создает схему этого стека и показывает ее более опытным инженерам. Он наверняка пропустит несколько незначительных деталей, и это послужит основой для обсуждения. При этом опытный инженер в процессе урока и сам узнает что-то новое, пополнив свои знания о постоянно изменяющейся системе. Из-за быстрого изменения производственных систем важно, чтобы ваша команда приветствовала любую возможность повторно ознакомиться с системой.

Пять приемов для вдохновления дежурных работников

Работа на дежурстве — не единственная важная обязанность любого SR-инженера, но производственная разработка обычно подразумевает какое-то количество срочных уведомлений. Человек, способный ответственно отнестись к работе на дежурстве, должен хорошо понимать, как функционирует система. Поэтому мы используем термин «способный работать на дежурстве» как синоним выражения «имеет достаточное количество знаний и может разобраться в том, чего он не знает».

Охотники за сбоями: чтение отчетов о происшествиях и обмен ими

*Тот, кто не помнит историю,
обречен ее повторять.*

Джордж Сантаяна, философ и эссеист

Отчеты о происшествиях, или *постмортемы* (см. главу 15), являются важной частью непрерывного улучшения. Они представляют собой безвинильный способ найти основную причину существенного и видимого пользователю сбоя. При написании такого отчета имейте в виду, что больше всего он пригодится инженеру, который еще даже не нанят. Без радикального переписывания вы можете внести небольшие правки в свои отчеты, чтобы сделать их «обучающими».

Но и самые лучшие отчеты могут оказаться бесполезными, если они прозябают на дне виртуальной базы данных. Из этого следует, что ваша команда должна собирать и курировать особо ценные отчеты для того, чтобы сделать их ресурсами для обучения новичков. Некоторые отчеты созданы по стандартному сценарию, но «обучающие постмортемы»,

способные поведать о структурных дефектах или новых сбоях крупномасштабных систем, ценятся учениками на вес золота.

Отчеты не принадлежат только их авторам. Этим гордятся многие команды, пережившие и задокументировавшие свои самые крупные сбои. Собирайте свои лучшие отчеты и делайте их доступными для новичков, а также для заинтересованных людей из интегрированных команд. В свою очередь, попросите членов этих команд опубликовать свои самые лучшие постмортемы в тех случаях, когда вы можете получить к ним доступ.

Одни команды SR-инженеров в нашей компании создали «Клуб читателей постмортемов», где можно найти восхитительные и поучительные отчеты, а после прочтения — обсудить их. Авторы отчетов могут стать почетными гостями таких заседаний. Другие команды организуют собрания на тему «Сказки о неудачах», где авторы отчетов присутствуют полуформально, пересказывая сбой и, по сути, управляя дискуссией.

Регулярные чтения отчетов и обсуждение сбоев, включая условия запуска и способы миграции, творят чудеса при построении ментальной карты новых SR-инженеров и упрощают понимание производственной среды и поиск ответов на дежурствах. Постмортемы также прекрасно подходят в качестве основы для создания абстрактных сценариев будущих сбоев.

Катастрофа: ролевая игра

Раз в неделю мы проводим собрание. Мы выбираем жертву, которой вручается сценарий — зачастую реальный сценарий,

взятый из анналов истории компании Google. Жертва, которая напоминает мне участника телешоу, говорит ведущему, что бы она сделала/запросила для того, чтобы понять или решить проблему, а ведущий рассказывает жертве, что случится в результате каждого действия. Это похоже на Zork для SR-инженеров. Вы находитесь в лабиринте запутанных консолей наблюдения, все они похожи друг на друга. Вы должны спасти невинных пользователей от попадания в пропасть Избыточной Задержки Обработки Запроса, спасти дата-центры от Практически Гарантированного Краха и спасти нас всех от позора, связанного с Некорректным Отображением Дудлов.

Роберт Кеннеди, бывший SR-инженер для сервиса Google Search и healthcare.gov6 (см. Life in the Trenches of healthcare.gov)

Когда у вас имеется группа SR-инженеров, навыки которых значительно различаются, что вы можете сделать для того, чтобы объединить их и позволить им учиться друг у друга? Как вы описываете культуру SRE и основные свойства вашей команды новичку? Как извещаете опытных сотрудников о новых изменениях и функциональности вашего стека? Команды Google SRE справляются с этими трудностями, придерживаясь проверенной временем традиции: регулярно

разыгрывая катастрофы. Это упражнение еще часто называется «Колесом неудачи» или «Проходом по доске».

В идеале, если эти упражнения станут еженедельным ритуалом, каждый член группы узнает что-то новое. Схема проста и несколько напоминает настольную ролевую игру: гейм-мастер (game master, GM) выбирает двух участников команды как основного и вспомогательного дежурных работников. Эти два инженера во главе с гейм-мастером составляют команду. Объявляется о поступающем запросе, и команда рассказывает, что бы они сделали для того, чтобы сгладить последствия и изучить сбой.

Гейм-мастер тщательно готовит сценарий мероприятия. Он может быть основан на существовавшем ранее сбое, во время которого новые SR-инженеры еще не работали в компании, а старые инженеры могли его забыть. Или, возможно, сценарий — это анализ гипотетической ошибки в новой или готовой к запуску функциональности стека, что делает всех членов группы одинаково неготовыми к ситуации. Или, что еще лучше, коллеги могут найти новый сбой в производственной системе.

На протяжении следующих 30–60 минут основной и вспомогательный дежурные пытаются найти главную причину проблемы. GM с радостью предоставляет дополнительный контекст по мере рассмотрения проблемы, возможно информируя участников (и публику) о том, как выглядят графы на информационной панели системы мониторинга во время сбоя. Если инцидент требует участия другой команды, GM выступает в роли ее участника для проигрывания сценария. Ни один виртуальный сценарий не будет идеальным, поэтому GM может вернуть участников к теме, уводя их в сторону от ложных умозаключений, внося ясность с помощью

дополнительных вводных¹⁷⁰ или задавая неожиданные и острые вопросы¹⁷¹.

Если ваша катастрофическая ролевая игра пройдет успешно, каждый чему-то научится: возможно, познакомится с новым инструментом или приемом, узнает о существовании другого взгляда на решение проблемы или (особенно доставляет удовольствие новым членам команды) осознает, что смог решить проблему этой недели, если его выбрали в качестве участника. При удачном раскладе это упражнение вдохновит участников команды с нетерпением ждать очередной ролевой игры и даже стать гейм-мастером.

Ломаем и чиним по-настоящему

Новичок может узнать о работе SRE, читая документацию, постмортемы или тренируясь. Разыгрывание катастроф поможет вовлечь его в эту игру. Однако опыт, полученный от исправления *реальных* производственных систем, поможет ему еще больше. Этого опыта у него будет предостаточно, как только он начнет работать на дежурстве, но подобное обучение должно начаться еще *до того*, как SR-инженер начнет дежурить. Поэтому предоставьте такой опыт как можно раньше, чтобы развить у ученика рефлексное реагирование, что позволит ему автоматически выбирать нужные инструменты и системы мониторинга компании для решения возникающих проблем.

В таких случаях важно, чтобы все выглядело максимально реалистично. В идеале ваша команда имеет стек технологий, расположенный в нескольких data-центрах и снабжаемый таким образом, что у вас есть как минимум один экземпляр, от которого вы можете отвести реальный трафик и временно использовать его для обучения. Или же вы можете иметь

небольшой тестовый экземпляр в вашем стеке, который можно одолжить на короткое время. Если это возможно, сымитируйте нагрузку, которая напоминает реальный пользовательский/клиентский трафик, а также потребление ресурсов.

Существует множество возможностей для обучения на реальных производственных системах, куда подается сымитированная нагрузка. Опытные SR-инженеры в такой системе сталкиваются с множеством проблем: неверной конфигурацией, утечками памяти, уменьшением производительности, аварийными запросами, нехваткой памяти в хранилищах и т.д. В этой реалистичной, но относительно свободной от риска среде наблюдатели могут управлять набором задач так, чтобы изменить поведение стека, заставив новых SR-инженеров искать различия, определять факторы, способствовавшие случившемуся, и в итоге восстанавливать системы, возвращая их к нормальному поведению.

Вместо того чтобы просить опытного SR-инженера тщательно спланировать определенный тип сбоя, который новички должны будут восстановить, можно двигаться в противоположном направлении: провести упражнение, которое потребует участия всей команды. Например, продолжая работу в хорошо знакомой конфигурации, можно незначительно повредить стек в выбранных узких местах. При этом необходимо наблюдать за работой сервиса до и после поломки с помощью системы мониторинга. Это упражнение любят SR-инженеры, работающие над системой Google Search. Их версия называется «Давайте сожжем поисковой кластер дотла!»

Упражнение выполняется так.

1. В группе мы обсуждаем, какие наблюдаемые характеристики производительности могут измениться, если мы навредим

стеку.

2. Перед тем как вывести из строя систему, мы опрашиваем участников, узнавая их мнения и обоснования прогнозов реакции системы.
3. Мы проверяем предположения и обосновываем варианты поведения, которые наблюдаем.

Это упражнение нужно выполнять ежеквартально — оно позволяет избавиться от новых багов, которые периодически появляются в системе.

Документация как обучение

Многие команды SR-инженеров ведут чек-листы обучения работе на дежурстве — в них содержится структурированная информация и полный список технологий и концепций, связанных с поддерживаемыми системами. Все ученики должны ознакомиться с этим списком перед тем, как работать на теневых дежурствах. Уделите минуту и еще раз взгляните на чек-лист обучения работе на дежурстве, приведенный в табл. 28.1. Этот чек-лист служит различным целям для разных людей.

- **Для учеников:**

- изучение этого списка поможет понять, какие системы важны и почему. Когда ученики поймут это, они смогут перейти к изучению других тем вместо того, чтобы вникать в несущественные детали, которые можно разобрать и позже;

- документ помогает установить границы системы, которую поддерживает их команда.
- **Для менторов и менеджеров:** прогресс ученика, изучающего чек-лист, можно отследить. Чек-лист отвечает на следующие вопросы:
 - над каким разделами вы работали сегодня;
 - какие разделы оказались наиболее трудными.
- **Для всех членов команды:** документ становится социальным контрактом, с помощью которого (усвоив нужные навыки) ученик вступает в ряды дежурных работников. Этот чек-лист является своего рода стандартом, который должны поддерживать все члены команды.

В быстро изменяющихся средах документация может быстро устаревать. Устаревшая документация — это не такая большая проблема для опытных SR-инженеров, которые уже давно вникли во все тонкости системы. Новые специалисты гораздо больше нуждаются в актуальной документации, но они не всегда готовы вносить в нее изменения. При качественном структурировании документация для работы на дежурстве может стать адаптируемым справочником, в котором объединены как энтузиазм новичков, так и знания опытных инженеров, что позволяет сохранять его актуальность.

В команде, отвечающей за работу сервиса Google Search, мы встречаем новых членов команды, предварительно просмотрев чек-лист и отсортировав его разделы по степени актуальности. По мере обучения новых сотрудников мы даем им задание переписать один или два наиболее устаревших раздела. В чек-

листе в каждом разделе указаны контакты опытных SR-инженеров и разработчиков, специализирующихся на той или иной технологии. Мы рекомендуем ученикам связаться с ними, чтобы подробнее разобраться в каждом процессе. Далее, когда они лучше знакомятся с содержимым и стилем чек-листа, они должны переписать определенный раздел, который затем рассматривают один или несколько SR-инженеров, указанных в качестве экспертов.

Периодически проводите теневые дежурства

И все же никакие гипотетические упражнения или другие методы обучения не помогут на 100 % подготовить SR-инженера к работе на дежурстве. В конечном счете работа с реальными сбоями всегда полезнее с точки зрения процесса обучения. Однако несправедливо заставлять новичков ожидать своего первого оповещения, чтобы начать реальное обучение.

Когда новичок изучит все основы системы (например, завершив анализ чек-листа для работы на дежурстве), постарайтесь сконфигурировать вашу систему оповещения так, чтобы она копировала все поступающие вызовы для новичка, поначалу только в рабочее время. Такие теневые дежурства — отличная возможность для ментора взглянуть на прогресс ученика и отличная возможность для ученика получить представление о работе на дежурстве. После того как новичок какое-то время поработает в тени нескольких членов команды, уверенность команды в том, что этот человек готов к дежурствам, будет только расти. Внушение уверенности таким способом — эффективный метод построить доверительные отношения.

Когда поступает очередное экстренное уведомление, новый SR-инженер не становится ответственным за него, и это позволяет ему спокойнее работать. Теперь он основной

наблюдатель, но он наблюдает развитие сбоя, а не его итоговый результат. Вполне возможно, что ученик и основной инженер, находящийся на дежурстве, проводят общую сессию работы в консоли или сидят рядом и делятся своими замечаниями. В удобное для всех время после сбоя дежурный работник может подробно рассмотреть все действия, которые он выполнил. Это упражнение позволит дежурному работнику, находящемуся в тени, лучше понять то, что на самом деле произошло.



Если произойдет сбой, для которого стоит написать постмортем, дежурный работник должен позволить новичку внести в него свою лепту. Но не оставляйте бумажную работу исключительно ученику, поскольку он может (неверно) усвоить, что написание подобных отчетов – грязная работа, которую следует поручать менее опытным сотрудникам.

Некоторые команды также делают финальный шаг: опытный дежурный работник какое-то время находится в тени ученика. Новичок становится основным дежурным инженером и работает со всеми оповещениями, но опытный работник остается в тени, самостоятельно диагностируя ситуацию, ничего не меняя. Опытный SR-инженер всегда сможет поддержать, помочь, подтвердить правильность действий и при необходимости дать совет.

На дежурстве и не только: обряд перехода и практика постоянного обучения

По мере увеличения знаний ученик достигнет точки в карьере, когда сможет комфортно работать с большей частью стека и импровизировать в непонятных ситуациях. К этому моменту он сможет работать на дежурствах. Некоторые команды проводят финальный экзамен для своих учеников перед тем, как наделить их правами и обязанностями дежурного работника. Новые SR-инженеры должны продемонстрировать, что они усвоили чек-лист, чтобы подтвердить свою готовность. Независимо от того, как вы пройдете этот рубеж, выход на дежурство — своеобразный обряд перехода, который должна отпраздновать вся команда.

Прекращается ли обучение, когда ученик начинает работать на дежурстве? Конечно, нет! Для того чтобы оставаться бдительными SR-инженерами, сотрудники вашей команды всегда должны быть активными и учитывать возможные изменения. Если вы будете отвлекаться на другие участки, части вашего стека могут измениться и расшириться, что приведет к устареванию знаний вашей команды.

Проводите регулярные курсы для всей команды, где обзоры новых и грядущих изменений вашего стека будут предоставлены в качестве презентаций от SR-инженеров, наблюдающих за этими изменениями, возможно, вместе с разработчиками. Если можете, запишите эти презентации, чтобы создать обучающую библиотеку для будущих учеников.

Существуют и другие пути для вовлечения в процесс обучения: рассмотрите возможность проведения семинаров, где SR-инженеры будут беседовать с коллегами-разработчиками. Чем лучше ваши коллеги-разработчики понимают вашу работу и трудности, с которыми вы сталкиваетесь, тем проще им будет принимать обоснованные решения для будущих проектов.

Итоги главы

Стартовые инвестиции в обучение SR-инженеров абсолютно стоят того — это нужно как ученикам, которые стремятся получить представление о производственной среде, так и командам, которые будут рады принять новичков в ряды дежурных работников. С помощью приемов, описанных в этой главе, вы быстрее подготовите всесторонне развитых SR-инженеров, постоянно улучшая при этом навыки команды. Только от вас зависит, как именно вы будете применять эти приемы, но задача ясна: как SR-инженер, вы должны «масштабировать» своих людей быстрее, чем вы масштабируете машины. Удачи вам и вашим командам в создании культуры обучения!

[165](#) И не работает!

[166](#) Примеры проактивной работы SR-инженеров: автоматизация ПО, консультирование по вопросам проектирования и координация запусков.

[167](#) Примеры реактивной работы SR-инженеров: отладка, поиск проблем и обработка срочных ситуаций на дежурстве.

[168](#) Дань видеоиграм недавнего прошлого.

[169](#) Такой подход («следуй за RPC») также хорошо работает для систем пакетной обработки/конвейерных систем. Он начинается с операции, которая запускает систему. Для систем пакетной обработки этой операцией может быть появление данных, которые нужно обработать, транзакция, которую нужно подтвердить, или одно из множества других событий.

[170](#) Например: «Вы получаете сообщение на пейджер от другой команды, в котором содержится более подробная информация. Вот что они говорят...»

[171](#) Например: «Мы стремительно теряем деньги! Как вы собираетесь быстро решить эту проблему?»

29. Справляемся с отвлекающими факторами и прерываниями

Автор — Дэйв О'Коннор

Под редакцией Дианы Бейтс

Если речь идет о сложных системах, операционная нагрузка — это работа, которую нужно выполнять для того, чтобы поддерживать систему в функциональном состоянии. Например, если у вас есть машина, вам (или тому, кому вы платите) всегда нужно будет заправлять ее, а также выполнять другие регулярные работы по ее обслуживанию для того, чтобы она продолжала ездить.

Любая сложная система неидеальна, как и ее создатели. При управлении операционной нагрузкой, созданной этими системами, помните, что создатели — такие же неидеальные машины.

Когда идет речь об управлении сложными системами, *операционная нагрузка* принимает множество форм, причем одни из них более очевидны, чем другие. Терминология может изменяться, но операционная нагрузка делится на три общие категории: оповещения, тикеты и продолжительную операционную работу.

Оповещения связаны с производственными уведомлениями и их последствиями. Они появляются в ответ на неотложные ситуации в производственных системах. Иногда они могут быть однотипными, повторяющимися и не требующими особых раздумий. Они также могут быть необычными и требовать серьезного анализа. Оповещения имеют целевое значение (SLO) ожидаемого времени ответа, которое обычно измеряется в минутах.

Тикеты связаны с запросами пользователей, требующих от вас каких-то действий. Как и вызовы, тикеты могут быть как простыми и скучными, так и требующими серьезных размышлений. Простой тикет может потребовать выполнения обзора кода для конфигурации, которой владеет команда. Более сложные тикеты могут повлечь за собой особый запрос, например, на оказание помощи в проектировании или планировании производительности. Тикеты также могут иметь целевые значения, но время ответа на них измеряется в часах, днях или неделях.

Продолжительная операционная работа (также известная как поверхностное решение проблемы или рутинна; см. главу 5) включает в себя такие действия, как отправка кода или ответы на чувствительные ко времени вопросы от клиентов. Несмотря на то что они не имеют определенных целевых значений, эти действия также могут прерывать вашу работу.

Некоторые виды операционной нагрузки можно предугадать или спланировать, но большую ее часть спланировать нельзя — те же поступающие запросы (прерывания) могут появиться в любое время, и инженеру придется определить, может ли проблема подождать.

Управляем операционной нагрузкой

Компания Google разработала несколько методов управления каждым видом операционной нагрузки на уровне команды.

Оповещениями зачастую управляет основной дежурный инженер. Этот человек отвечает на вызовы и управляет возникающими инцидентами или сбоями. Основной дежурный инженер также может отвечать на сообщения пользователей, отправлять сообщения о сбоях разработчикам продукта и т.д. Для того чтобы минимизировать количество поступающих

запросов, которые затрагивают команду, а также избежать эффекта постороннего, на дежурстве работает один инженер. Он может отправлять оповещения другим членам команды, если сам не в состоянии разобраться с проблемой.

Как правило, вспомогательный дежурный инженер выступает в качестве помощника основного. Его обязанности могут меняться. Иногда его единственной задачей является взаимодействие с основным инженером, если экстренные уведомления остаются без ответа. В этом случае вспомогательный инженер может находиться в другой команде. Он может сам выбрать, работать ему с поступающими запросами или нет, в зависимости от его обязанностей.

Тикетами управляют несколькими разными способами в зависимости от команды SR-инженеров: основной дежурный инженер может работать над ними, пока находится на дежурстве, вспомогательный инженер может работать над ними, находясь на дежурстве, или же в команде предусмотрен человек, который отвечает за тикеты и не работает на дежурстве. Тикеты могут автоматически распределяться между участниками команды в случайном порядке, или же члены команды могут просто обрабатывать тикеты по мере их поступления.

Непрерывной операционной работой также можно управлять несколькими способами. Иногда всю работу (установка новой версии и т.д.) делает дежурный инженер. В то же время за каждым членом команды может быть закреплена определенная обязанность, или же дежурный инженер может браться за работу (например, многонедельное обновление), которая длится дольше, чем он работает на дежурстве.

Факторы, определяющие обработку поступающих запросов

Отступая на шаг назад от механизмов управления операционной нагрузкой, рассмотрим несколько показателей, которые влияют на способ обработки каждого поступающего запроса. Некоторые команды SR-инженеров при определении способа управления такими запросами обращают внимание на:

- целевое значение ожидаемого времени ответа;
- количество запросов, которое обычно не выполнено;
- серьезность запросов;
- частоту поступающих запросов;
- количество людей, доступных для обработки определенных видов запросов (например, некоторым командам требуется определенное количество тикетов для того, чтобы начать дежурство).

Вы можете заметить, что все эти показатели подходят для того, чтобы время ответа было минимальным, без привлечения большого количества человеческих ресурсов.

Неидеальные машины

Люди — это неидеальные машины. Им бывает скучно, у них есть плохо изученные процессоры (и иногда даже пользовательский интерфейс), а еще они не очень эффективны. Описание человеческой природы и способов улучшить работу людей не относятся к теме этой книги, поэтому сейчас мы рассмотрим только базовые идеи, которые могут быть полезны для определения того, как нужно реагировать на поступающие запросы.

Состояние когнитивного потока

Концепция *состояния когнитивного потока*¹⁷² широко распространена и может быть эмпирически подтверждена практически каждым, кто работал в области разработки ПО, системного администрирования, в службе SRE или был связан с другими дисциплинами, которые требовали сконцентрированной работы. Будучи в ударе, человек обычно работает более эффективно, а также отличается артистической и научной креативностью. В таком состоянии люди стремятся улучшить задачу или проект, над которыми работают. Поступающий запрос может выбить вас из этого состояния, если он достаточно серьезный. Конечно, вы предпочтете уменьшить количество таких запросов.

Когнитивный поток также может быть актуален и для менее креативных занятий, где требуется меньший уровень навыков, но при этом возможны четкие цели, мгновенная обратная связь, ощущение контроля и соответствующее искажение времени; в качестве примера можно привести выполнение работы по дому или вождение.

Вы можете достичь оптимального уровня концентрации внимания, работая над нетрудными задачами, не требующими особых навыков, например играя в повторяющуюся видеоигру. Так же легко вы можете войти в это состояние, выполняя трудные задачи, требующие значительных навыков, например, такие, с которыми сталкивается инженер. Методы входления в состояние когнитивного потока могут различаться, но результат останется тем же.

Состояние когнитивного потока: креативность и вовлечение

Когда человек работает над задачей какое-то время, он хорошо понимает особенности проблемы и, скорее всего, в состоянии

ее решить. Будучи в состоянии подъема, инженер может неотрывно работать над проблемой, теряя счет времени и максимально игнорируя поступающие запросы. Очень желательно максимизировать количество времени, которое работник может провести в этом состоянии, — так больше шансов получить креативные результаты и успеть выполнить задание в срок. В то же время специалист будет чувствовать удовлетворение от выполнения своей работы.

К сожалению, множество людей в роли SR-инженеров либо стремятся войти в это состояние, а у них не получается и они выходят из равновесия, либо никогда не пытаются достичь его, вместо этого страдая из-за постоянных прерываний.

Состояние когнитивного потока: Angry Birds

Людям нравится решать такие задачи, которые им понятны. Фактически выполнение такой работы — один из простейших путей к достижению состояния когнитивного потока. Некоторые SR-инженеры входят в это состояние, находясь на дежурстве. Так можно очень эффективно отслеживать источники проблем, работать с коллегами и улучшать общее состояние системы.

И наоборот, для большинства раздраженных инженеров, находящихся на дежурстве, стресс вызывается либо количеством экстренных уведомлений, либо тем, что они считают дежурства отвлекающим от основной работы фактором. Они пытаются писать код или на полную ставку работать над проектами, одновременно дежурия или реагируя на поступающие запросы. В итоге эти инженеры постоянно *работают в режиме прерывания*, что, несомненно, приводит к стрессам.

С другой стороны, когда человек полностью концентрируется на прерываниях, они *перестают быть прерываниями* в прямом смысле этого слова. На очень глубоком уровне постепенное усовершенствование системы, выполнение требований тикетов, а также исправление проблем и сбоев становится четким набором целей, границ и прозрачной обратной связи: вы закрываете X багов или вам перестают поступать экстренные уведомления. Все остальное — это раздражители. Когда вы работаете с прерываниями, ваши проекты становятся раздражителями. Даже несмотря на то, что работа с прерываниями может быть удовлетворительной в краткосрочной перспективе, в смешанной среде, где одновременно приходится работать над проектами и дежурить, люди чувствуют себя счастливее, если сбалансированно тратят свое время на эти два вида работы. У каждого инженера будет свой показатель идеального баланса. Надо лишь учитывать, что некоторые инженеры могут не знать, какое именно сочетание этих видов работы может их мотивировать (или они могут думать, что знают, но вы придерживаетесь другого мнения).

Делаем хорошо что-то одно

Вы можете задуматься о том, чтобы применить прочитанное на практике. Следующие рекомендации основаны на том, что сработало для разных команд SR-инженеров, которыми я управлял в компании Google, и предназначены скорее для удобства менеджеров команды или руководителей. Этот список не затрагивает индивидуальные особенности человека — люди могут управлять своим свободным временем так, как посчитают нужным.

Рассеянность

Существует множество вещей, способных отвлечь инженера и вывести его из состояния воодушевления. Например, рассмотрим случайного SR-инженера по имени Фред. Фред приходит на работу утром в понедельник. Сегодня он не дежурит и не работает с прерываниями, поэтому он планирует заняться своими проектами. Он делает себе кофе, надевает свои наушники марки «Не беспокоить» и садится за свой стол. Пришло время почувствовать себя в ударе, верно? Не совсем, ведь в любой момент может случиться что-нибудь неожиданное.

- Команда Фреда использует систему автоматического распределения тикетов, случайным образом назначающую его ответственным за тикет, который нужно закрыть сегодня.
- Коллега Фреда сегодня работает на дежурстве, получает экстренное уведомление, связанное с компонентом, в котором Фред отлично разбирается, и отвлекает его, задав вопрос об этом компоненте.
- Пользователь сервиса, за который отвечает Фред, повышает приоритет тикета, адресованного Фреду на прошлой неделе, когда тот находился на дежурстве.
- Подготовка нового релиза, который будет выпущен через 3–4 недели и за который отвечает Фред, выполняется неверно, что заставляет Фреда бросить все и проверить релиз, откатить изменение и т.д.
- Пользователь сервиса, за который отвечает Фред, связывается с ним, чтобы задать вопрос, потому что Фред всегда так любезен!

- И т.д.

В итоге, несмотря на то что у Фреда был целый календарный день для работы над проектами, он не смог сосредоточиться. Одними отвлекающими факторами он мог управлять, закрыв электронную почту, отключив сервисы обмена сообщениями и пр. Другие вызваны политикой компании или соглашениями о прерываниях и продолжительных обязанностях.

Вы можете сказать, что некоторых отвлекающих факторов нельзя избежать по умолчанию. Это так: у каждого инженера есть своя зона ответственности и обязательства. Однако существуют способы, с помощью которых команда может управлять ответами на прерывания, чтобы большинство инженеров могли приходить утром на работу и *спокойно трудиться*.

Распределение времени

Для того чтобы свести к минимуму свою отвлекаемость, вы должны минимизировать количество переключений контекста. Некоторых прерываний избежать невозможно. Однако нельзя рассматривать инженера как работника, который может легко отвлекаться, и это никак не повлияет на его основную деятельность. Определите для себя стоимость переключений контекста. Двадцатиминутное прерывание при работе над проектом влечет за собой два переключения контекста, ведь на самом деле это прерывание приведет к потере нескольких часов действительно продуктивной работы. Чтобы избежать постоянной потери производительности, нацельтесь на распределение времени между разными видами работы, максимально увеличивая продолжительность каждого периода.

В идеале таким периодом должна стать неделя, но день или полдня могут быть даже более подходящими. Эта стратегия также вписывается в концепцию *потерянного времени* [Graham, 2009].

Распределение времени означает, что, приходя на работу, человек должен знать, над чем он сегодня работает: *только* над проектом или *только* над прерываниями. Распределение времени таким способом означает, что инженеры смогут концентрироваться на работе над задачей на протяжении длительного периода. Они не будут раздражаться из-за того, что приходится переключаться между задачами, отвлекающими их от работы, которую они должны выполнять на самом деле.

Серьезно, скажите, что мне делать

Если общая модель, представленная в этой главе, вам не подходит, предлагаю несколько конкретных рекомендаций. Все дальнейшие идеи вы можете реализовать поэтапно.

Общие рекомендации

Если объем прерываний любого класса слишком велик для одного человека, *добавьте еще одного исполнителя*. Эта концепция наиболее очевидно применяется к тикетам, но потенциально ее можно применить и к вызовам — основной инженер может начать действовать вспомогательного или же понижать вызовы до тикетов.

Дежурство

Основной инженер, работающий на дежурстве, должен концентрироваться исключительно на дежурстве. Если

пейджер вашего сервиса молчит, то частью ваших обязанностей как дежурного должны стать тикеты или другая работа, основанная на прерываниях, которую можно относительно быстро прекратить. Если инженер дежурит на протяжении недели, эта неделя должна быть исключена из табеля времени работы над проектом. Если проект слишком важен и не может ждать неделю, этот человек не должен находиться на дежурстве. В таком случае назначьте дежурным кого-то другого. *Не стоит ожидать от людей того, что они будут дежурить и одновременно продвигаться в работе над проектом (или над чем-то еще, что имеет высокую стоимость переключения контекста).*

Обязанности вспомогательного инженера зависят от того, насколько обременительна работа на дежурстве. Если его функция заключается в помощи основному в случае неудачи, то вспомогательный инженер наверняка может работать и над проектом. Если кто-то, кроме вспомогательного инженера, работает над тикетами, рассмотрите возможность объединения ролей. Если вспомогательный инженер на самом деле должен помогать основному из-за большого количества вызовов, то он должен работать и с прерываниями.

(Ремарка: *работы по уборке всегда будет предостаточно*. Количество тикетов может быть равно нулю, но всегда будет документация, которую нужно обновить, конфигурация, которую нужно очистить, и т.д. Ваши будущие дежурные работники только поблагодарят вас, а значит, скорее всего, не будут вас беспокоить.)

Тикеты

Если в данный момент вы присваиваете тикеты инженерам случайным образом, остановитесь. Такое отношение к времени

вашей команды крайне неуважительно, оно полностью отменяет принцип минимальных прерываний.

Работа над тикетами должна выполняться на протяжении рабочего дня в таких объемах, с которыми может справиться один человек. Если получается так, что количество тикетов превышает совместные возможности основного и вспомогательного инженеров, организуйте очередь тикетов так, чтобы в любой момент времени у вас было два человека, работающих с ней. Люди не машины, вы лишь отвлекаете их от основной работы, отнимая драгоценное время.

Текущие задачи

Определите роли в команде так, чтобы любой человек мог взяться за текущие задачи. Если существует хорошо определенная процедура для выполнения и проверки новых версий кода, то нет необходимости заставлять кого-то одного постоянно выполнять эту работу даже после того, как он перестает дежурить или обрабатывать прерывания. Определите роль *push-менеджера* (отвечающего за установку новой версии), который сможет управлять релизом во время дежурства или работы с прерываниями. Формализуйте процесс передачи — это небольшая цена за то, чтобы остальные люди, не работающие на дежурстве, имели возможность спокойно выполнять свои обязанности.

Работать или не работать с прерываниями

Иногда команда получает прерывание, для обработки которого лучше всего подходит человек, не работающий с прерываниями в данный момент. Хотя в идеале такая ситуация никогда не должна возникать, так все же случается. Вы должны поработать над тем, чтобы это происходило как можно реже.

Бывает, что человек работает с тикетами, в то время как не должен этого делать, — хочет создать видимость занятости. Это говорит о том, что такой работник недостаточно эффективен. В итоге искажается реальное представление о том, насколько управляемым является количество тикетов. Если для работы с тикетами назначен один человек, но два или три других человека также начнут с ними разбираться, ваша очередь тикетов все еще может оставаться неконтролируемой, хотя вы этого и не заметите.

Уменьшение количества прерываний

Количество прерываний, с которыми сталкивается ваша команда, может быть неконтролируемым, если они требуют от команды постоянного внимания в любой момент времени. Существует несколько приемов, позволяющих уменьшить общее количество тикетов.

Анализируйте тикеты

Большая часть дежурных инженеров, работающих с тикетами, воспринимают их как наказание. Это особенно верно для крупных команд. Если вы работаете с прерываниями всего раз в несколько месяцев, очень легко пробежаться по ним [173](#), облегченно вздохнуть и вернуться к своим обычным обязанностям. Далее сменившие вас коллеги сделают то же самое, и никто не станет разбираться в основных причинах тикетов: вместо того чтобы двигаться вперед, ваша команда погрязнет в рутине из-за того, что людей раздражают одни и те же проблемы.

Должен быть предусмотрен процесс передачи тикетов, а также работы на дежурстве. Процесс передачи предусматривает распределение обязанностей между

обработчиками тикетов по мере переключения ответственности. Даже минимальная самодиагностика основных причин может дать хорошие результаты для уменьшения общего количества тикетов. Многие команды выполняют эстафетную передачу дежурства и делают сводку экстренных уведомлений. Очень немногие команды делают то же самое для тикетов.

Ваша команда должна регулярно избавляться от тикетов и экстренных уведомлений, при этом нужно проверять группы прерываний, чтобы увидеть, сможете ли вы определить их основную причину. Если вы думаете, что от главного источника прерываний можно избавиться в разумный период времени, то вам следует *приглушить остальные прерывания до тех пор, пока основная причина не будет устранена*.

Уважайте себя и своих клиентов

Это правило больше применимо к пользовательским, а не к автоматическим прерываниям, несмотря на то что принципы актуальны для обоих случаев. Если тикеты особенно раздражают или с ними трудно справляться, вы можете изменить линию поведения для облегчения своей ноши.

Помните:

- ваша команда задает уровень обслуживания, предоставляемый вашим сервисом;
- допустимо перекладывать какие-то обязанности на клиентов.

Если ваша команда отвечает за обработку тикетов или прерываний для клиентов, вы можете сделать свою рабочую нагрузку более управляемой. Это решение может быть как

временным, так и постоянным, в зависимости от того, что имеет больший смысл. Так вы определите баланс между уважением к клиенту и уважением к себе.

Например, если вы поддерживаете особенно капризный инструмент, для которого практически нет ресурсов разработчика, и с ним работает небольшая группа требовательных клиентов, рассмотрите другие варианты. Подумайте о времени, которое вы потратите на обработку прерываний для этой системы, и ответьте на вопрос, будете ли вы тратить его оптимально. Если в какой-то момент вы не сможете уделить системе достаточно внимания, чтобы исправить основную причину прерываний, возможно, компонент, который вы поддерживаете, не так уж и важен. Вы должны рассмотреть возможность возврата пейджера, отключения компонента, его замены или другую подобную стратегию, которая может иметь смысл.

Если конкретные шаги по исправлению ошибки требуют большого количества времени или запутаны, но не требуют ваших привилегий для их завершения, рассмотрите возможность отправки запроса обратно. Например, если люди должны потратить вычислительные ресурсы, подготовьте код или изменение конфигурации или сделайте аналогичный шаг, а затем скажите клиенту выполнить этот шаг и проверьте его работу. Помните, что, если клиент хочет, чтобы некоторая задача была выполнена, он должен быть готов приложить усилия для того, чтобы получить желаемое.

Подводным камнем таких решений является необходимость поиска баланса между уважением к клиенту и уважением к себе. При создании стратегии работы с запросами клиентов руководствуйтесь тем, что запрос должен быть осмысленным, а вы должны предоставить всю информацию, необходимую для

его выполнения. Ваш ответ должен быть полезным и своевременным.

172 См. в «Википедии»: Flow (psychology),
[http://en.wikipedia.org/wiki/Flow_\(psychology\)](http://en.wikipedia.org/wiki/Flow_(psychology)). Потоком называют оптимальный уровень концентрации внимания.

173 См. http://en.wikipedia.org/wiki/Running_the_gauntlet.

30. Добавляем в команду нового SR-инженера, чтобы предотвратить операционную перегрузку

Автор — Рэндэлл Босетти

Под редакцией Дианы Бейтс

Стандартная политика компании Google для команд SR-инженеров заключается в том, чтобы одинаково распределять их время для работы над проектами и операционными задачами. На практике этот баланс может быть нарушен из-за увеличения количества тикетов. Чрезмерное количество операционной работы особенно опасно, поскольку команда SR-инженеров может перегореть или не иметь возможности продвигаться в работе над проектами. Когда команда обязана выделять слишком много времени на разрешение тикетов ценой времени, которое можно было бы потратить на работу над сервисом, страдает масштабируемость и надежность.

Один из способов уменьшить это бремя — временно добавить нового SR-инженера в перегруженную команду. Попав в эту команду, он концентрируется на улучшении ее приемов вместо того, чтобы просто помогать им разбираться с тикетами. SR-инженер наблюдает за ежедневной рутиной команды и дает рекомендации для повышения эффективности их работы. Такие консультации помогают команде по-новому взглянуть на рутину, чего они не могут сделать самостоятельно.

Используя этот подход, не обязательно переводить нескольких инженеров. Если вы введете двоих новых специалистов, то это может дать даже обратный эффект и привести к проблемам.

Если вы создаете свою первую команду SR-инженеров, то советы из этой главы помогут вам избежать ее превращения в команду операционистов, которая работает исключительно с тикетами. Если вы решили, что вы или ваш коллега хотели бы поработать в команде, уделите время тому, чтобы ознакомиться с практическими рекомендациями и принципами работы SR-инженеров, описанными во введении Беном Трейнором Слоссом, а также с информацией о мониторинге из главы 6.

В следующих разделах приводятся практические рекомендации для SR-инженеров, которым предстоит поработать в другой команде.

Фаза 1: изучаем сервис и рабочее окружение

Пока вы находитесь в другой команде, ваша задача будет заключаться в том, чтобы разъяснить, почему привычные сценарии работы влияют (иногда отрицательно) на масштабируемость сервиса. Напомните команде, что большее количество тикетов не должно требовать участия большего числа SR-инженеров: цель модели SRE заключается в том, чтобы вводить новых людей только в том случае, если усложняется сервис. Вместо этого попробуйте обратить внимание на то, как полезные новые навыки снижают время, за которое выполняются тикеты. Сделать это так же важно, как и указать на пропущенные возможности по автоматизации или упрощению сервиса.

Операционная работа против нелинейного масштабирования

Термин «операционная работа» характеризует определенный метод поддержания сервиса в рабочем состоянии. Различные

рабочие элементы увеличиваются с разрастанием сервиса. Например, сервису по мере его роста нужен способ увеличить количество сконфигурированных виртуальных машин (virtual machines, VM). Команда, выполняющая операционную работу, отвечает увеличением количества администраторов, управляющих этими VM. SR-инженеры вместо этого концентрируются на написании ПО или избавлении от проблем с масштабируемостью, чтобы количество людей, необходимое для работы сервиса, не увеличивалось согласно функции увеличения нагрузки на сервис.

Команда, переходящая в режим выполнения операционной работы, может быть уверена, что масштаб не имеет для них значения («мой сервис крошечный»). Проведите сессию теневых дежурств, чтобы определить, верна ли эта оценка, поскольку даже незначительное масштабирование может повлиять на вашу стратегию.

Если основной сервис важен для бизнеса, но он на самом деле невелик (что влечет за собой малое потребление ресурсов или низкую сложность), сосредоточьтесь на том, что мешает команде повысить его надежность при текущем подходе. Помните, что ваша работа заключается в том, чтобы заставить сервис работать, а не оградить разработчиков от оповещений.

С другой стороны, если сервис только запускается, сосредоточьтесь на способах подготовки команды к его резкому увеличению. Сервис, получающий 100 запросов в секунду, уже через год превратится в сервис, который получает 10 000 запросов в секунду.

Определите главные источники стресса

Команды SR-инженеров зачастую переходят в режим выполнения операционной работы из-за того, что концентрируются на том, как быстро они реагируют на неотложные ситуации, вместо того, чтобы снижать количество таких ситуаций. Переход в режим выполнения операционной работы обычно происходит в ответ на огромное давление, реальное или *воображаемое*. После того как вы изучите сервис настолько, чтобы задавать сложные вопросы о его проекте и развертывании, потратьте немного времени на то, чтобы приоритизировать некоторые виды сбоев сервиса согласно их влиянию на уровень стресса команды. Имейте в виду, что некоторые незначительные проблемы могут повлечь за собой большое количество стресса.

Определите очаги возгорания

Как только вы определите основные проблемы команды, займитесь анализом возможных неотложных ситуаций. Иногда неминуемые неотложные ситуации появляются в виде новых подсистем, которые не могут управлять собой. К другим источникам проблем можно отнести следующие.

- *Пробелы в знаниях.* В крупных командах люди бывают чересчур узкоспециализированными. В таких случаях им может не хватать широких знаний, требуемых для выполнения работы на дежурстве. Кроме того, из-за этого инженер может проигнорировать критически важные участки системы, за которую они отвечают.
- *Сервисы, разработанные SR-инженерами, важность которых повышается.* Эти сервисы обычно получают меньше внимания по мере запуска новой функциональности,

поскольку их масштаб меньше, и они явно поддерживаются как минимум одним SR-инженером.

- *Сильная зависимость от «следующего прорыва».* Люди могут игнорировать проблемы месяцами, поскольку они верят, что новое решение, которое скоро будет реализовано, устранит все недостатки.
- *Распространенные оповещения, которые не диагностируются ни командой разработчиков, ни SR-инженерами.* Такие оповещения часто считаются *временными*, но они все еще могут отвлекать ваших коллег от исправления реальных проблем. Вам следует либо полностью разобраться с ними, либо исправить правила оповещения.
- *SLI/SLO/SLA.* См. главу 4, в которой рассматриваются показатели SLI, SLO и SLA.
- *Любой сервис, чей план производительности выглядит как «Добавьте еще серверов: у наших серверов этой ночью закончилась память».* Планы производительности должны быть достаточно перспективными. Если модель вашей системы прогнозирует, что серверу нужно 2 Гбайт памяти, то нагрузочный тест, который проходит в краткосрочной перспективе (показывая значение 1,99 во время последнего запуска), не обязательно означает, что ваша система имеет достаточную производительность.
- *Постмортемы, которые рекомендуют лишь откат изменений, вызвавших сбой.* Например, «Измените потоковый тайм-аут обратно на 60 секунд» вместо «Определите, почему иногда требуется 60 секунд на то, чтобы получить первый мегабайт наших промовидеороликов».

- Любой критически важный для обслуживания компонент, на все вопросы о котором SR-инженеры отвечают: «Мы ничего об этом не знаем, им владеют разработчики». Для того чтобы предоставить приемлемую поддержку компонента на дежурстве, вы должны хотя бы знать о том, что произойдет, когда он перестанет выполнять свои функции, а также о сроках, в которые нужно исправить проблему.

Фаза 2: делимся контекстом

Определив взаимоотношения в команде и уязвимые места, подготовьте почву для изменений с помощью постмортемов, а также предложите новые способы избавления от рутины.

Напишите для команды хороший постмортем

Отчеты о происшествиях дают много информации о совместных обсуждениях команды. Отчеты, написанные несработавшимися командами, зачастую неэффективны. Некоторые члены команд могут считать выполнение постмортемов наказанием или даже бесполезным занятием. Даже если вы захотите пересмотреть архивные отчеты и дать свои советы по их улучшению, лучше этого не делать. Вместо того чтобы пойти вам навстречу, члены команды могут воспринять вашу идею в штыки.

Лучше всего, если вы сами напишете следующий постмортем. Пока вы будете работать в другой команде, какой-нибудь сбой обязательно случится. Если вы в этот момент не дежурите, объединитесь с дежурным инженером и напишите хороший, безобвинительный постмортем. Этот документ прекрасно продемонстрирует, что переход к SRE-модели поможет команде и упростит исправления ошибок. Чем

меньше приходится вносить исправлений, тем меньше времени сбои отнимают у членов команды.

Как уже упоминалось, вы можете встретить возражения вроде «Почему я?». Наиболее вероятно, что вы услышите это, если команда считает, что написание постмортемов — сущее наказание. Такое отношение появляется из-за теории «избавления от паршивых овец»: система в порядке, и, если мы избавимся от «паршивых овец» и ошибок, связанных с ними, система продолжит оставаться в порядке. Эта теория неверна, о чем свидетельствуют [Dekker, 2014] несколько дисциплин, включая авиационную безопасность. Вы должны указать на ее неверность. В частности, можно сказать следующее: «Нельзя избежать ошибок в системах с таким большим количеством взаимодействий. Вы дежурили, и я верю, что вы принимаете правильные решения на основе верной информации. Я бы хотел, чтобы вы описали все, о чем думали в каждый момент времени, и тогда мы сможем понять, почему система ввела вас в заблуждение и где требования были слишком жесткими».

Сортируйте перегрузки в соответствии с их типами

В упрощенной для удобства модели приводятся два вида перегрузок.

- Одни перегрузки не должны происходить. Они вызывают то, что часто называется операционной работой или рутиной (см. главу 5).
- Другие перегрузки, которые вызывают стресс и/или приводят к гневной переписке, на самом деле являются частью работы. В любом случае команде нужны инструменты для того, чтобы управлять ситуацией.

Разделите перегрузки на рутину и не рутину. Когда вы закончите, представьте этот список команде и четко объясните, в каком случае работу нужно автоматизировать, а в каком это допустимая ситуация, связанная с работой сервиса.

Фаза 3: навязываем изменения

Добиться установления здоровых отношений в команде непросто. Для того чтобы научить членов команды подстраиваться друг под друга, вы можете построить для них специальную ментальную модель.



Люди всегда стремятся к поддержанию баланса, поэтому сконцентрируйтесь на создании (или восстановлении) правильных начальных условий и обучите их небольшому набору принципов, необходимых для принятия верных решений.

Начните с основ

Команды, которые не могут понять разницу между работой SRE и традиционных операционистов, зачастую не способны выразить, почему их беспокоят некоторые особенности программы, процессов или культуры команды. Вместо того чтобы пытаться разобрать эти моменты шаг за шагом, вам следует обратиться к принципам, описанным в главах 1–6.

Ваша первая цель заключается в определении целевого уровня обслуживания (SLO), если он еще не определен. Наличие SLO очень важно, поскольку он представляет собой

количественный показатель влияния сбоев и описывает, насколько изменение процесса может повлиять на работу сервиса. SLO — это, возможно, самый важный рычаг, позволяющий переключить команду от активной операционной работы к спокойной и длительной работе в качестве SRE. В ином случае ни один совет из этой главы не будет полезен. *Если вы обнаружили, что попали в команду, где не заданы SLO, прочтите сначала главу 4, а затем соберите в одной комнате технических лидеров и менеджеров и обсудите проблему.*

Получите помощь в исправлении ошибок

У вас может появиться желание сразу решать проблемы, которые вы диагностируете. Пожалуйста, воздержитесь от этого и лучше сделайте следующее.

1. Найдите полезную работу, которую может выполнить один член команды.
2. Четко объясните ему, как эта работа решает проблему, указанную в постмортеме. Даже четко работающие в остальном команды могут описывать в отчетах недальновидные действия.
3. Выполняйте обзоры изменений кода и редакций документов.
4. Повторите эти шаги для 2–3 проблем.

Когда вы определяете дополнительную проблему, поместите ее в отчет об ошибках или другой подобный документ для того, чтобы проконсультироваться с командой.

Это позволяет достичь двух целей: распространить информацию и вдохновить членов команды вести соответствующую документацию (это часто игнорируют из-за дедлайнов). Всегда обосновывайте свои аргументы и делайте акцент на том, что хорошая документация гарантирует, что команда не повторит свои старые ошибки в слегка изменившейся ситуации.

Обосновывайте свои аргументы

По мере того как команда наладит свою работу и усвоит базовые принципы предложенных вами изменений, переходите к разбору повседневных решений, которые привели к операционной перегрузке. Подготовьтесь к тому, что это будет трудной задачей. Если вам повезет, то вы услышите: «Объясни почему. Прямо сейчас. Во время еженедельного собрания, связанного с производственными системами».

Если вам не повезет, никто не потребует объяснения. Вы можете избежать этой проблемы, в любом случае аргументируя свои решения, независимо от того, требует ли кто-то объяснений. Ссыльайтесь на базовые сведения, которые подчеркивают ваши предположения. Это позволит сформировать ментальную модель команды. После того как вы покинете команду, ее члены будут способны спрогнозировать ваши потенциальные комментарии о проекте или списке изменений. Если вы не будете обосновывать свои идеи или сделаете это плохо, появится риск того, что команда просто проигнорирует ваши слова, поэтому четко выражайте мысли.

Вот примеры тщательных объяснений своих решений.

- «Я откатываю изменения в последнем релизе не потому, что тесты прошли плохо. Я откатываю их потому, что вы истратили весь бюджет ошибок для релизов».

- «Релизы должны быть созданы так, чтобы можно было легко выполнить их откат, поскольку наши целевые значения строго определены. Соответствие этим требованиям означает, что у нас мало времени на восстановление, поэтому нереально выполнить глубокую диагностику перед откатом».

Примеры плохих объяснений своих решений приведены ниже.

- «Я не думаю, что будет безопасно заставить каждый сервер генерировать собственную конфигурацию маршрутизации, поскольку мы не можем ее увидеть».

Это решение может быть верным, но оно имеет слабое (или плохо аргументированное) обоснование. Команда не может читать ваши мысли, поэтому, скорее всего, они просто отметят вашу неубедительную аргументацию. Вместо этого вам следует сказать: «...небезопасно, поскольку ошибка в этом коде может повлечь сбои во всем сервисе, и дополнительный код может стать источником новых ошибок, которые способны замедлить откат».

- «Автоматизация не должна прерываться, если она встретит несовместимое развертывание».

Как и в предыдущем случае, это объяснение может и корректно, но его недостаточно. Вместо этого вам следует сказать следующее: «...поскольку мы делаем упрощающее допущение о том, что все изменения проходят автоматизацию, и что-то явно нарушило это правило. Если

это случается часто, нам нужно определить источники неорганизованных изменений и избавиться от них».

Задавайте наводящие вопросы

Наводящие вопросы — это не вопросы с подвохом. Когда вы беседуете с командой SR-инженеров, старайтесь задавать вопросы, которые стимулируют людей задуматься о базовых принципах. Если на обсуждение ваших аргументов будет потрачено достаточно времени, такое обсуждение поможет всей команде лучше усвоить философию SRE.

Примеры наводящих вопросов:

- «Я вижу, что оповещение TaskFailures часто запускается, но дежурные инженеры зачастую ничего не делают для того, чтобы на него отреагировать. Как это влияет на SLO?»;
- «Эта процедура запуска сервиса выглядит слишком сложной. Знаете ли вы, зачем нужно выполнять так много обновлений конфигурационных файлов при создании новых экземпляров сервиса?».

Примеры некорректных наводящих вопросов:

- «Что не так со всеми этими старыми релизами?»;
- «Почему Frobnilzer так много всего делает?».

Итоги главы

Положения, показанные в этой главе, предоставят команде SR-инженеров следующее:

- техническую, возможно, численную перспективу того, почему им следует измениться;
- яркий пример того, какими могут быть изменения;
- логическое объяснение большого количества «народной мудрости», используемой SR-инженерами;
- основные принципы, которые нужны для решения новых проблем и которые могут масштабироваться.

Ваша финальная задача заключается в том, чтобы написать отчет о проделанной работе. В этом отчете должны быть отражены ваши мнения, примеры и объяснения. Также в нем должны быть перечислены действия для команды, которые гарантируют, что они будут практиковаться в том, чему вы их научили. Вы можете организовать поствитам¹⁷⁴, объясняя критически важные решения, принятые на каждом шаге на пути к успеху.

Хотя ваша работа по участию в развитии команды закончится, вы должны продолжать участвовать в обзорах проектов и кода. Следите за командой еще несколько месяцев для того, чтобы убедиться, что они все лучше планируют производительность и процессы выпуска новых версий, а также реагируют на аварийные и критические ситуации.

¹⁷⁴ Postvitam — в противоположность postmortem — постмортему.

31. Общение и взаимодействие в службе SRE

Автор — Нейл Мёрфи при участии Алекса Родригеса, Карла Крауса, Дарио Френи, Диланы Керли, Лоренцо Бланко и Тодда Андервуда

Под редакцией Бетси Бейер

Служба SRE занимает особое место в организационной структуре Google, и это влияет на наше общение и взаимодействие.

В первую очередь, у службы SRE существует множество разнообразных обязанностей, а также множество способов их выполнения. У нас есть инфраструктурные команды, команды, работающие с сервисами, и горизонтальные команды разработки. У нас налажены связи с командами разработчиков продуктов, размер которых во много раз превышает размер нашей команды, и с командами меньше нашей. Иногда возникают ситуации, когда мы сами становимся командой разработчиков. Команды SR-инженеров создаются из людей, имеющих опыт системной и структурной разработки (см. [Hixson, 2015b]), разработки ПО, а также управления проектами. У них есть задатки лидера, они обладают знаниями, полученными из различных отраслей (см. главу 33) и т.д. У нас нет конкретной рабочей модели, мы нашли множество работающих конфигураций, и эта гибкость идеальна для нашей деятельности.

Верно и то, что служба SRE не является административно-управленческой организацией. Как правило, команды, работающие с сервисами и инфраструктурой, тесно взаимодействуют с соответствующими командами разработчиков; кроме того, мы вместе работаем в контексте SRE. Отношения между нашими службами неразрывны, поскольку мы отвечаем за производительность наших систем, и, можно сказать, что мы являемся одной командой. Сегодня

мы тратим больше времени на поддержку наших отдельных сервисов, а не на кросс-производственную работу, но наша культура и общие ценности формируют строгие подходы к решению проблем.

Если охарактеризовать общение и взаимодействие между службами, то уместной метафорой мог бы стать поток данных: данные должны «протекать» через всю действующую систему, они же должны «течь» через команду SR-инженеров — это информация о проектах, состоянии сервисов, систем и отдельных элементов. Для того чтобы команда была максимально эффективной, данные должны надежно поступать от одной заинтересованной стороны к другой. Для визуализации этого потока вообразите себе интерфейс, который команда SR-инженеров должна представить другим командам, что похоже на API. Как и в случае API, хороший проект критически важен для эффективной работы, и, если API написан неверно, впоследствии его может быть трудно исправить.

Метафора «API как контракт» также уместна для взаимодействий как между командами SR-инженеров, так и между командами SR-инженеров и разработчиков продукта — все они должны выполнять свою работу в условиях постоянно появляющихся изменений. Поэтому наши взаимодействия выглядят так же, как и взаимодействия любой другой динамично развивающейся компании. Разница заключается в сочетании навыков проектирования ПО, разработки систем и опыта, полученном в ходе работы с производственными системами, которыми SR-инженер пользуется для того, чтобы участвовать в этом взаимодействии. Лучшие проекты и лучшая реализация получаются при работе в атмосфере взаимного уважения. В этом и заключается потенциал службы SRE: организация отвечает за надежность, имея те же навыки, что и

команда разработчиков, что способствует повышению качества наших программ. Из опыта можно предположить, что специалиста, который отвечает за надежность, но не имеет полного набора навыков, будет недостаточно.

Общение: производственные совещания

Хотя уже написано множество книг на тему эффективных совещаний [Krattenmaker, 2008], очень трудно найти кого-то, кто участвует *только* в полезных и эффективных совещаниях. Это особенно верно для SRE.

Надо признать, что существуют действительно полезные совещания по сравнению с остальными — они называются *производственными*. Это совещания, где SR-инженеры четко формулируют себе — и остальным участникам — информацию о состоянии сервиса, за который они отвечают. Это улучшает общую осведомленность всех заинтересованных лиц, а также повышает качество работы службы. Как правило, эти совещания *связаны с сервисами*. Они посвящены не только изложению информации о состоянии дел, предназначеннной для конкретных людей, — их цель заключается в том, чтобы каждый, покидая совещание, понимал, что происходит. Еще одной важной целью производственных совещаний является улучшение наших сервисов путем применения знаний, полученных на производстве. Это означает, что мы подробно рассматриваем производительность нашего сервиса и связываем ее с проектом, конфигурацией или реализацией, после чего даем рекомендации, как исправить возникшие проблемы. Связывание производительности сервиса с решениями, принятыми во время проектирования, на обычном совещании — очень действенная обратная связь.

Обычно мы проводим производственные совещания раз в неделю. Учитывая нелюбовь SR-инженеров к бессмысленным обсуждениям, это оптимальная регулярность: можно накопить достаточно материала для того, чтобы совещание было содержательным, и в то же время это не слишком часто, чтобы люди искали повод его пропустить. Такие совещания обычно делятся 30–60 минут. Если совещание будет занимать меньше времени, вы, скорее всего, незаслуженно сократите какую-то тему или же просто увеличите количество бумажных документов о сервисе. Если оно будет занимать больше времени, вы утонете в подробностях. Если же у вас много тем для обсуждения, то лучше разбить команду на подгруппы, чтобы каждая обсуждала определенную часть сервиса.

Как и на любом другом, на производственном совещании нужно выбрать председателя. Многие команды SR-инженеров позволяют побывать в этой роли разным их участникам — так работник получает возможность почувствовать особую заинтересованность в работе сервиса и ответственность за возникновение проблем. Это верно, что не все одинаково способны руководить процессом проведения совещаний, но стоит дать каждому возможность внести свой вклад в обсуждение, для чего можно поступиться некоторым уровнем оптимальности. Помимо этого, у работников появляется отличная возможность получить опыт председательства, что очень полезно при возникновении инцидентов, с которыми часто сталкиваются SR-инженеры.

Повестка дня

Существует множество способов проведения производственного совещания — выбор обычно зависит от типа объектов, за которые отвечает служба SRE, а также от вида ответственности ее членов за конкретный объект. Поэтому не

обязательно проводить все совещания по одной схеме. Однако стандартная повестка дня (см. приложение Е для получения примера) может выглядеть следующим образом.

- *Грядущие изменения на производстве.* Совещания, на которых отслеживаются изменения, хорошо известны в нашей отрасли, и бывает, что вся встреча посвящается обсуждению прекращения изменений. В нашем производственном окружении мы обычно позволяем изменениям проявляться, но приходится отслеживать полезные свойства этих изменений: время начала, продолжительность, ожидаемый эффект и т.д.
- *Показатели.* Чаще всего, обсуждая сервис, мы анализируем основные показатели наших систем; см. главу 4. Даже если системы не давали критических сбоев на прошлой неделе, можно оказаться в ситуации, когда нагрузка плавно (или резко!) увеличивается на протяжении года. Отслеживая изменения показателей задержки обработки данных, показаний загруженности процессора и т.д., можно определить предельные характеристики системы.

Некоторые команды отслеживают использование ресурсов и эффективность, что также может быть подходящим показателем более медленных и, возможно, бессимптомных изменений системы.

- *Сбои.* Эта тема затрагивает проблемы, которые требуют написания незаменимых для обучения постмортемов. Здесь также можно обсудить примерный объем таких отчетов. Качественный анализ постмортемов, как говорится в главе 15, всегда будет полезен.

- *События, связанные с экстренными уведомлениями.* Бывают уведомления, сигнализирующие о проблемах, которые могут потребовать написания постмортема, но зачастую такой отчет не требуется. В то время как на этапе «Сбои» рассматриваются лишь особо крупные сбои, на текущем этапе вы можете получить более подробную информацию: список уведомлений, кто был вызван на работу, что произошло и т.д. На этом этапе нужно задать себе два вопроса: следует ли давать детальную информацию о происшествии и нужно ли вообще отправлять экстренное уведомление? Если ответ на последний вопрос отрицателен, удалите эти вызовы.
- *События, не связанные с вызовами.* Этот этап разбит на три части.
 - *Проблема, для которой должно быть отправлено экстренное уведомление, но оно не было отправлено.* В таких случаях вам, возможно, нужно исправить систему мониторинга, чтобы подобные происшествия инициировали выдачу оповещений. Зачастую такая проблема возникает, когда вы пытаетесь вносить в сервис какие-то исправления, или же хотите установить отправку уведомлений для конкретного показателя, который вы отслеживаете.
 - *Проблема, не требующая экстренного уведомления, но требующая внимания, например незначительное повреждение данных или медленное выполнение на участке, не влияющее на работу пользователей.* Здесь также можно отслеживать оперативную работу.

- *Проблема, не требующая экстренного уведомления и не требующая внимания.* Такие оповещения лучше удалить, поскольку они создают лишний шум, отвлекающий инженеров от того, что действительно требует их внимания.
- *Предыдущие действия.* Подробные обсуждения проблем, перечисленных в пунктах выше, зачастую приводят к тому, что SR-инженерам требуется выполнить определенные действия: исправить первое, понаблюдать за вторым, разработать подсистему для того, чтобы сделать третье. Отслеживайте эти изменения так же, как они отслеживались бы на других совещаниях: распределяйте задания между людьми и проверяйте их выполнение.

Посещение

Посещение совещаний обязательно для всех членов команды SR-инженеров. Это особенно актуально, если сотрудники вашей команды работают в разных странах и/или часовых поясах, поскольку в таком случае совещание становится единственной возможностью поработать в группе.

Основные заинтересованные лица также должны присутствовать на этих совещаниях, как и любые команды разработчиков-партнеров. Некоторые команды SR-инженеров организуют совещания так, что вопросы, важные только для SR-инженеров, рассматриваются в первой половине. Такой прием хорош до тех пор, пока все участники имеют представление о том, что происходит. Время от времени на совещании могут появляться представители других команд SR-инженеров, особенно если необходимо обсудить вопросы, затрагивающие обе команды, но, как правило, на совещаниях собираются члены команды, обслуживающей

рассматриваемый сервис, а также других крупных команд. Если ваши отношения складываются так, что вы не можете пригласить разработчиков-партнеров, лучше это исправить: возможно, первым шагом будет приглашение представителя этой команды или поиск посредника, который станет передавать сообщения. Существует множество причин, по которым команды не ладят, написано множество статей о том, как решить данную проблему. Эта информация актуальна и в отношении команд SR-инженеров, но очень важно несмотря ни на что наладить обратную связь между сотрудниками, иначе невозможно будет добиться их эффективной совместной работы.

Иногда у вас на совещании будет присутствовать слишком много команд или же вам потребуется пригласить «занятых, но критически важных» людей.

Существует несколько приемов, которые позволят вам справиться с этой ситуацией.

- На совещания по менее активным сервисам можно отправлять одного человека из команды разработчиков. Или же выделить конкретное время для выступления представителя этой команды.
- Если команда разработчиков продукта достаточно велика, выделите небольшую группу ее представителей.
- Занятые, но критически важные участники могут передавать свои отзывы и комментарии и/или удаленно управлять процессом, используя прием предварительного составления повестки дня (описан далее).

При планировании стратегии проведения собраний в первую очередь нужно руководствоваться здравым смыслом, с

уклоном в сторону рассматриваемого сервиса. Один уникальный способ, позволяющий сделать совещания более увлекательными и эффективными, заключается в том, чтобы использовать инструменты взаимодействия в реальном времени, предоставляемые Google Docs. Многие команды SRE-инженеров совместно пользуются такими документами, и любой инженер может получить к ним доступ. Такой документ позволяет:

- предварительно составить повестку дня «снизу вверх»: каждый может добавить свои идеи, комментарии и информацию;
- готовить повестку дня параллельно и заранее, что очень эффективно.

Продукт дает возможность полноценно использовать все преимущества одновременного взаимодействия нескольких людей. Какое удовольствие наблюдать за тем, как председатель вносит предложение, затем кто-то другой дает ссылку на исходный материал в скобках, а потом кто-то третий исправляет орфографию и грамматику! Благодаря такому взаимодействию задачи решаются быстрее и большая группа людей может почувствовать, как они внесли свой вклад в общее командное дело.

Взаимодействия внутри службы SRE

Разумеется, компания Google является многонациональной организацией. Из-за специфики нашей работы, связанной с реагированием на неотложные ситуации и дежурствами, у нас есть весомые с точки зрения бизнеса причины быть

распределенной организацией, разнесенной как минимум на несколько часовых поясов. В результате этого наше определение понятия «команда» довольно гибкое в сравнении, например, со стандартной командой разработчиков продукта. У нас есть локальные команды, команды на местах установки, межконтинентальные команды, виртуальные команды разных размеров, а также множество промежуточных вариантов. Это создает хаотический набор ответственности, навыков и возможностей. Такое распределение характерно для любой относительно крупной компании (но наиболее ярко выражено в технических корпорациях). Учитывая, что большинство локальных взаимодействий происходят беспрепятственно, интересными вариантами являются межкомандное и межплощадочное взаимодействия, а также взаимодействия виртуальной команды.

Подобная схема распределения также указывает на способ организации команд SR-инженеров. Поскольку наша *основная цель* заключается в принесении пользы с помощью технического мастерства, а техническое мастерство, как правило, приходит с опытом, мы пытаемся найти способ научиться мастерски работать со связанными наборами систем или инфраструктур для того, чтобы снизить когнитивную нагрузку. Специализация — один из способов достижения этой цели; например, когда команда *X* работает только над продуктом *Y*. Специализация хороша тем, что позволяет повысить ваше техническое мастерство, и плоха потому, что приводит к разрозненности и не дает увидеть полный расклад. Мы стараемся прописывать в командном уставе, что команда будет — и, что еще важно, не будет — поддерживать, но не всегда добиваемся успеха.

Структура команды

У наших инженеров имеется широкий набор навыков: от разработки систем до разработки ПО. Можно с уверенностью сказать, что вероятность успешного взаимодействия — как и все другое — повышается, если ваша команда более разнородна. Есть множество свидетельств тому, что разнородные команды — это лучшие команды [Nelson, 2014].

Формально команды SR-инженеров имеют в своем составе технического лидера (tech lead, TL), менеджера (manager, SRM) и проектного менеджера (project manager, также известного как PM, TPM, PgM). Одни люди выполняют свою работу лучше, если у каждого из этих специалистов ответственность четко определена: основное преимущество такого подхода заключается в том, что они могут принимать решения быстро и безопасно. Другие люди лучше работают в гибкой среде, где ответственность меняется в зависимости от новых задач и решений. В общем, чем более гибкой является команда, тем более она развита с точки зрения возможностей отдельных людей и тем больше она готова адаптироваться к новым ситуациям. Это достигается за счет необходимости чаще общаться, поскольку неизвестно заранее, что человек знает, а чего не знает.

Независимо от того, как хорошо определены эти роли, на базовом уровне технический лидер отвечает за технические решения команды и может управлять ею несколькими способами — от подробного комментирования чужого кода до проведения ежеквартальных презентаций, до достижения консенсуса в команде. В компании Google технические лидеры (TL) могут выполнять большую часть обязанностей менеджера, поскольку наши менеджеры имеют технические знания. Но при этом у менеджера есть две особые обязанности, которых нет у TL: управление производительностью и выполнение всей той работы, которой не занимается кто-то другой.

Опытные TL, SRM и TPM владеют полным набором навыков и легко могут заняться организацией проекта, комментируя его план или занимаясь при необходимости написанием кода.

Приемы эффективной работы

Существует несколько способов эффективно работать в службе SRE.

Как правило, одиночные проекты не становятся успешными, если только человек, работающий с ними, не является особенно одаренным. Для того чтобы достичь чего-то значительного, вам понадобится несколько человек. Поэтому вам также нужны хорошие навыки взаимодействия. Опять же на данную тему было написано много книг и статей, и большая часть этой литературы также применима к SRE.

Для того чтобы SR-инженеры хорошо работали, вам нужно поддерживать с ними связь, когда вы находитесь за пределами вашей локальной команды. Для взаимодействий за пределами здания — по сути, для работы в нескольких часовых поясах — вам потребуется либо четко излагать свои мысли на бумаге, либо часто путешествовать, чтобы общаться с коллегами вживую. Это в конечном счете необходимо для установления отношений на должном уровне.

Пример сотрудничества SR-инженеров: Viceroy

Одним из примеров успешного сотрудничества SR-инженеров является проект под названием Viceroy. Это фреймворк и сервис для мониторинга и работы с информационными панелями. Текущая организационная структура службы SRE может привести к тому, что команды будут создавать несколько лишь слегка отличающихся копий одного и того же

продукта. По разным причинам фреймворки для мониторинга и работы с информационными панелями — это особенно подходящие варианты для такого дублирования¹⁷⁵.

Мотивы беспорядочного появления множества заброшенных фреймворков мониторинга были довольно просты: каждую команду вознаграждали за разработку собственного решения, работать в одиночку было трудно, и инфраструктура, которой должна была пользоваться вся служба SRE, была больше похожа на набор инструментов («тулкит»), а не на продукт. Эти условия вдохновляли отдельных инженеров использовать тулкит для создания еще одного решения-однодневки вместо того, чтобы исправлять проблему для максимально большого количества людей (на что потребовалось бы гораздо больше времени).

Пришествие Viceroy

Viceroy был другим. Его разработка началась в 2012 году, когда несколько команд планировали перейти на использование Monarch, новой системы мониторинга в нашей компании. SR-инженеры очень консервативно относятся к системам наблюдения, поэтому для того, чтобы получить поддержку команд SR-инженеров, Monarch потребовалось больше времени, чем для команд не-SR-инженеров. Но никто не мог спорить с тем, что нашу предыдущую систему наблюдения, Borgmon (см. главу 10), улучшать было некуда. Например, наши консоли были громоздкими, поскольку использовали собственную систему шаблонов HTML, которая была нестандартной и которую было сложно протестировать. В то время система Monarch была достаточно зрелой для того, чтобы стать заменой устаревшей системы, и поэтому ею стали

пользоваться все больше инженеров компании Google, но оказалось, что у нас все еще возникают проблемы с консолями.

Те из нас, кто пытался использовать систему Monarch для своих сервисов, быстро обнаружили, что ей не хватало поддержки консоли по двум основным причинам.

- Консоль можно легко настроить для небольшого сервиса, но это решение плохо масштабируется для сервисов со сложными системами.
- Не поддерживались устаревшие системы мониторинга, что очень усложнило переход на Monarch.

Поскольку в то время не существовало хорошей альтернативы развертыванию Monarch, были запущены несколько проектов для конкретных команд. Кроме того, тогда почти не было скоординированных решений по разработке или даже по межгрупповому отслеживанию (но теперь такой проблемы нет), и мы продолжали повторно выполнять одну и ту же работу. Несколько команд, занимавшихся системами Spanner, Ads Frontend и множеством других сервисов, в какой-то момент объединили свои усилия (одним из наиболее известных примеров этого объединения стала Consoles++). Но в итоге здравый смысл восторжествовал, и однажды инженеры всех этих команд проснулись и посмотрели со стороны на все те усилия, которые приходилось прикладывать другим людям. Они решили поступить разумно и объединились для создания общего решения для всех команд SR-инженеров. Так в середине 2012 года родился проект Viceroy.

К началу 2013 года Viceroy заинтересовались команды, которым только предстояло уйти от устаревшей системы и которые уже были готовы попробовать нечто новое. Очевидно,

что команды с более крупными проектами для мониторинга не хотели переходить на новую систему: им было сложно обосновать обмен своих неприхотливых в обслуживании инструментов, которые, по сути, хорошо работали, на что-то относительно новое и непроверенное, что потребовало бы множества усилий для запуска в работу. Разнообразие требований было еще одним фактором, влиявшим на нежелание этих команд, даже несмотря на то, что все консольные системы наблюдения соответствовали двум основным требованиям, а именно:

- поддерживали сложные курируемые информационные панели;
- поддерживали Monarch и устаревшую систему мониторинга.

Каждый проект также имел свои технические требования, которые зависели от предпочтений автора и его опыта. Например:

- наличие нескольких источников данных за пределами основной системы мониторинга;
- определение консолей с помощью конфигурации, а не макета HTML;
- полноценная поддержка JavaScript с AJAX вместо просто JavaScript;
- использование статического контента, чтобы консоли могли кэшироваться в браузере.

Несмотря на то что одни из этих требований были более простыми, чем другие, все они усложнили объединение усилий. Хотя команда разработчиков Consoles++ была заинтересована в том, чтобы сравнить свой проект с Viceroy, их первичный анализ в первой половине 2013 года показал, что между двумя системами были фундаментальные различия, которые серьезно мешали интеграции. Главная сложность состояла в том, что фреймворк Viceroy по умолчанию почти не использовал JavaScript, а Consoles++ была почти полностью написана на JavaScript. Но в то же время обе системы имели некоторое сходство.

- Использовали одинаковый синтаксис для отрисовки шаблонов HTML.
- Имели общие долгосрочные цели, которые не начинала реализовывать ни одна команда. Например, обе системы хотели кэшировать данные, полученные в ходе наблюдения, и поддерживать офлайн-конвейер для периодического создания данных, которые могут быть использованы в консоли, но для того, чтобы выдавать данные по запросу, требовалось выполнить слишком много вычислений.

Все кончилось тем, что мы приостановили обсуждение создания унифицированной консоли. Однако к концу 2013 года Consoles++ и Viceroy значительно изменились. Технические различия уменьшились, поскольку Viceroy начал использовать JavaScript для отрисовки графов мониторинга. Две команды встретились и поняли, что интеграцию провести стало гораздо легче, поскольку она свелась к выдаче данных Consoles++ с помощью сервера Viceroy. Первый прототип интегрированной системы был завершен в начале 2014 года и доказал, что системы могут работать вместе. Обе команды к тому моменту

научились комфортно сотрудничать, и, поскольку Viceroy уже представлял собой сформировавшийся бренд в области решений по наблюдению, объединенный проект сохранил имя Viceroy.

Разработка всей функциональности заняла еще несколько кварталов, но к концу 2014 года объединенная система была готова.

Объединение усилий принесло огромную пользу.

- Viceroy получил доступ к источникам данных, и клиенты, применяющие JavaScript, смогли ими воспользоваться.
- Компиляция кода JavaScript была переработана для поддержки отдельных модулей, которые можно было бы включать выборочно. Это было важно для того, чтобы команды могли масштабировать систему, используя свой код JavaScript.
- Система Consoles++ выиграла за счет того, что во фреймворк Viceroy было внесено множество улучшений, включая использование кэша и фонового конвейера обработки данных.
- В итоге скорость разработки *одного* решения была выше суммарной скорости создания дублирующихся проектов.

В конечном счете совместный обзор перспектив оказался ключевым фактором при объединении проектов. Обе команды увидели плюсы в расширении группы разработчиков и получили пользу от общего вклада в проект. Импульс имел такую силу, что к концу 2014 года Viceroy был официально объявлен общим решением по мониторингу для всех SR-инженеров. Что характерно для компании Google, такое

объявление не значило, что команды обязаны использовать именно Viceroy: это рекомендовалось делать, вместо того чтобы создавать еще одну консоль для мониторинга.

Сложности

Несмотря на итоговый успех, создание Viceroy прошло не без трудностей, которые в основном были связаны с тем, что проект размещался на разных площадках.

В первое время после создания расширенной команды Viceroy удаленным сотрудникам было трудно координироваться друг с другом. При общении малознакомых людей возможно недопонимание как в письмах, так и в устных обсуждениях, из-за того что стили общения беседующих могут значительно различаться.

Несмотря на то что основная команда Viceroy в Маунтин-Вью¹⁷⁶ оставалась относительно устойчивой, состав расширенной команды периодически изменялся. Участники имели и другие обязанности, которые со временем менялись, и поэтому многие из них могли уделить проекту лишь 1–3 месяца. Таким образом, группы разработчиков, которые были крупнее основной команды Viceroy, характеризовались большим объемом текучки.

Добавление в проект новых людей требовало обучения каждого из них общим принципам организации системы, что занимало некоторое время. С другой стороны, после того как SR-инженеры вносили вклад в основную функциональность Viceroy и возвращались в свою команду, они становились экспертами в конкретной области этой системы. Это неожиданное рассредоточение локальных специалистов помогло повысить используемость сервиса и упростить его освоение.

По мере того как люди присоединялись к команде и покидали ее, мы обнаружили, что такие случайные вклады хотя и полезны, но дорого обходятся. Основные издержки заключалась в распределении ответственности: когда функциональность была добавлена и человек покидал команду, через какое-то время она переставала поддерживаться и, как правило, отбрасывалась.

Помимо этого, границы проекта Viceroy с течением времени разрастались. Перед ним стояли амбициозные цели при запуске, но изначальная *область применения* была ограничена. Однако мы упорно работали, чтобы вовремя предоставить основную функциональность, несмотря на масштабирование.

Наконец, команда разработчиков Viceroy обнаружила, что очень сложно работать с компонентом, в создании которого участвовали инженеры с распределенных площадок. Даже при наличии сильной воли люди стремятся пойти по пути наименьшего сопротивления и предпочитают обсуждать проблемы и принимать решения локально, не задействуя удаленных владельцев, что может привести к конфликтам.

Рекомендации

Вам следует разрабатывать проекты на нескольких площадках только по необходимости. Недостатки такой работы заключаются в высокой задержке действий и необходимости больше общаться; достоинство в том, что, если вы правильно поняли технологию, ваша производительность труда значительно увеличится. Проект, разрабатываемый на одной площадке, также может застопориться из-за того, что никто за ее пределами не будет знать, чем вы занимаетесь. Таким образом, у каждого подхода есть своя цена.

Мотивированные сотрудники всегда ценятся на вес золота, но не все вклады одинаково полезны. Убедитесь, что участники проекта на самом деле увлечены своим делом и что они присоединились к проекту не только ради личной самореализации (стремясь получить значок на свою жилетку, добавить свое имя к списку создателей модного проекта, желая написать код для нового восхитительного проекта без необходимости его дальнейшей поддержки). Участники, ставящие перед собой определенную цель, как правило, лучше мотивированы и будут качественнее делать свою работу.

Проекты имеют свойство разрастаться, и у вас не всегда будет достаточное количество людей в локальной команде, которые смогут участвовать в разработке. Поэтому тщательно продумайте структуру проекта. Очень важно выбрать лидеров, у которых будет стратегическое видение проекта и которые будут гарантировать, что вся работа соответствует этому видению и корректно приоритизируется. Кроме того, нужно выработать способ принятия решений и оптимизировать его так, чтобы принимать больше решений локально, если, конечно, вам удалось построить в команде доверительные отношения.

Стандартная стратегия «разделяй и властвуй» может быть применима к проектам, которые создаются на нескольких площадках. Разбив проект на максимально возможное (оптимальное) количество компонентов, вы снизите затраты на взаимодействие. При этом убедитесь, что каждый компонент можно присвоить небольшой группе, предпочтительно работающей на одной площадке. Разделите эти компоненты среди подкоманд проекта и установите четкие дедлайны. (Попытайтесь не дать закону Конвея нарушить естественную структуру ПО.)[177](#)

Ставя перед командой разработчиков задачу, помните, что она должна быть ориентирована на предоставление какой-то функциональности или на решение проблемы. Этот подход гарантирует, что отдельные люди, работающие над компонентом, будут понимать, что от них требуется, и их работа будет завершена только в тот момент, когда компонент будет полностью интегрирован и станет использоваться внутри основного проекта. Очевидно, к совместным проектам применимы обычные инженерные требования: каждый компонент должен быть зафиксирован в документе проекта, а команда должна выполнять его обзоры. Таким образом, каждый член команды будет в курсе всех изменений, а также сможет повлиять на проект и улучшить его. Документирование всех этапов работ — это один из важнейших приемов, позволяющих преодолеть физическое и/или логическое расстояние, используйте его. Стандарты очень важны. Руководство по написанию кода — это хороший старт, но они обычно носят тактический характер, поэтому могут быть лишь начальной точкой создания норм для команды. Каждый раз, когда начинается спор на тему того, что выбрать, тщательно обсудите этот вопрос с командой, но не затрачивая много времени. Затем выберите решение, задокументируйте его и двигайтесь дальше. Если вы не можете прийти к соглашению, вам нужно выбрать арбитра, которого все уважают, и опять же двигаться дальше. Со временем вы соберете коллекцию наработанных методик, которые помогут новым людям быстрее включаться в работу.

В конечном счете заменить живое общение нельзя, однако некоторую его часть можно восполнить электронной перепиской. Если есть возможность, организуйте личную встречу лидеров проекта и остальной части команды. Если время и бюджет позволяют, соберите саммит команды, чтобы

все ее члены смогли пообщаться друг с другом. Саммит также дает отличную возможность прояснить дизайн и цели проекта.

Наконец, используйте тот стиль управления, который больше всего подходит проекту в его текущем состоянии. Даже проекты с амбициозными целями начинали с малого. По мере роста проекта может иметь смысл пересмотреть способ его управления.

Взаимодействие за пределами службы SRE

Как мы предполагаем (и рассмотрим в главе 32), взаимодействие между организацией разработчиков продукта и службой SRE наиболее эффективно, если происходит на ранних стадиях разработки, в идеале еще до того, как будет написана хоть одна строка кода. Благодаря специфике своей работы SR-инженеры могут давать полезные советы по созданию архитектуры и настройке поведения ПО, которые может быть довольно трудно (если не невозможно) усовершенствовать. Наличие такого советника при проектировании новой системы пойдет всем только на пользу. Проще говоря, мы используем процесс «целей и ключевых результатов» (Objectives & Key Results, OKR) [Klau, 2012] для того, чтобы отслеживать новые проекты, вносить рекомендации, оказывать помощь в их реализации и наблюдать за ними на производстве.

Пример: переход с DFP на F1

Крупные проекты по переходу существующих сервисов к новым версиям довольно распространены в нашей компании. Типичные примеры: перенос компонентов сервиса на новую технологию или обновление компонентов так, чтобы они

поддерживали новый формат данных. С недавним введением технологий баз данных, которые могут масштабироваться на глобальном уровне, наподобие Spanner [Corbett, 2012] и F1 [Shute, 2013] компания Google запустила несколько крупномасштабных обновлений проектов, включающих базы данных. Один из примеров: переход основной базы данных сервиса DoubleClick for Publishers (DFP)¹⁷⁸ с MySQL на F1. В частности, несколько авторов этой главы отвечали за часть системы (показанной на рис. 31.1), которая непрерывно извлекала и обрабатывала данные из базы для того, чтобы сгенерировать набор индексированных файлов, а они затем загружались и распространялись по всему миру. Эта система размещалась на нескольких дата-центрах и использовала примерно 1000 процессоров и 8 Тбайт оперативной памяти для индексирования 100 Тбайт данных ежедневно.

Обновление было нестандартным: в дополнение к переходу на новую технологию схема базы данных была значительно изменена и упрощена благодаря способности F1 хранить и индексировать данные буферов протоколов в столбцах таблицы. Нам требовалось перенести систему обработки, чтобы она могла создавать выходные данные, полностью идентичные существующей системе. Это позволило нам оставить систему обработки нетронутой и выполнить незаметное для пользователя обновление. В качестве дополнительного ограничения продукт требовал, чтобы мы завершили переход, не затронув при этом наших пользователей. Для того чтобы этого достигнуть, команда разработчиков продукта и команда SR-инженеров работали сообща, чтобы создать новый сервис индексирования.

Как основные создатели, команды разработчиков продукта обычно больше знакомы с бизнес-логикой программы, а также находятся в более тесном контакте с менеджерами продукта и

знают больше о его «бизнес-потребностях». С другой стороны, команды SR-инженеров обычно имеют больше опыта в работе с инфраструктурными компонентами ПО (например, библиотеками для взаимодействия с распределенными системами хранения или базами данных), поскольку они зачастую повторно используют одни и те же модули для разных сервисов, будучи в курсе многих подводных камней и нюансов. Это позволяет масштабировать ПО и надежно запускать его.

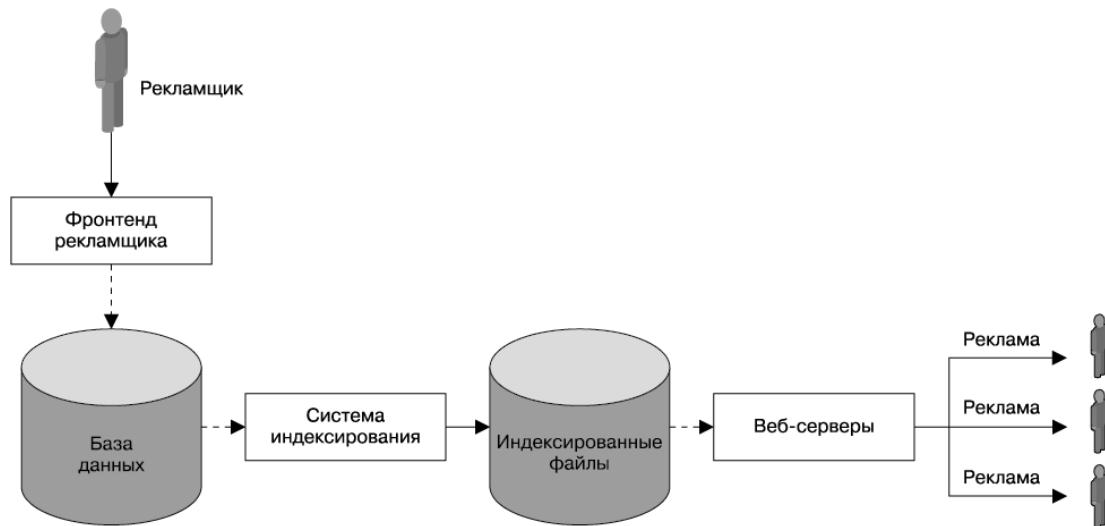


Рис. 31.1. Общая система показа рекламы

С самого начала обновления разработчики продукта и SR-инженеры знали, что им придется работать еще более тесно, проводя еженедельные совещания для синхронизации прогресса на проекте. В этом конкретном случае изменения бизнес-логики частично зависели от изменений инфраструктуры. По этой причине разработка проекта началась с проектирования новой инфраструктуры. SR-инженеры, имевшие широкие познания в области извлечения и обработки данных в крупных масштабах, управляли проектированием этих изменений. Этот процесс включал в себя определение того, как извлекать разные таблицы из F1,

как фильтровать и объединять данные, как извлекать только изменившиеся данные (в противоположность извлечению всей базы данных), как выдержать потерю нескольких машин, не затрагивая сервис, как гарантировать, что использование ресурсов будет расти линейно по мере роста количества извлекаемых данных, как спланировать производительность и другие похожие моменты. Предложенная новая инфраструктура была похожа на другие сервисы, которые уже извлекали данные из F1 и обрабатывали их. Поэтому мы могли убедиться в верности решения и повторно использовать его компоненты, касающиеся мониторинга и инструментария.

Перед тем как начать разрабатывать эту новую инфраструктуру, два SR-инженера создали подробный документ проекта. Далее команды разработчиков продукта и SR-инженеров тщательно проанализировали этот документ, внося небольшие изменения в решение, чтобы справиться с некоторыми пограничными случаями, и в итоге утвердили план проекта. План четко показал, какого рода изменения новая инфраструктура внесет в бизнес-логику. Например, мы разработали новую инфраструктуру для того, чтобы извлекать только изменившиеся данные вместо того, чтобы постоянно извлекать всю базу данных; бизнес-логика должна была принять во внимание этот новый подход. На ранних стадиях мы определили новые интерфейсы между инфраструктурой и бизнес-логикой, и это позволило команде разработчиков независимо работать над изменениями бизнес-логики. Аналогично команда разработчиков информировала SR-инженеров об изменениях бизнес-логики. В местах, где они взаимодействовали (например, изменения бизнес-логики, зависимые от инфраструктуры), такая схема координации позволила нам узнать о возникновении изменений и быстро и корректно обработать их.

На поздних стадиях проекта SR-инженеры начали развертывать новый сервис в тестовой среде, которая напоминала итоговую производственную среду проекта. Этот шаг был необходим для того, чтобы измерить ожидаемое поведение сервиса — в частности, производительность и использование ресурсов — в то время как разработка бизнес-логики все еще продолжалась. Команда разработчиков продукта использовала эту тестовую среду для выполнения проверки сервиса: индекс рекламных роликов, созданный старым сервисом (находящимся в промышленной эксплуатации), должен был точно совпадать с индексом, полученным новым сервисом (работающим в тестовой среде). Как и предполагалось, процесс проверки выявил расхождения между старой и новой версиями (из-за пары пограничных случаев, связанных с форматом данных), с которыми команда разработчиков смогла справиться итеративно. В частности, для каждого рекламного ролика они выполнили отладку, чтобы найти причину различий, и исправили бизнес-логику, которая выдавала плохие результаты. В то же время команда SR-инженеров начала подготовку производственного окружения: выделяла необходимые ресурсы в разных data-центрах, настраивала процессы и правила мониторинга, а также обучала инженеров, которым предстояло дежурить. Команда SR-инженеров также определила процесс получения окончательной версии, включающий в себя проверку правильности — задачу, которой обычно занимается команда разработчиков или релиз-инженеры, но в этом конкретном случае она была выполнена SR-инженерами для ускорения обновления.

Когда сервис был готов, SR-инженеры вместе с командой разработчиков продукта подготовили план его отправки и запустили новый сервис. Запуск прошел успешно, гладко и незаметно для пользователей.

Итоги главы

Учитывая специфику географического распределения команд SR-инженеров, эффективное взаимодействие всегда имело для них высокий приоритет. В этой главе были рассмотрены инструменты и приемы, которые служба SRE использует для поддержания результативных взаимоотношений в команде и со своими партнерами.

Взаимодействие между командами SR-инженеров имеет свои сложности, но потенциально несет и большую выгоду, которая заключается в создании совместных подходов к решению проблем, что позволяет сконцентрироваться на более сложных проблемах.

[175](#) В этом конкретном случае дорога в ад была определенно вымощена кодом на JavaScript.

[176](#) В Маунтин-Вью расположены главные офисы нескольких крупных компаний, в том числе Google. — Примеч. ред.

[177](#) Так и есть, ПО имеет ту же структуру, что и коммуникация в организации, создающей ПО. См. https://en.wikipedia.org/wiki/Conway%27s_law.

[178](#) DoubleClick for Publishers — это инструмент для издателей, предназначенный для управления рекламой, показываемой на их сайтах и в их приложениях.

32. Развитие модели вовлеченности SR-инженеров

Авторы — Акацио Круз и Ашиш Бамбани

Под редакцией Бетси Бейер и Тима Харви

Вовлеченность SR-инженеров: что, как и почему

В большей части этой книги мы рассматривали, что происходит, когда SR-инженер уже находится у руля сервиса. Немногие сервисы начинают свой жизненный цикл, обладая поддержкой SR-инженеров, поэтому должен быть предусмотрен процесс оценки сервиса. Так можно будет убедиться, что сервис достоин поддержки SRE, обсудить способы исправления любых недостатков, которые мешают получить поддержку SRE, и, собственно, добиться этой поддержки. Мы называем данный процесс *освоением*.

Существует как минимум два оптимальных способа воспользоваться опытом разработчиков производственных систем и службы SRE, актуальных как для старых, так и для новых сервисов. В первом случае, как и при разработке ПО (когда чем раньше найдена ошибка, тем дешевле ее исправить), чем раньше вы проконсультируетесь со специалистами SRE, тем лучше будет сервис и тем быстрее вы почувствуете эффект. Когда SR-инженеры вовлекаются в работу на ранних стадиях *проектирования*, время освоения сокращается и сервис оказывается надежнее изначально обычно потому, что нам не приходится тратить время на откат неоптимального проекта или реализации.

Еще один способ, возможно самый лучший, — это упрощение процесса, с помощью которого специально

созданные системы с большим количеством вариаций «прибывают» к дверям SRE. Предоставьте разработчикам продукта платформу проверенной SR-инженерами инфраструктуры, с помощью которой они будут создавать свои системы. Эта платформа принесет двойную пользу, ведь она будет как надежной, так и масштабируемой. Используя распространенные приемы работы с инфраструктурой, разработчики смогут сконцентрироваться на инновациях уровня приложения.

В следующих разделах мы рассмотрим каждую из этих моделей по очереди, начав с классической модели PRR.



Рис. 32.1. Жизненный цикл типичного сервиса

Модель PRR

Наиболее типичный первый шаг вовлечения SR-инженеров — это проверка готовности продукта (Production Readiness Review, PRR). На этом этапе определяются потребности в надежности для сервиса на основе специфических деталей. Во время PRR SR-инженеры стараются применить все, что они изучили для того, чтобы гарантировать надежность работы сервиса на производстве. Выполнение PRR считается обязательным условием для команды SR-инженеров для принятия ответственности за управление производственными характеристиками сервиса.

На рис. 32.1 показан жизненный цикл типичного сервиса. Проверка готовности продукта может быть начата в любой момент жизненного цикла, но этапы, на которых можно привлечь SR-инженера, со временем могут расширяться. В этой главе мы рассмотрим простую модель PRR и поговорим о том, как ее модификация в расширенную модель вовлечения, а также структура фреймворков и платформы SRE позволили SR-инженерам пересмотреть степень их вовлечения и влияния.

Модель вовлечения SR-инженеров

Служба SRE ответственна за важные сервисы, запущенные в эксплуатацию, и периодически повышает их надежность. SR-инженеры отвечают за несколько характеристик сервиса, которые вместе называются промышленным окружением. Это следующие характеристики:

- системная архитектура и зависимости между сервисами;
- инструментарий, показатели и мониторинг;
- реагирование на чрезвычайные ситуации;

- планирование производительности;
- управление изменениями;
- производительность: доступность, задержка и эффективность.

Вовлекая SR-инженера в работу над сервисом, мы стремимся улучшить нашу систему по всем этим параметрам, что позволяет проще управлять производством сервиса.

Альтернативная поддержка

Не все сервисы компании Google получают поддержку SR-инженеров. На это влияют несколько факторов:

- многим сервисам не требуется высокая надежность и доступность, поэтому поддержка может быть предоставлена другими средствами;
- по умолчанию количество команд разработчиков, запрашивающих поддержку SRE, слишком велико и имеющаяся команда SR-инженеров не в состоянии обработать все запросы (см. главу 1).

Когда SR-инженеры не могут предоставить полноценную поддержку, они предлагают другие варианты совершенствования сервиса, например документирование и консультирование.

Документация

Руководства по разработке доступны для внутренних технологий и клиентов широко используемых систем. В Production Guide компании Google задокументированы практические рекомендации для работы с сервисами, определенные из опыта SR-инженеров и разработчиков. Разработчики могут реализовывать решения и рекомендации из этих справочных руководств, чтобы улучшать свои сервисы.

Консультации

Разработчикам также может потребоваться консультация SR-инженеров, для того чтобы обсудить конкретные сервисы или проблемные области. Команда инженеров-координаторов запуска (см. главу 27) тратит большую часть своего времени на консультирование команд разработчиков. Бывает, что и сами SR-инженеры делают это.

Когда запускается новый сервис или внедряется новая функциональность, разработчики обычно консультируются с SR-инженерами о том, как лучше подготовиться к этапу запуска. На консультациях о запуске обычно присутствуют один или два SR-инженера, которые тратят несколько часов на изучение проекта. Далее консультанты встречаются с разработчиками и рассказывают об уязвимых местах, которые требуют внимания, а также обсуждают распространенные решения, которые могут быть внедрены для улучшения сервиса на производстве. Некоторые из этих советов можно почерпнуть из Production Guide, упомянутого ранее.

На подобных консультациях всегда касаются лишь вершины айсберга, поскольку невозможно глубоко вникнуть в суть заданной системы за ограниченный промежуток времени. Некоторым разработчикам таких консультаций оказывается недостаточно.

- Сервисы, которые выросли на несколько порядков после своего запуска и теперь требуют большего времени на изучение, при этом рассмотрения документации и консультирования недостаточно.
- Сервисы, от которых зависит множество других сервисов и которые теперь принимают значительно больше трафика от разных клиентов.

Когда сервисы разрослись до такой степени, что начали сталкиваться со значительными трудностями на производстве, одновременно становясь все более важными для пользователей, вовлечение SR-инженеров попросту необходимо — сервисы должны поддерживаться на производстве по мере их роста.

Проверка готовности продукта: простая модель PRR

Когда команда разработчиков просит SR-инженеров взяться за управление работой сервиса на производстве, SRE оценивают как важность сервиса, так и доступность команд SR-инженеров. Если сервис заслуживает поддержки SRE, а команда SR-инженеров и разработчики приходят к соглашению по поводу объема штата для осуществления этой поддержки, то служба SRE запускает проверку готовности продукта с помощью команды разработчиков.

Цели проверки готовности продукта заключаются в следующем.

- Убедиться, что сервис соответствует принятым стандартам организации производства и операционной готовности и

что владельцы сервиса готовы работать с SR-инженерами и пользоваться их знаниями.

- Улучшить надежность сервиса в промышленной эксплуатации и минимизировать количество и степень тяжести ожидаемых инцидентов. PRR нацеливается на все аспекты производства, о которых заботится служба SRE.

После того как будет сделана работа над ошибками и сервис покажется готовым к поддержке SRE, команда SR-инженеров приступает к своим обязанностям.

Это приводит нас к самому процессу проверки готовности продукта. Существует три разных модели вовлечения, связанные друг с другом (простая модель PRR, модель раннего вовлечения, а также фреймворки и платформа SRE). Мы рассмотрим их по очереди.

Сначала мы разберем простую модель PRR, которая обычно нацелена на уже запущенные сервисы, которые должна перехватить команда SR-инженеров. PRR состоит из нескольких фаз, как и жизненный цикл разработки, хотя эта проверка может выполняться параллельно с ним и независимо от него.

Вовлечение

Руководители службы SRE сначала решают, какая команда SR-инженеров подойдет для работы над сервисом.

Обычно выбирают 1–3 SR-инженеров. Эта небольшая группа начинает работу с совещания с командой разработки. На нем рассматриваются следующие темы.

- Установление SLO/SLA для сервиса.

- Планирование потенциального переформатирования проекта, необходимое для повышения надежности.
- Планирование расписания обучения.

Цель этого этапа заключается в том, чтобы прийти к соглашению о процессе, конечных целях и результатах, что необходимо для вовлечения команды SR-инженеров в работу над сервисом.

Анализ

Анализ — это первый крупный этап работы. Во время этого этапа наблюдатели из SRE изучают сервис и начинают анализировать его недостатки. Они стремятся измерить уровень развития сервиса, а также другие показатели, критически важные для SR-инженеров. Они также исследуют проект сервиса и его реализацию, чтобы убедиться в том, что выбраны подходящие методы производства. Обычно команда SR-инженеров создает и поддерживает чек-лист PRR непосредственно для этапа анализа. Чек-лист уникален для сервиса и, как правило, основан на опыте работы в конкретной области, а также на опыте работы со связанными или похожими системами. В нем также учтены передовые методики из Production Guide. Команда SR-инженеров также может консультироваться с другими командами, имеющими больший опыт работы с определенными компонентами или зависимостями сервиса.

Рассмотрим примеры элементов чек-листа.

- Влияют ли обновления сервиса одновременно на непомерно большую часть системы?

- Соединяется ли сервис с соответствующими обслуживающими экземплярами своих зависимостей? Например, запросы конечного пользователя к сервису не должны зависеть от системы, которая разработана для пакетной обработки данных.
- Запрашивает ли сервис от сети достаточно высокий уровень качества обслуживания (QoS) при взаимодействии с критически важным удаленным сервисом?
- Отправляет ли сервис информацию об ошибках в центральную систему журналирования для анализа? Сообщает ли он обо всех исключительных ситуациях, которые привели к появлению неподходящих ответов или сбоев, видимых конечным пользователям?
- Наблюдаете ли вы за всеми сбоями запросов, заметными пользователям? Настроена ли у вас система оповещения?

Чек-лист также может включать операционные стандарты и практические рекомендации, которыми пользуется определенная команда SR-инженеров. Например, идеально работающая конфигурация сервиса, которая не соответствует «золотым стандартам» команды SR-инженеров, может быть переписана для того, чтобы использовать инструменты SRE для управления масштабирующими конфигурациями. SR-инженеры проверяют информацию о недавних происшествиях и постмортемы, а также смотрят, какие действия были выполнены после инцидентов. В результате этого анализа они определяют потребность сервиса в реагировании на чрезвычайные ситуации и доступность проверенных операционных элементов управления.

Улучшения и рефакторинг

На этапе анализа определяется, как еще нужно доработать сервис. Это будет очередной шаг, который заключается в следующем.

1. Улучшения приоритизируются в зависимости от их важности для надежности сервиса.
2. Приоритеты рассматриваются вместе с командой разработчиков, затем утверждается план выполнения.
3. Команды SR-инженеров и разработчиков продукта выполняют рефакторинг частей сервиса или реализуют дополнительную функциональность, помогая друг другу.

Продолжительность этого этапа и объем затрачиваемых усилий обычно изменяются от сервиса к сервису. Они также зависят от доступности времени инженеров для рефакторинга, от уровня развития и сложности сервиса в начале обзора, а также от множества других факторов.

Обучение

Ответственность за управление «промышленным» сервисом нередко лежит на всей команде SR-инженеров. Для того чтобы гарантировать, что команда готова, наблюдатели из SRE, выполнившие PRR, начинают руководить процессом обучения команды. Этот процесс обычно включает в себя изучение всей документации, необходимой для поддержки сервиса. При участии команды разработчиков SR-инженеры выполняют различные задания и упражнения. Это могут быть:

- обзор проекта;

- погружения в разнообразные потоки запросов системы;
- описание настройки промышленного окружения;
- опыт из первых рук для разных показателей работы системы.

По завершении обучения команда SR-инженеров должна быть готова к управлению сервисом.

Освоение

Фаза обучения откроет возможность освоения сервиса командой SR-инженеров. Этап освоения включает в себя последовательное перераспределение обязанностей и ответственности за различные части сервиса. Команда SR-инженеров продолжает концентрироваться на разных областях производственной среды, упомянутых ранее. Чтобы можно было запустить сервис в эксплуатацию, команда разработчиков должна быть доступна для оказания поддержки SR-инженерам. Это отношение в дальнейшем станет залогом эффективной непрерывной работы между командами.

Непрерывное улучшение

Популярные сервисы постоянно меняются, подстраиваясь под современные потребности и условия, включая в том числе запросы пользователями новой функциональности, развитие зависимостей системы и обновление технологий. Команда SR-инженеров должна поддерживать стандарты надежности сервиса с учетом этих изменений, непрерывно улучшая его. Ответственная команда SR-инженеров тщательно анализирует работу сервиса во время его использования, обзора изменений, реагирования на инциденты и особенно при написании

постмортемов и поиске основных причин. В дальнейшем результаты такого анализа передаются команде разработчиков как указания и рекомендации по внесению изменений в сервис: добавлению новой функциональности, компонентов и зависимостей. Уроки, усвоенные в ходе управления сервисом, также сохраняются в виде практических рекомендаций, которые задокументированы в Production Guide и других источниках.

Вовлекаемся в работу над сервисом Shakespeare

Изначально ответственность за сервис Shakespeare несли его разработчики. Ответственность заключалась в том, что они носили пейджеры, сигнализировавшие об аварийных ситуациях. Однако по мере повышения популярности сервиса и увеличения его прибыльности появилась необходимость в поддержке SRE. Продукт уже был запущен, поэтому SR-инженеры проверили его готовность. Помимо всего прочего, они обнаружили, что информационные панели охватывали не все показатели, определенные в SLO, что требовалось исправить. После того как все проблемы были устраниены, SR-инженеры взяли на себя обязанности по реагированию на аварийные ситуации, однако на дежурства ходят и два разработчика сервиса. Разработчики еженедельно проводят собрания, куда зовут и дежуривших работников, где обсуждаются проблемы, появившиеся на прошлой неделе, и способы преодоления грядущих крупномасштабных сбоев или отключений кластера. Предстоящие планы по развитию сервиса обсуждаются и с SR-инженерами, чтобы убедиться,

что новые запуски пройдут идеально (однако не забывайте про закон Мерфи, который способен нарушить идиллию).

Развиваем простую модель PRR: раннее вовлечение

Итак, мы обсудили проверку готовности продукта в виде простой модели PRR, которую можно использовать для сервисов, уже вышедших на стадию запуска. Существует несколько ограничений и условий, связанных с этой моделью.

- Дополнительное взаимодействие между командами может усложнить некоторые виды деятельности команды разработчиков и повысить нагрузку на наблюдателей из SRE.
- Наблюдатели из SRE должны быть доступны и обязаны правильно распределять свои приоритеты и рабочее время.
- Работа, выполняемая SRE, должна быть заметна и достаточно проанализирована командой разработчиков для того, чтобы гарантировать эффективный обмен знаниями. SR-инженеры обязаны, по сути, работать как часть команды разработчика, а не как внешний элемент.

Однако основное ограничение модели PRR вытекает из того факта, что сервис уже запущен и широко развернут, а вовлечение SR-инженеров начинается на очень поздних этапах жизненного цикла разработки. Если PRR будет происходить на более ранних этапах создания сервиса, возможность SR-инженеров исправить его потенциальные проблемы значительно увеличится. В результате успех вовлечения SR-

инженеров и будущий успех сервиса, скорее всего, тоже возрастут.

Кандидаты для раннего вовлечения

Модель раннего вовлечения позволяет SR-инженерам раньше начать работать с сервисом для того, чтобы получить существенные дополнительные преимущества. Применение такой модели требует определения важности и/или бизнес-ценности сервиса на ранних этапах жизненного цикла разработки, а также определения того, будет ли сервис масштабироваться или усложняться до такой степени, чтобы ему потребовалась поддержка SR-инженера. К таковым относятся сервисы, которые отвечают следующим условиям.

- Сервис реализует заметную новую функциональность и является частью существующей системы, уже управляемой службой SRE.
- Сервис представляет собой значительно измененную или альтернативную версию существующей системы.
- Команда разработки просит совета у SR-инженеров или предлагает им принять ответственность за сервис после его запуска.

Модель раннего вовлечения, по сути, погружает SR-инженеров в процесс разработки. Они концентрируются на тех же показателях, но способы создания сервиса будут отличаться. SR-инженеры участвуют в работе на стадии проектирования и более поздних этапах, в итоге принимая на себя управление сервисом в любой момент во время или после его сборки. Эта

модель основана на активном взаимодействии между командами разработчиков и SR-инженеров.

Преимущества модели раннего вовлечения

Несмотря на то что модель раннего вовлечения связана с некоторыми рисками и сложностями, дополнительное участие SR-инженеров на протяжении всего жизненного цикла продукта дает значительные преимущества по сравнению с вовлечением, выполненным на более поздних этапах жизненного цикла сервиса.

Фаза проектирования

Участие SR-инженеров в проектировании может предотвратить множество проблем или инцидентов, которые могли бы случиться позже на производстве. Хотя решения по проекту могут быть отменены или исправлены на более поздних этапах жизненного цикла, такие изменения будут стоить дороже с точки зрения затраченных усилий и сложности. Лучшие инциденты — те, которые никогда не происходят!

Необходимость время от времени искать компромиссы приводит к созданию неидеального проекта. Участие SR-инженеров в проектировании означает, что им заранее известно об этих компромиссах и они частично ответственны за решение использовать неидеальный вариант. Раннее вовлечение службы SRE позволит минимизировать будущие споры насчет проекта, которые могут произойти, когда сервис поступит в промышленную эксплуатацию.

Сборка и реализация

На этапе сборки принимаются решения по техническим характеристикам. Сюда относятся выбор инструментов и определение показателей, управление аварийными ситуациями и плановой работой, использование ресурсов и установка эффективности. На этой стадии SR-инженеры могут повлиять на реализацию проекта и улучшить ее, порекомендовав определенные библиотеки и компоненты или оказав помочь при сборке некоторых элементов управления для системы. Участие службы SRE на данном этапе помогает упростить использование сервиса в будущем и позволяет SR-инженерам получить опыт работы с ним еще до его запуска.

Запуск

SR-инженеры также могут помочь в реализации популярных методов запуска. К одному из них относится «темный запуск», при котором часть трафика существующих пользователей направляется на новый сервис в дополнение к тому, что она поступает и на уже работающую промышленную версию сервиса. Ответы нового сервиса будут считаться «темными», поскольку они не учитываются и не показываются пользователям. Такие приемы позволяют команде понять, как работает сервис, и решить проблемы, не затрагивая существующих пользователей, тем самым снизив риск появления этих проблем после запуска. Постепенный запуск значительно снижает уровень операционной нагрузки и дает команде разработчиков возможность в спокойном режиме исправлять все ошибки и продолжать работать над будущей функциональностью.

После запуска

Если система стабильна во время запуска, то в дальнейшем команде разработчиков не приходится выбирать между улучшением надежности сервиса и добавлением новой функциональности. На более поздних этапах развития сервиса уже будет понятно, требуется рефакторинг или полная смена проекта.

При полноценном участии команда SR-инженеров может быть готова взять на себя управление новым сервисом гораздо раньше, чем это было бы возможно при простой модели PRR. Чем дольше и глубже SR-инженеры вовлекаются в работу над сервисом, тем выше вероятность того, что команды SR-инженеров и разработчиков наладят эффективные отношения, которые со временем будут только укрепляться.

Отстранение от сервиса

Иногда сервису не требуется полноценное управление со стороны SRE — это можно определить после запуска. Или же SR-инженеры могут вовлекаться в работу над сервисом, но не брать на себя управление им. Это говорит о том, что сервис надежный, почти не требует обслуживания и его поддержку способна выполнять команда разработчиков.

Возможны также ситуации, когда SR-инженер вовлекается в работу над сервисом, но нагрузка не соответствует прогнозируемой. В таких случаях усилия, приложенные SR-инженерами, оказываются частью бизнес-риска, который характерен для новых проектов. Команда SR-инженеров может быть перенаправлена для работы над другим проектом, и полученный ими опыт может быть применен в процессе вовлечения.

Развитие процесса разработки сервисов: фреймворки и платформа SRE

Модель раннего вовлечения стала серьезным шагом в развитии модели вовлечения SR-инженеров за пределы PRR, которая могла быть применена только к уже запущенным сервисам. Однако нам все еще предстояло «масштабировать» вовлечение SR-инженеров на следующий уровень, обеспечив еще большую надежность.

Усвоенные уроки

Со временем на базе модели вовлечения SR-инженеров, описанной ранее, было выведено несколько закономерностей.

- Освоение каждого сервиса требовало 2–3 SR-инженеров и, как правило, продолжалось 2–3 квартала. Период освоения для PRR было относительно большим (несколько кварталов). Уровень требуемых усилий был пропорционален числу наблюдаемых сервисов и был ограничен незначительным количеством SR-инженеров, доступных для выполнения PRR. Эти условия привели к разделению сервисов на определенные группы и четкой их приоритизации.
- Из-за того, что для разных сервисов применялись разные методики, каждый участок функциональности реализовывался по-разному. Для того чтобы соответствовать стандартам PRR, функциональность должна была быть повторно реализована отдельно для каждого сервиса или в лучшем случае один раз для каждого подмножества сервисов, использовавших одинаковый код. Эти повторные реализации были бесполезной тратой времени инженеров. Один из типичных примеров — реализация функциональности похожих фреймворков

журналирования на одном и том же языке из-за того, что у разных сервисов различалась структура кода.

- Обзор распространенных проблем и сбоев сервисов выявил определенные закономерности, но не было способа быстро решить проблему и доработать одновременно все сервисы. Типичный пример — перегрузка сервисов.
- Вклад SR-инженеров в разработку ПО зачастую был лишь локальным — ограничивался конкретным сервисом. Поэтому оказывалось непросто вывести общие решения, которые впоследствии можно использовать повторно. Из-за этого не существовало простого способа применить новые знания и практические навыки, полученные отдельными командами SR-инженеров, для всех сервисов, которые они поддерживали.

Внешние факторы, влияющие на SRE

Внешние факторы традиционно влияют несколькими способами на организацию SRE и ее ресурсы.

Компания Google все больше следует тренду отрасли, заключающемуся в использовании микросервисов¹⁷⁹. В результате количество запросов на поддержку SRE и количество сервисов, претендующих на такую поддержку, увеличивается. Поскольку каждый сервис отличается конкретными издержками на сопровождение, даже простые сервисы требуют участия большого количества людей. Микросервисы также предполагают меньший срок развертывания, что было невозможно при использовании предыдущей модели PRR (период разработки длился несколько месяцев).

Наём опытных, квалифицированных SR-инженеров будет дорого стоить. Несмотря на значительные усилия организаций, занимающихся наймом, SR-инженеров для всех нуждающихся в поддержке сервисов всегда не хватает. Обучение такого специалиста длится дольше, чем обучение разработчика.

Наконец, служба SRE ответственна за обслуживание запросов большого и постоянно растущего количества команд разработчиков, которые в данный момент не пользуются поддержкой SR-инженеров. В связи с этим есть необходимость расширить сферу поддержки SRE за пределы исходной модели сотрудничества.

На пути к структурному решению: фреймворки

Для того чтобы эффективно отреагировать на эти запросы, было принято решение разработать модель, учитывающую следующее.

- *Закодированные лучшие практические приемы.* Возможность зафиксировать в коде то, что хорошо работает в условиях промышленной эксплуатации, чтобы разработчики сервисов могли легко использовать этот код и создавать свои сервисы сразу «готовыми к выходу на производство».
- *Решения, используемые повторно.* Популярные и простые реализации методов, которые можно использовать для сокращения количества проблем с надежностью и масштабируемостью.
- *Общепринятая платформа продукта с унифицированным уровнем управления.* Однаковые наборы интерфейсов для производства, одинаковые наборы операционных

элементов управления и одинаковое журналирование и конфигурация для всех сервисов.

- *Упрощенная автоматизация и гибкая система.* Унифицированный уровень управления, который позволяет выполнять автоматизацию и создавать гибкие системы на новом уровне. Например, SR-инженеры всегда готовы получить релевантную информацию о сбое из одного достоверного источника, вместо того чтобы собирать вручную и анализировать необработанные данные из разных мест (журналы, данные системы мониторинга и т.д.).

На основе этих принципов был создан набор платформ и служебных фреймворков, поддерживаемый SRE, по одному для каждой обслуживаемой нами системы (Java, C++, Go). Сервисы, построенные с использованием этих фреймворков, имеют общую реализацию. Она спроектирована так, чтобы работать внутри платформы, поддерживаемой SR-инженерами, и обслуживаться командами SR-инженеров и разработчиков. Основное изменение, привнесенное этими фреймворками, заключалось в том, что разработчики теперь могли использовать готовые решения, которые были созданы и одобрены SR-инженерами, а не изменять приложения по спецификациям SRE постфактум либо вовлекать множество SR-инженеров для поддержки сервисов, значительно отличающихся от других сервисов нашей компании.

Приложение обычно содержит бизнес-логику, которая, в свою очередь, зависит от множества инфраструктурных компонентов. Опасения SR-инженеров на тему производственной среды касаются связанных с инфраструктурой частей сервиса. Фреймворки сервисов реализуют код инфраструктуры стандартизированным

способом и решают многие производственные задачи. Любая задача инкапсулирована в один или несколько модулей фреймворка, каждый из которых предоставляет целостное решение для проблемной области или зависимости инфраструктуры. Модули фреймворка решают проблемы:

- инструментария и показателей;
- журналирования запросов;
- систем управления, включающих управление трафиком и нагрузкой.

SR-инженеры создают модули фреймворка для того, чтобы реализовать конкретные решения для требуемой производственной области. В результате команды разработчиков могут сконцентрироваться на бизнес-логике, поскольку фреймворк сам по себе заботится о корректном использовании инфраструктуры.

Фреймворк, по сути, является источником программных компонентов и предлагает определенный способ их объединения. Он также может предоставлять функциональность для управления разными компонентами, например следующее.

- Бизнес-логику, организованную в виде структурированных семантических компонентов, которые можно обозначать стандартными терминами.
- Стандартные единицы измерения для инструментария мониторинга.
- Общепринятый формат для журналов отладки запросов.

- Стандартный формат конфигурации для управления сегментацией нагрузки.
- Производительность одного сервера и определение «перегрузки» не противоречат друг другу и могут характеризовать оперативность обратной связи при работе с различными управляющими системами.

Фреймворки повышают согласованность и эффективность работы. Они освобождают разработчиков от необходимости объединять и конфигурировать отдельные потенциально несовместимые компоненты, которые затем должны будут пройти проверку SR-инженеров. Они предлагают единое популярное решение задач промышленно эксплуатируемых сервисов, а это означает, что такие сервисы будут иметь одинаковую реализацию и минимальные различия в конфигурации.

Компания Google использует для разработки приложений несколько полнофункциональных языков программирования, и фреймворки реализованы для каждого из этих языков. Несмотря на то что разные реализации фреймворка (например, C++ против Java) не могут иметь общий код, их цель заключается в том, чтобы предоставлять одинаковые API, поведение, конфигурацию и элементы управления для однотипных задач. Поэтому команды разработчиков могут выбирать языковую платформу, которая лучше всего подходит их запросам и опыту, а SR-инженеры — ожидать привычного поведения на производстве и стандартных инструментов для управления сервисом.

Преимущества новой модели обслуживания и управления

Структурный подход, основанный на фреймворках сервисов, единой производственной платформе и единой системе управления, дает множество преимуществ.

Значительно снижена операционная нагрузка

Производственная платформа, построенная на основе фреймворков с четко прописанными соглашениями, значительно снижает операционную нагрузку по следующим причинам.

- Поддерживает проверки совместимости для кодовой структуры, зависимостей, тестов, руководств по стилю кодирования и т.д. Эта функциональность также гарантирует конфиденциальность данных пользователя, тестирование и совместимость с системой безопасности.
- Предоставляет встроенную функциональность развертывания, мониторинга и автоматизации для всех сервисов.
- Позволяет проще управлять большим количеством сервисов, особенно микросервисов, число которых постоянно увеличивается.
- Дает возможность быстрее развертывать сервисы: путь от идеи до полностью развернутой качественной системы может быть пройден всего за несколько дней!

Универсальная поддержка по умолчанию

Постоянный рост количества сервисов в нашей компании означает, что бо́льшая их часть либо не требует вовлечения SR-инженеров, либо просто ими не поддерживается. Независимо

от этого сервисы, которые не получают полной поддержки службы SRE, могут быть построены так, чтобы использовать производственную функциональность, которая разработана SR-инженерами. Распространение поддерживаемых SR-инженерами производственных стандартов и инструментов для всех команд улучшает общее качество обслуживания в компании Google. Помимо этого, все сервисы, которые реализованы на базе фреймворков, автоматически обновляются, когда вносятся изменения в модули соответствующих фреймворков.

Быстрое вовлечение с низкими издержками

Подход с использованием фреймворков приводит к более быстрому выполнению PRR, поскольку мы можем рассчитывать:

- настроенную функциональность сервисов как часть реализации фреймворка;
- быстрейшее освоение сервисов (обычно выполняется одним SR-инженером примерно за квартал);
- снижение когнитивной нагрузки для команд SR-инженеров, управляющих сервисами на базе фреймворка.

В результате командам SR-инженеров не приходится тратить много времени на оценку сервиса для его освоения и они могут поддерживать высокую планку качества обслуживания.

Новая модель вовлечения, основанная на общей ответственности

Оригинальная модель вовлечения SR-инженеров предлагала всего два варианта: либо полную поддержку, либо полное ее отсутствие¹⁸⁰.

Наличие производственной платформы, предлагающей распространенную структуру сервиса, соглашения и программную инфраструктуру, сделало возможным тот факт, что команда SR-инженеров отвечает за инфраструктуру «платформы», а команды разработчиков предоставляют поддержку на дежурстве при возникновении проблем с функциональностью — багов в коде приложения. С учетом этой модели SR-инженеры предполагают, что на них лежит ответственность за разработку и обслуживание крупных фрагментов инфраструктуры сервисов, а также систем управления распределением нагрузки, перегрузкой, автоматизацией, трафиком, журналированием и мониторингом.

Эта концепция не соответствует изначально принятому способу управления сервисом. Она влечет за собой новую модель взаимоотношений между командами SR-инженеров и разработчиков, а также новую технологию найма персонала для управления сервисами, которые должны поддерживаться SR-инженерами¹⁸¹.

Итоги главы

Надежность сервиса может быть повышена путем вовлечения SR-инженеров, задача которых состоит в систематическом анализе и улучшении его производственных показателей. Изначальный системный подход SR-инженеров компании Google — простой анализ готовности к запуску — значительно усовершенствовался, но его все еще можно было применять

только для тех сервисов, которые уже перешли на стадию запуска.

Со временем в Google SRE расширяли и улучшали эту модель. Модель раннего вовлечения подразумевала присоединение SR-инженеров на более ранних этапах жизненного цикла, чтобы они выполняли «проектирование для надежности».

Спрос на SR-инженеров продолжал расти, и необходимость более масштабируемой модели вовлечения стала очевидной. В итоге начали активно разрабатывать фреймворки для производственных сервисов. Их использование значительно упростило процесс сборки сервисов, готовых к передаче в промышленную эксплуатацию.

Все три описанные модели вовлечения все еще используются в компании Google. Однако в Google при создании рабочих производственных систем все активнее применяются фреймворки; эта же практика помогает принципиально усилить проникновение SRE на всех уровнях и тем самым снизить издержки на управление сервисами и повысить (минимальную) планку качества обслуживания в рамках всей организации.

179 См. страницу «Википедии», посвященную микросервисам:
<http://en.wikipedia.org/wiki/Microservices>.

180 Не считая того, что время от времени команды SR-инженеры консультировали разработчиков неосвоенных сервисов.

181 Новая модель управления сервисом меняет схему найма персонала с двух сторон: во-первых, поскольку почти все сервисы созданы с применением одинаковых технологий, для их поддержки требуется меньше SR-инженеров; во-вторых, она позволяет создавать производственные платформы с разделением зон ответственности между поддержкой самой платформы (выполняется SR-инженерами) и поддержкой логики, уникальной для сервиса, которая остается за командой разработчиков. Команды наполняются с учетом необходимости обслуживать платформу, а не из-за увеличения количества сервисов и могут работать с несколькими продуктами.

Часть V. Выводы

После того как мы всесторонне рассмотрели работу службы SRE компании Google, имеет смысл взглянуть на главу 33 «Полезные уроки из других отраслей», чтобы сравнить методы работы службы SRE с методиками, принятыми в других сферах, где надежность также критически важна.

Кроме того, вице-президент службы SRE в компании Google, Бенджамин Лутч, в главе 34 «Заключение» поведает о развитии службы за время его карьеры, рассмотрев SRE с точки зрения авиационной отрасли.

33. Полезные уроки из других отраслей

Автор — Дженифер Петофф

Под редакцией Бетси Байер

Глубокое погружение в культуру и практики Google SRE естественным образом приводит к вопросу: как в других отраслях налажено управление надежностью бизнеса? Во время составления этой книги у нас была возможность поговорить с несколькими инженерами, работавшими в компании Google, об их предыдущих местах работы, где тоже требовалась высокая надежность, и получить ответы на следующие вопросы.

- Важны ли принципы, применяемые службой SRE, за пределами компании Google? Используются ли в остальных сферах какие-то другие методы обеспечения надежности?
- Если в других отраслях также соблюдают принципы SRE, как эти принципы воплощаются?
- Какие факторы влияют на сходства и различия в реализации?
- Что может почерпнуть из этих сравнений компания Google и наша отрасль?

В этой книге рассматривается множество принципов, фундаментальных для SRE. Для того чтобы упростить сравнение передовых методик, принятых в других сферах, мы выделили четыре основные категории.

- Готовность к катастрофам и их тестирование.

- Культура написания постмортема.
- Автоматизация и снижение служебной нагрузки.
- Структурированное и рациональное принятие решений.

Мы проинтервьюировали нескольких опытных специалистов, каждый из которых является профессионалом в определенной сфере. Мы обсудили основные понятия SRE, поговорили о том, как они реализованы в компании Google, и для сравнения рассмотрели примеры того, как базовые для SRE принципы проявляются в других отраслях. Завершили главу размышлениями на тему обнаруженных нами шаблонов и антишаблонов.

Познакомьтесь с нашими участниками

Питер Даль — главный инженер компании Google. Ранее работал военным подрядчиком некоторых систем с высокой надежностью, включая GPS и системы инерциальной навигации для многих воздушных и колесных транспортных средств. Ошибки в таких системах могут повлечь отказ транспортного средства или привести к его серьезной поломке, а также чреваты финансовыми последствиями.

Майк Доэрти — SR-инженер компании Google. На протяжении 10 лет работал в Канаде спасателем и инструктором. Надежность в этой профессии крайне важна, поскольку каждый день на кону стоят человеческие жизни.

Эрик Гросс в данный момент работает программным инженером в Google. До того как присоединиться к компании, он 7 лет разрабатывал алгоритмы и код для лазеров и систем, предназначенных для проведения рефракционной хирургии

(например, LASIK). В этой отрасли также многое стоит на кону и требуется высочайшая надежность.

Гас Хартман и Кевин Греер имеют опыт в отрасли телекоммуникаций — они отвечали за обслуживание системы ответов на неотложные ситуации Е911¹⁸². В данный момент Кевин работает программным инженером в команде Google Chrome, а Гас — системным инженером в Corporate Engineering. Ожидания пользователей от отрасли телекоммуникаций требуют высокой надежности. Последствия ошибок обслуживания ранжируются от доставления пользователям неудобства до летальных исходов, если сервис Е911 будет отключен.

Рон Хайби — технический программный менеджер в Google SRE. Рон имеет опыт разработки сотовых телефонов и медицинских устройств. Он также работал в автомобильной промышленности. В некоторых случаях он отвечал за создание интерфейсов (например, для устройства, которое позволяло отправлять данные ЭКГ по цифровой беспроводной телефонной сети). В этих отраслях очень важно обеспечить высокую надежность сервиса — ошибки могут повлиять как на работу медицинского оборудования, так и непосредственно на жизнь и здоровье людей (например, люди не получат медицинской помощи, если результаты ЭКГ невозможно передать в больницу).

Эдриан Хилтон — инженер-координатор запуска в компании Google. Ранее он работал в компании, выпускающей военные воздушные суда для Великобритании и Соединенных Штатов Америки, в том числе проектировал резервные системы управления самолетов и железнодорожные системы сигнализации для Великобритании. Надежность в этих областях критически важна, поскольку сбои могут привести к многомиллионным финансовым потерям, связанным с

оборудованием, и в самых плачевых ситуациях — к травмам и даже летальным исходам.

Эдди Кеннерли — менеджер проектов в команде Global Customer Experience в компании Google и инженер-механик по образованию. Эдди 6 лет проработал инженером-технологом в Six Sigma Black Belt, где создавались искусственные алмазы. В этой промышленной отрасли безопасности уделяют особое внимание, поскольку крайне высокие температуры и давление на производстве несут большую опасность для работников.

Джон Ли в данный момент работает SR-инженером. Ранее был системным администратором и разработчиком ПО в частной финансовой компании. Как вы понимаете, проблемы с надежностью в финансовом секторе крайне нежелательны.

Дэн Шеридан — SR-инженер в компании Google. До этого он работал консультантом по безопасности в отрасли промышленности, связанной с использованием ядерной энергии в мирных целях, в Великобритании. Надежность в этой отрасли очень важна, поскольку любой инцидент способен привести к серьезным последствиям: из-за сбоя можно потерять миллионы долларов прибыли в день. Атомная инфраструктура разработана с большим количеством предохранителей, которые останавливают работу системы, не доводя до происшествия.

Джефф Стивенсон в данный момент является менеджером по работе с оборудованием в компании Google. Ранее он работал инженером-ядерщиком в ВМС США на подводной лодке. Требования к надежности в атомном флоте чрезвычайно высоки — результатом ошибки может стать как повреждение оборудования, так и утечка радиации вплоть до летальных исходов.

Мэттью Тойя — SR-инженер, работающий с системами хранения. До Google он разрабатывал ПО и обслуживал

программные системы контроля воздушного трафика. Сбои, происходящие в этой отрасли, могут приводить как к неудобствам для пассажиров (например, отложенные рейсы, изменение курса самолета), так и к летальным исходам в случае аварии. Глубокая защита — это основная стратегия, позволяющая избежать катастрофических ситуаций.

Теперь, когда вы познакомились с нашими экспертами и поняли, почему в отраслях, где они работали ранее, так важна безопасность, рассмотрим четыре основные идеи, связанные с надежностью систем.

Готовность к катастрофам и их тестирование

«Надежда — плохая стратегия». Этот лозунг команды SR-инженеров в компании Google подытоживает все, что мы имеем в виду касательно готовности и тестирования катастроф. Культура SRE заключается в том, чтобы всегда быть бдительными и постоянно задаваться вопросами: «Что может пойти не так?», «Какие действия мы можем предпринять для того, чтобы справиться с проблемами до того, как они приведут к сбою или потере данных?». На наших тренингах Disaster and Recovery Testing (DiRT) мы ежегодно поднимаем эти вопросы [Krishan, 2012]. В упражнениях DiRT SR-инженеры доводят производственные системы до предела их возможностей и провоцируют реальные сбои для того, чтобы:

- гарантировать, что системы будут реагировать именно так, как ожидается;
- определить неожиданные уязвимые места;
- найти способы сделать систему более устойчивой, чтобы предотвратить неконтролируемые сбои.

Некоторые стратегии тестирования готовности к катастрофам сформировались в результате наших обсуждений и включают в себя следующее:

- постоянную организационную концентрацию на безопасности;
- внимание к деталям;
- мгновенную производительность;
- симуляции и прогоны;
- обучение и сертификацию;
- концентрацию на сборе детальных требований и проектировании;
- глубокую защиту.

Постоянная концентрация на безопасности

Этот принцип особенно важен в промышленном инженерном контексте. По словам Эдди Кеннеди, работавшего на производственном участке с повышенной опасностью, где «каждое совещание по управлению начиналось с вопросов безопасности», обрабатывающая промышленность готовит себя к неожиданностям, прописывая строго определенные правила, которым четко следуют на всех уровнях организации. Для всех сотрудников критически важно серьезно относиться к вопросу безопасности, и работники знают, что могут высказаться, если и когда им кажется, что что-то упущено. В сфере ядерной энергетики, создания военных самолетов и железнодорожных сигналов стандарты безопасности для ПО прописаны четко (например, в UK Defence Standard 00-56, IEC

61508, IEC513, US DO-178B/C и DO-254), а уровни надежности для таких систем однозначно определены (например, уровень полноты безопасности 1–4 — Safety Integrity Level, SIL)¹⁸³. Это сделано для установления приемлемых подходов к созданию продукта.

Внимание к деталям

Вспоминая свою работу в US Navy, Джек Стивенсон рассказывает, что даже недостаточно усердное выполнение мелких заданий (например, замена смазочного масла) могло привести к серьезному сбою на подлодке. Очень маленький просчет или ошибка могли иметь значительные последствия. Системы подлодки сильно взаимосвязаны, поэтому инцидент в одной области может затронуть несколько связанных компонентов. На атомных субмаринах регулярно проводят профилактическое ТО, чтобы гарантировать, что небольшие проблемы не приведут к катастрофе.

Мгновенная производительность

Работа систем в отрасли телекоммуникаций может быть очень непредсказуемой. Их производительность может быть нарушена непредвиденными ситуациями вроде природных катастроф, а также крупными предсказуемыми событиями вроде Олимпийских игр. По словам Гаса Хартмана, отрасль справляется с такими инцидентами, разворачивая мгновенную производительность вида SOW (switch on wheels — «переключатель на колесах») — мобильный офис телекоммуникаций. Эта дополнительная производительность может быть предоставлена в неотложных ситуациях или в том случае, если ожидается известное событие, которое, скорее всего, перегрузит систему.

Проблемы с производительностью могут неожиданно проявиться и в обычное время. Например, в 2005 году в общий доступ попал личный номер телефона одного известного человека и в один момент тысячи фанатов одновременно попытались ему дозвониться. Система телекоммуникаций была сильно перегружена — отмечалось нечто похожее на DDoS-атаку или массивную ошибку маршрутизации.

Симуляции и прогоны

Тесты восстановления от катастроф в компании Google имеют много общего с симуляциями и прогонами, которые являются основными средствами для предотвращения таких ситуаций во многих рассмотренных нами сферах. Изучив потенциальные последствия сбоя системы, можно определить, что лучше использовать — симуляцию или прогоны. Например, Мэттью Тойя указывает, что в авиационной промышленности нельзя выполнить тесты «на производстве», не рискуя оборудованием и пассажирами. Вместо этого они используют крайне реалистичные симуляторы, в которых пульт управления и оборудование моделируются вплоть до мельчайших деталей, что гарантирует максимально реалистичный опыт, без риска для людей. Гас Хартманн сообщает, что в сфере телекоммуникаций обычно проводят учения, связанные с выживанием во время ураганов и других природных катаклизмов. Такое моделирование привело к созданию защищенных от непогоды зданий, с генераторами, способными выдержать шторм.

В ВМФ США для проверки готовности к катастрофам используют мысленные эксперименты вида «Что если...» и тестовые прогоны. Согласно Джону Стивенсу, прогоны включают в себя «реальное повреждение оборудования, но под наблюдением; они выполняются в обязательном порядке,

каждую неделю, 2–3 дня в неделю». Для атомного флота мысленные эксперименты полезны, но их недостаточно для подготовки к реальным инцидентам. Реакции на непредвиденные ситуации должны быть отработаны столько раз, сколько требуется, чтобы они выполнялись на автомате.

Что касается спасателей, то, по словам Майка Доэрти, здесь тестирование чрезвычайных ситуаций проходит несколько иначе. Как правило, менеджер тайком просит ребенка или другого спасателя притвориться, что он тонет. Такие сценарии должны быть максимально реалистичными, чтобы спасатели не могли отличить реальные и разыгранные ситуации.

Тренировка и сертификация

Обучение и аттестация особенно важны, когда на кону стоят жизни. Например, Майк Доэрти описал, как у спасателей проводится обучение в особо трудных условиях. Курсы включают в себя физическую подготовку (например, спасатель должен быть в состоянии вытащить из воды человека любого веса), техники вроде оказания первой помощи, а также практические занятия (например, если один спасатель вошел в воду, как должны действовать другие?). Для каждого объекта предусмотрены отдельные курсы обучения, поскольку работа спасателем в бассейне значительно отличается от работы спасателем на пляже озера или океана.

Концентрация на сборе детальных требований и проектировании

Некоторые из опрошенных нами инженеров говорили о важности сбора подробных требований и создания документов, описывающих проект. Эта практика особенно важна при работе с медицинской техникой. Во многих случаях реальные условия эксплуатации или особенности

обслуживания оборудования не описаны разработчиками продукта. Поэтому требования к использованию и обслуживанию должны быть собраны из других источников.

Например, как рассказывает Эрик Гросс, аппараты для выполнения лазерной глазной хирургии максимально защищены от неосторожного обращения. Поэтому особенно важно учитывать пожелания хирургов, использующих это оборудование, и техников, ответственных за его обслуживание. В то же время бывший военный подрядчик Питер Даль рассказал, какое значение имеет детально разработанный план: создание новой системы защиты, как правило, подразумевало целый год проектирования, за которым следовало всего три недели написания кода для реализации проекта. Оба этих примера значительно отличаются от культуры запуска, принятой в компании Google, где предусмотрен гораздо более быстрый уровень изменений при обдуманном риске. Прочие отрасли (например, медицинская и военная, как говорилось ранее) имеют совершенно иной предельно допустимый уровень риска и другие требования, и эти обстоятельства влияют на все рабочие процессы.

Всесторонняя и глубокая защита

В отрасли ядерной энергетики глубокая защита — основной показатель готовности [International, 2012]. В ядерных реакторах для всех систем предусматривается резервное оборудование, которое способно заменить основное в случае сбоя. Система проектируется с несколькими уровнями защиты, включая конечный физический барьер вокруг самой станции. В ядерной отрасли глубокая защита особенно важна из-за недопустимости сбоев и инцидентов.

Культура написания постмортемов

Меры по устраниению и профилактике выявленных нарушений (Corrective and preventative action, CAPA)¹⁸⁴ — это широко распространенная концепция для улучшения надежности. Она состоит в систематических исследованиях основных причин конкретных проблем или рисков, связанных с предотвращением повторяемости этих проблем. Этот принцип воплощается в культуре SRE, связанной с безобвинительными постмортемами. Когда что-то идет не так (и, учитывая масштаб, сложность и быстрый уровень изменений в компании Google, что-то рано или поздно пойдет не так), важно оценить следующее.

- Что случилось.
- Эффективность ответа.
- Что мы можем сделать по-другому в следующий раз.
- Какие действия нужно предпринять, чтобы убедиться, что этот инцидент не произойдет снова.

Все это делается без обвинения конкретных людей. Вместо того чтобы искать виноватого, гораздо важнее понять, что пошло не так и как, будучи организацией, мы можем гарантировать, что это не случится снова. Определять, из-за кого случился сбой, контрпродуктивно. Инциденты анализируются, и отчеты о них открыто публикуются, чтобы любая команда SR-инженеров могла что-то извлечь из полученного опыта.

Наши интервью показали, что в различных отраслях предусмотрены свои аналоги постмортемов (чаще всего с

другим названием). Многие сферы контролируются государственными властями и, если что-то идет не так, обязаны предоставить отчет. Такое регулирование особенно важно, когда цена ошибки высока (например, на кону стоят человеческие жизни). Подотчетными государству организациями являются FCC (телекоммуникации), FAA (авиация), OSHA (производственные и химические отрасли), FDA (медицинское оборудование), а также множество национальных компетентных органов в Европе¹⁸⁵. Ядерная энергетика и перевозки также тщательно контролируются.

Постмортемы необходимы в том числе из соображений безопасности. В производственных и химических отраслях работа чаще всего травмоопасна из-за особых условий получения готового продукта (среди них высокая температура, давление, токсичность и коррозийность). Например, стоит обратить внимание на технику безопасности компании Alcoa. Бывший руководитель Пол О'Нил требовал от персонала в любое время суток оповещать его обо всех травмах, полученных рабочими на производстве. Каждый работник фабрики знал его домашний телефонный номер и мог лично сообщить ему о нарушениях техники безопасности¹⁸⁶.

В этих промышленных отраслях ставки настолько велики, что даже ситуации, когда кто-то чуть *не* получил травму, тщательно разбираются. Эти сценарии работают как своего рода предупредительные постмортемы. Как было сказано в выступлении на YAPC NA 2015, «существует множество потенциально опасных событий при каждой катастрофе или бизнес-кризисе, и обычно их игнорируют. Скрытая ошибка плюс соответствующее условие равнозначны краху ваших планов» [Brasseur, 2015]. Потенциально опасные события — это, по сути, катастрофы, которые ждут своего часа. Например, когда служащий не следует стандартной рабочей процедуре,

рабочий в последнюю минуту успевает избежать опасности или уборщик забывает вытереть нечаянно пролитый на лестницу чай. В следующий раз работникам может и не повезти. Британская программа конфиденциальных донесений о происшествиях, связанных с человеческим фактором (Confidential Reporting Programme for Aviation and Maritime, CHIRP) предназначена для повышения осведомленности об инцидентах в авиации и судоходстве. Согласно этой программе любой работник может анонимно отправить отчет о потенциально опасном событии. Отчеты и их анализ далее публикуются в периодических рассылках.

У спасателей своя система анализа инцидентов и планирования действий. Майк Доэрти смеется: «Если ноги спасателя оказались в воде, бумажной работы не избежать!» После любого инцидента необходимо представить детальный отчет. В случае серьезных происшествий команда коллективно разбирает каждое из них, обсуждая, что пошло не так. Далее изменяется режим работы и проводятся совместные тренировки — это помогает людям выработать уверенность в том, что в следующий раз они смогут самостоятельно справиться с инцидентом. В случае особенно шокирующих или травмирующих ситуаций с членами команды работает психолог. Спасатели могут быть хорошо подготовлены к любым ситуациям, но на практике может произойти все что угодно. Как и в компании Google, спасатели используют культуру безвинительного анализа инцидента. На возникновение происшествий влияет множество факторов, поэтому неразумно обвинять кого-то одного.

Автоматизация и снижение служебной нагрузки

По своей сути SR-инженеры — это программные инженеры, которые стремятся избегать повторяющейся реактивной работы. В нашей культуре строго прописано не допускать повторения операций, которые не приносят явной пользы. Если задача может быть автоматизирована, то зачем запускать систему, которая зависит от низкоквалифицированной однообразной работы? Автоматизация снижает операционные издержки и освобождает инженеров, чтобы те могли улучшать поддерживаемые ими сервисы.

В отраслях, которые мы рассмотрели, не пришли к единому мнению на тему «как, зачем и почему проводить автоматизацию». В некоторых сферах людям доверяют больше, чем машинам. Во время службы Джеффа Стивенсона на подводной лодке атомный флот США отказался от автоматизации в пользу набора блокировок и административных процедур. Например, по словам Джеффа, работа с клапаном требовала присутствия оператора, наблюдателя и члена команды, который был на связи с офицером, обязанным следить за реакцией на предпринятые меры. Эти операции выполнялись вручную, так как считалось, что автоматизированная система не выявит проблему, которую заметит человек. Операции на подлодке управляются цепочкой ответственных за принятие решений проверенных людей, а не одним человеком. При этом остаются опасения, что компьютеры работают так быстро, что могут совершить крупную непоправимую ошибку. Когда вы работаете с ядерными реакторами, медленный и последовательный подход важнее быстрого выполнения задачи.

Согласно Джону Ли, в финансовой сфере в последние годы также стали осторожнее использовать автоматизацию. Опыт показал, что некорректно сконфигурированное решение по автоматизации может нанести значительный урон и привести

к финансовым потерям за очень короткий промежуток времени. Например, в 2012 году компания Knight Capital Group столкнулась с «багом», который привел к потере \$440 млн всего за несколько часов^{[187](#)}. Аналогично в 2010 году фондовый рынок США обвинил капера, который попытался манипулировать рынком с помощью автоматизированных средств. Несмотря на то что рынок быстро восстановился, Flash Crash всего за 30 минут привел к потерям, исчисляемым триллионами долларов^{[188](#)}. Компьютеры могут очень быстро решать задачи, и, если задачи некорректно сконфигурированы, скорость только усугубит ситуацию.

В то же время некоторые компании используют решения по автоматизации именно потому, что компьютеры действуют быстрее людей. Согласно Эдди Кеннерли, эффективность и экономия средств являются ключевыми понятиями в производственных отраслях, а автоматизация предоставляет способ более эффективного и менее затратного выполнения задач. Помимо этого, автоматизированные действия, как правило, более надежны, чем ручная работа, что означает, что автоматизация задает более высокую планку и допускает меньшую погрешность. Дэн Шеридан рассматривает вопрос автоматизации с точки зрения ее развертывания в ядерной отрасли Великобритании. В частности, есть правило, которое указывает, что, если станция должна отреагировать на заданную ситуацию меньше чем за 30 минут, этот ответ должен быть автоматизирован.

Из опыта Мэтта Тойи, в авиационной отрасли автоматизация применяется только выборочно. Например, переход на другой ресурс в случае сбоя происходит автоматически, но, когда речь идет о других задачах, отрасль доверяет автоматизации только в том случае, если результат ее работы проверяет человек. Хотя в организации используется

большое количество сервисов автоматического наблюдения, реальные реализации систем управления воздушным трафиком должны проверяться человеком.

Согласно Эрику Гроссу, автоматизация довольно эффективно снижает количество человеческих ошибок при выполнении операций на глазах. Перед тем как приступить к операции с помощью LASIK, доктор лично осматривает пациента и анализирует данные рефракционного теста. Ранее врачу достаточно было ввести данные о пациенте и нажать кнопку, а затем в дело вступал лазер. Однако ошибки при вводе данных могли привести к большим проблемам. Например, можно перепутать данные пациентов или ошибиться при вводе значений для правого и левого глаза. Автоматизация теперь значительно снижает риск человеческой ошибки, которая способная повлиять на чье-то зрение. Компьютеризованная проверка правильности введенных вручную данных стала первым серьезным достижением: если человек-оператор введет данные, выходящие за пределы ожидаемого диапазона, система пометит этот случай как необычный. За этим нововведением последовали и другие: теперь зрачок фотографируется во время предварительного рефракционного теста. Когда приходит время делать операцию, зрачок пациента автоматически сравнивается со зрачком на фотографии, что устраняет вероятность перепутать данные пациента. Когда это автоматизированное решение было реализовано, удалось избавиться от целой группы медицинских ошибок.

Структурированное и рациональное принятие решений

В компании Google в целом и в службе SRE в частности данные критически важны. Команда стремится принимать решения

структурированно и рационально, гарантируя, что:

- основа для решений утверждается заранее, а не постфактум;
- входные данные, необходимые для принятия решения, прозрачны;
- все предположения явно указываются;
- решения на основе данных предпочтительнее решений, основанных на ощущениях, предчувствиях или мнении наиболее высокооплачиваемого сотрудника.

Служба SRE в нашей компании работает, предполагая, что все члены команды:

- заинтересованы в работе над сервисом;
- могут определить, как продолжить работу, учитывая доступные данные.

Решения должны быть осознанными, а не регламентированными, они должны приниматься без оглядки на личное мнение — даже мнение самого старшего сотрудника в комнате, которого Эрик Шмидт и Джонатан Розенберг назвали HiPPO (Highest-Paid Person's Opinion — «мнение наиболее высокооплачиваемого сотрудника») [Schmidt, 2014].

Процедура принятия решений различается в разных отраслях. Мы узнали, что в некоторых сферах используется подход *«не трохь, если работает»*. Отрасли, работающие с системами, проектирование которых потребовало длительных размышлений и большого количества усилий, часто характеризуются нежеланием менять технологию, лежащую в

их основе. Например, в области телекоммуникаций все еще используются дистанционные коммутаторы, которые были реализованы в 1980-х годах. Почему люди полагаются на технологию, разработанную десятки лет назад? Эти коммутаторы, по словам Гаса Хартмана, «довольно устойчивы». Как сообщает Дэн Шеридан, атомная промышленность также медленно меняется. Все решения обусловлены правилом: «*Если сейчас это работает, ничего не менять*».

Многие организации ориентируются на сценарии и руководства, а не на свободное решение проблем. Каждый вероятный сценарий зафиксирован в чек-листе или в специальном «блокноте». В случае аварийной ситуации в этом источнике можно найти конкретный перечень действий. Такой регламентированный подход работает для отраслей, которые развиваются относительно медленно, поскольку сценарии аварийных ситуаций не изменяются при каждом обновлении системы. В частности, он подходит для организаций, где от работников не требуют сложных навыков. Чтобы убедиться, что люди будут правильно реагировать на аварийные ситуации, лучше всего предоставить им простой и четкий набор инструкций.

Наконец, некоторые организации вроде торговых компаний для лучшего управления рисками разбивают процесс принятия решений на этапы. По словам Джона Ли, в таких организациях предусмотрено создание особой группы специалистов, которые гарантируют, что никто не предпринимает чрезмерно рискованных действий для получения прибыли. Если происходит что-то неожиданное, первой реакцией этой команды должно быть отключение системы. Как говорит Джон Ли, «если мы не торгуемся, мы не теряем деньги. Конечно, мы и не зарабатываем деньги, но хотя бы не теряем их». Только особая группа может вновь запустить систему в работу.

Итоги главы

Многие ключевые принципы SRE, применяемые в Google, хорошо прослеживаются в самых разных отраслях. Нам в Google также помогли уроки, усвоенные на опыте состоявшихся промышленных отраслей.

Основной вывод, который мы получили по итогам сравнительного изучения различных отраслей, таков: скорость работы в Google ценится выше, чем в других крупных компаниях, лидирующих в иных сегментах. Умение действовать быстро и хорошо адаптироваться — ценная вещь, но никогда не нужно забывать о различных последствиях потенциальной неудачи. Так, в ядерной энергетике, авиации или медицине неудачи и отказы могут привести к травмам или даже к гибели людей. Когда ставки высоки, можно не сомневаться, что высокая надежность будет обеспечиваться консервативными методами.

В Google нам постоянно приходится, словно канатоходцам, балансировать между пользовательскими ожиданиями (пользователь рассчитывает на высокую доступность) и прицельным стремительным внедрением изменений и инноваций. Как упоминалось выше, многие наши софтверные проекты, например поисковик, требуют осознанно решать, какова именно мера «достаточной надежности».

Большинство софтверных услуг и решений Google достаточно гибки для такой среды, где ошибки не связаны с чрезмерными рисками. Следовательно, мы можем рассчитывать на такие средства, как бюджет ошибок (обоснование его необходимости см. в главе 3), из которого «финансируется» культура инноваций и осознанного риска. В сущности, компания Google адаптировала под свои нужды известные принципы обеспечения надежности, многие из которых были отточены в других отраслях, и смогла создать

собственную уникальную культуру надежности, позволяющую учесть все переменные в сложном уравнении, сочетающем масштаб, сложность и скорость с высокой доступностью.

182 E911 (Enhanced 911 — «улучшенная 911»): линия аварийно-спасательной службы в Соединенных Штатах, которая использует данные о местоположении.

183 https://en.wikipedia.org/wiki/Safety_integrity_level.

184 https://en.wikipedia.org/wiki/Corrective_and_preventive_action.

185 https://en.wikipedia.org/wiki/Competent_authority.

186 <http://ehstoday.com/safety/nsc-2013-oneill-exemplifies-safety-leadership>.

187 См. «ФАКТЫ, раздел Б», где рассматриваются Knight и ПО Power Peg [Securities, 2013].

188 Regulators blame computer algorithm for stock market ‘flash crash’, Computerworld, <http://www.computerworld.com/article/2516076/financial-it/regulators-blame-computer-algorithm-for-stock-market-flash-crash-.html>.

34. Заключение

Автор — Бенджамин Латч [189](#)

Под редакцией Бетси Байер

Я читал эту книгу с невероятной гордостью. С того момента, как я пришел в компанию Excite в начале 1990-х, где моя команда представляла собой группу своего рода SRE-неандертальцев, отвечающих за «работу с программным обеспечением», всю свою карьеру я пытался методом тыка разобраться в процессах сборки систем. В свете моего многолетнего опыта работы в отрасли высоких технологий особенно удивительно наблюдать, как идеи SRE укоренились в Google и получили столь скорое развитие. Когда я присоединился к Google в 2006 году, служба SRE включала пару сотен программистов, а сейчас тут работают более 1000 человек, которые распределены по дюжинам площадок и поддерживают функционирование, на мой взгляд, самой интересной вычислительной инфраструктуры на планете.

Так что же заставило Google SRE за последнее десятилетие превратиться в организацию, способную интеллектуально, динамично и эффективно обслуживать такую колоссальную инфраструктуру? Я думаю, что секрет этого ошеломляющего успеха SRE кроется в самой сути его принципов действия.

Команды SRE формируются так, чтобы наши программисты распределяли свое время между двумя одинаково важными видами работы. Дежурные смены SRE-сотрудников позволяют нам наблюдать за работой всех систем, отслеживая место и характер неполадок и определяя, как их лучше масштабировать. Но у нас также остается время, чтобы провести анализ и решить, какие разработки упростят управление этими системами. По сути, нам выпал шанс

попробовать себя в роли как операторов, так и дизайнеров и программистов. Наш опыт в области разработки глобальных решений зашифрован в рабочем коде.

Этими решениями впоследствии могут без проблем воспользоваться другие SRE-команды и в конечном счете все, кто так или иначе связан с Google (или не связан... вспомните Google Cloud!) и хочет задействовать или усовершенствовать накопленный нами опыт и разработанные нами системы.

Когда вы приступаете к формированию команды или системы, за основу предпочтительно взять свод правил или аксиом, достаточно общих, чтобы быть пригодными к немедленному использованию, но при этом не теряющих актуальности со временем. Во многом именно об этом и говорит Бен Трейнор Слосс во вступлении к данной книге: о гибком, в основном ориентированном на успешную проверку временем наборе компетенций, который остается актуальным спустя 10 лет после утверждения, несмотря на изменения и рост, характерные для инфраструктуры Google и команды SRE.

В процессе повышения значимости SRE мы отметили несколько разных движущих факторов. Первый — не изменившийся со временем характер основных компетенций и задач SRE: наши системы могут быть в тысячу раз больше или быстрее, но в конечном итоге они должны оставаться надежными, гибкими, простыми в обращении в случае непредвиденной ситуации, хорошо контролируемыми и со сбалансированными мощностями. В то же время привычная деятельность SR-программистов становится более значимой по мере развития сервисов Google и повышения профессионализма самих работников. К примеру, то, что раньше было задачей «создать информационную панель для 20 машин» сейчас может представлять собой «автоматизировать

представление, создать панель инструментов и систему оповещения для парка из 10 000 машин».

Для тех, кто в прошедшее десятилетие не работал на передовых в области SRE, проще будет осмыслить динамику развития и совершенствования SRE, проведя аналогию между взглядами SRE на сложные системы и подходом к самолетостроению в авиационной промышленности. Хотя цена ошибки в этих двух отраслях совершенно разная, определенные сходства все же сохраняются.

Представьте, что 100 лет назад вам потребовалось бы совершить перелет между двумя городами. У вашего самолета, скорее всего, лишь один двигатель (или два, если вам повезло). Летчик также исполняет роль механика и, возможно, дополнительно грузчика. В кабине есть место для пилота и, если вам повезло, второго пилота или штурмана. В отличную погоду ваш самолетик оттолкнулся бы от полосы, и, если бы все прошло хорошо, вы бы медленно поднялись к облакам и в итоге приземлились в другом городе, на расстоянии, возможно, пары сотен миль от места взлета. Отказ любой системы самолета в то время приводил к катастрофическим последствиям, и нельзя было назвать беспрецедентными случаи, когда пилоту приходилось что-то ремонтировать прямо в полете! Подведенные к кабине системы были жизненно важными, простыми и нестабильными, и в большинстве случаев резервных элементов не предусматривалось.

Прыгнем на сотню лет вперед, в салон огромного Boeing 747, стоящего на взлетной полосе. Сотни пассажиров заполняют обе палубы, в то время как тонны грузов загружаются в нижний отсек. Самолет «напичкан» надежными резервированными системами. Это образец безопасности и надежности, — собственно, поднявшись на нем в небо, вы

будете в большей безопасности, нежели на земле в салоне автомобиля. Ваш самолет оторвётся от размеченной полосы на одном континенте и свободно приземлится на другой размеченной полосе на расстоянии 6000 миль точно по расписанию, в пределах пары минут от прогнозируемого времени посадки. Но загляните в кабину — кого вы там обнаружите? Все тех же двух пилотов!

Как же все остальные характеристики — безопасность, вместимость, скорость и надежность — так улучшились, в то время как в кабине по-прежнему остались те же два пилота? Ответив на этот вопрос, вы можете смело провести параллель с огромными, фантастически сложными системами под управлением SRE. Интерфейсы систем управления самолетом хорошо продуманы и достаточно доступны для того, чтобы обучение летному делу в обычных условиях не стало непосильной задачей. Однако эти же интерфейсы предоставляют достаточно маневренности, и люди, управляющие ими, хорошо обучены, поэтому реакция на непредвиденную ситуацию будет отлаженной и быстрой. Кабину проектировали люди, которые разбираются в сложных системах и понимают, как сделать их простыми для освоения. Подведенные к кабине системы характеризуются теми же параметрами, которые обсуждались в этой книге: доступность, оптимизация производительности, управление изменениями, мониторинг и оповещение, планирование производительности, а также реагирование на чрезвычайные ситуации.

В конечном итоге цель SRE — придерживаться похожего курса. Команда SRE должна быть одновременно компактной и способной действовать на высоком уровне абстракции, полагаясь на множество вспомогательных систем вроде отказоустойчивых интерфейсов и продуманных API для

общения с системами. В то же время команда SRE должна обладать всесторонними познаниями о системах — как они работают, как отказывают, как нужно реагировать на отказы, — и это их повседневная работа.

[189](#) Вице-президент SRE в Google, Inc.

Приложения

А. Таблица доступности

Доступность, как правило, рассчитывается на основе периода времени, в течение которого сервис был недоступен. Таблица А.1 показывает допустимое время простоя для достижения заданного уровня доступности (при условии отсутствия плановых простоев).

Таблица А.1. Таблица доступности

Уровень доступности, %	Допустимое окно недоступности					
	В год	В квартал	В месяц	В неделю	В день	В час
90	36,5 дня	9 дней	3 дня	16,8 часа	2,4 часа	6 минут
95	18,25 дня	4,5 дня	1,5 дня	8,4 часа	1,2 часа	3 минуты
99	3,65 дня	21,6 часа	7,2 часа	1,68 часа	14,4 минуты	36 секунд
99,5	1,83 дня	10,8 часа	3,6 часа	50,4 минуты	7,2 минуты	18 секунд
99,9	8,76 часа	2,16 часа	43,2 минуты	10,1 минуты	1,44 минуты	3,6 секунды
99,95	4,38 часа	1,08 часа	21,6 минуты	5,04 минуты	43,2 секунды	1,8 секунды
99,99	52,6 минуты	12,96 минуты	4,32 минуты	60,5 секунды	8,64 секунды	0,36 секунды
99,999	5,26 минуты	1,3 минуты	25,9 секунды	6,05 секунды	0,87 секунды	0,04 секунды

Использовать совокупный показатель недоступности (к примеру, «X % всех невыполненных операций») более эффективно, чем фокусироваться на продолжительности сбоев для сервисов, которые могут быть частично доступны, — к примеру, ввиду наличия нескольких копий, нагрузка которых

варьируется в течение дня или недели, а не остается постоянной.

См. формулы (3.1) и (3.2) в главе 3 для расчетов.

Б. Практические рекомендации для сервисов в промышленной эксплуатации

Автор — Бен Трейнор Слосс

Под редакцией Бетси Байер

Не бойтесь неудач

Проверяйте и редактируйте ввод конфигурации и отвечайте на неверный ввод *одновременно* продолжением работы и соответствующим оповещением.

- *Неправильные данные.* Проверяйте как синтаксис, так и, если это возможно, семантику. Следите за пустыми, а также частичными или усеченными данными (к примеру, предусматривайте оповещение в том случае, если конфигурация на N % меньше предыдущей версии).
- *Опоздавшие данные.* Могут привести к аннулированию текущих данных из-за задержек. Запланируйте оповещения, которые будут отправляться задолго до ожидаемого времени истечения данных.

Ошибайтесь так, чтобы можно было продолжить работу, пусть даже рискуя показаться слишком либеральным или бесхитростным. Мы пришли к выводу, что для систем в целом безопаснее продолжать работу в прежней конфигурации и ожидать подтверждения от человека для использования новых, возможно, неверных данных.

Примеры

В 2005 году глобальная DNS-система Google, отвечающая за загрузку и балансировку времени ожидания, в качестве разрешений для файла получила пустой начальный файл DNS. Система приняла его и 6 минут обслуживала ответ NXDOMAIN для всех параметров Google. В качестве ответной меры сейчас система выполняет ряд проверок доступности на новых конфигурациях, включая подтверждение наличия виртуальных IP для **google.com**, и продолжает обслуживать предыдущие записи DNS до получения нового файла, который прошел проверку ввода.

В 2009 году ввод неправильных (но допустимых) данных привел к тому, что Google отметила всю сеть как содержащую вредоносные программы [Mayer, 2009]. Конфигурационный файл со списком подозрительных URL был заменен слешем (/), который соответствовал всем URL. Такие изменения не вступили бы в силу, если бы были проведены проверки резких изменений размера файла и соответствий конфигурации сайтам, которые едва ли могут содержать вредоносные программы.

Постепенное выведение на рынок

Несрочные выпуски должны проходить в несколько этапов. Как конфигурационные, так и бинарные изменения несут определенный риск, и вы его сглаживаете, одновременно применяя изменения к небольшим долям трафика и производительности. Масштаб вашего сервиса или релиза, как

и ваш профиль риска, покажут в процентном отношении производительность производства, для которого вы готовите новый выпуск сервиса, и подходящий промежуток времени между этапами. Рекомендуется также выполнять различные этапы на разных территориях, чтобы выявить проблемы с суточными циклами трафика и различия в структуре трафика, обусловленные географическими факторами.

Выпуски новых версий должны быть контролируемы. Чтобы гарантировать отсутствие любых неожиданностей в процессе релиза, за ним должен наблюдать либо выполняющий его программист, либо — предпочтительно — надежная система контроля. При обнаружении неожиданного поведения первым делом необходимо сделать откат и впоследствии провести диагностику, чтобы минимизировать среднее время восстановления.

Определяйте целевые значения с точки зрения пользователя

Измеряйте доступность и производительность с точки зрения важных для конечного пользователя условий. Подробно этот вопрос обсуждается в главе 4.

Пример

Измерение частоты ошибок и времени ожидания в клиенте Gmail вместо сервера привело к существенному сокращению наших оценок доступности Gmail и повлекло изменения как в клиенте Gmail, так и в серверном коде. В результате доступность Gmail за пару лет возросла от 99,0 до 99,9 %.

Бюджет ошибок

Надежность баланса и темп нововведений с бюджетом ошибок (см. раздел «Обоснование критерия суммарного уровня ошибок (бюджета ошибок)» главы 3) вместе определяют допустимый уровень отказов для сервиса за определенный период — для нас это месяц. Бюджет представляет собой просто единицу за вычетом показателя целевого уровня обслуживания; к примеру, сервис с целевой доступностью 99,99 % располагает бюджетом недоступности 0,01 %. Пока сервис не исчерпал свой бюджет ошибок в течение месяца через фоновую величину ошибок в сочетании с любым показателем простоя, команда разработчиков может свободно (хотя и в пределах разумного) вводить новые возможности, применять обновления и т.д.

Если бюджет ошибок исчерпан, изменения замораживаются (за исключением неотложных решений в области безопасности и багов, применяемых к любой причине увеличения ошибок) до момента, пока сервис не вернется в рамки бюджета или не начнется новый месяц. Для крупных сервисов с целевым уровнем обслуживания выше 99,99 % вполне применимо квартальное, а не месячное обновление бюджета по причине малого показателя допустимого времени простоя.

Бюджеты ошибок устраниют структурное напряжение, которое в противном случае может развиться между SRE и командами разработчиков продуктов. Они получают простой, управляемый данными механизм для оценки риска запуска. Бюджеты также предоставляют как службе SRE, так и команде разработчиков общую цель для развития технологий и практик разработки, которые позволяют скорейшее внедрение нововведений и больше запусков без «раздувания бюджета».

Мониторинг

При мониторинге предусмотрены лишь три типа выходного документа.

- *Оповещения*. Человек немедленно должен предпринять какие-то действия.
- *Тикеты*. Человек должен предпринять какие-то действия в течение пары дней.
- *Журналирование*. Никому не нужно заниматься этими журналами прямо сейчас, но их при необходимости делают доступными для последующего анализа.

Если для вызова человека достаточно оснований, то требуются незамедлительные действия (то есть мы имеем дело с оповещением) или вы распознаете проблему как баг и заносите его в вашу систему отслеживания ошибок. Размещать оповещения в электронных письмах и надеяться, что кто-нибудь прочтет их все и заметит важные, эквивалентно сваливанию их в `/dev/null`: в конечном счете их проигнорируют. История показывает, что такая стратегия привлекательна, но опасна, поскольку работает какое-то время, но опирается на бесконечную человеческую бдительность, и по этой причине неизбежное отключение, когда оно все-таки произойдет, будет еще более разрушительным.

Постмортемы

Постмортемы (см. главу 15) должны быть безвинительными и фокусироваться на процессе и технологиях, а не на людях. Предполагайте, что люди, вовлеченные в инцидент, умны, действовали из лучших побуждений и принимали наилучшие

решения из возможных с доступной им в тот период информацией. Из этого следует, что мы не можем исправить людей, но должны вместо этого исправлять их окружение: иными словами, улучшать конструкцию системы, чтобы избегать целых классов проблем, делать нужную информацию легкодоступной и автоматически подтверждать оперативные решения, чтобы системы было затруднительно подвергнуть риску опасного состояния.

Планирование производительности

Представим необходимость справиться одновременно с запланированным и незапланированным отключением с сохранением доступа к пользовательскому интерфейсу, что приводит нас к конфигурации $N + 2$, где пиковую нагрузку могут обрабатывать N экземпляров (возможно, в режиме ограниченной функциональности), в то время как два крупнейших экземпляра остаются недоступными.

- Проверяйте предварительные прогнозы спроса на соответствие реальности, пока они не покажут корректное сочетание. Расхождение ведет к нестабильному прогнозированию, неэффективному распределению ресурсов и риску дефицита производительности.
- Используйте нагрузочное тестирование вместо традиционного, чтобы установить отношение ресурса к производительности; кластер из X машин мог справиться с Y запросами 3 месяца назад, но сохранил ли он эту способность с учетом изменений в системе?
- Не путайте нагрузку первого дня с нагрузкой устойчивого состояния. Запуски часто привлекают больше трафика, и

вместе с тем это тот период, когда вам особенно необходимо представить продукт с наилучшей стороны. См. главу 27 и приложение Д.

Перегрузки и отказы

Перегруженные сервисы должны выдавать умеренные, но приближенные к оптимальным результаты. К примеру, при перегрузке Google Search будет осуществлять поиск по меньшему количеству параметров и прекратит обслуживание функционала вроде Instant, чтобы и дальше предоставлять возможность поиска по Сети. Протестируйте систему веб-поиска, задав высокие значения производительности, чтобы обеспечить удовлетворительную работу сервиса при перегрузке трафиком.

В периоды, когда нагрузка настолько высока, что даже слабый отклик серьезно потребляет ресурсы, применяйте постепенную сегментацию нагрузки (см. главу 21). Прочие техники предусматривают ответы на запросы со значительной задержкой («режим увязания») и выбор последовательного подмножества клиентов, чтобы можно было получать сообщения об ошибках, не затрагивая работу пользователей.

Повторения могут увеличить вероятность ошибки при высоких уровнях трафика, что приведет к каскадным отключениям (см. главу 22). Отвечайте на каскадные отключения проходами долей трафика (включая повторения!) вверх по системе, как только общая нагрузка превысит общую производительность.

Каждый клиент, производящий вызов удаленных процедур (RPC), при планировании повторных попыток должен использовать экспоненциальный откат (с небольшой погрешностью), чтобы не допустить распространения ошибки.

Особое внимание стоит уделить мобильным клиентам, поскольку их могут быть миллионы и изменение их кода занимает длительное время — возможно, недели — и требует от пользователя установки обновлений.

Команды SRE

Команды SRE должны тратить не больше 50 % своего времени на оперативную работу (см. главу 5), нагрузку сверх этого следует перенаправлять команде разработки продукта. Многие команды также подключают разработчиков продуктов к дежурствам и работе с тикетами, даже если в текущий момент перегрузки не наблюдается. Это стимулирует проектировать системы, которые минимизируют или устраняют оперативную переработку, а также обеспечивают постоянный контакт между разработчиками продуктов и оперативной стороной сервиса. Регулярные рабочие совещания между SRE и командой разработчиков (см. главу 31) также полезны.

Мы выяснили, что в состав дежурной команды должны входить как минимум восемь человек, чтобы можно было избежать переутомления, а также поддерживать постоянный штат и низкую текучесть. Предпочтительно размещать этих дежурных в двух значительно отдаленных географически местах (к примеру, в Калифорнии и Ирландии), чтобы избавить их оточных вызовов. В этом случае минимальный состав команды — шесть человек на каждой стороне.

Ожидайте иметь дело не более чем с двумя событиями за дежурство (или за каждые 12 часов): на реагирование и устранение отключений, начало изучения и документирование обнаруженных ошибок требуется время. Более частые события могут снизить качество реагирования и свидетельствуют о том, что что-то неладно с конструкцией системы,

чувствительностью системы контроля или с реакциями на обнаруженные ошибки.

Как это ни парадоксально, если вы учтете эти практические рекомендации, команда SRE в конечном счете может остаться без практики реагирования на инциденты ввиду их нечастого возникновения и краткий сбой превратится в длительный. Упражняйтесь в устраниении гипотетических сбоев (см. подраздел «Катастрофа: ролевая игра» раздела «Пять приемов для вдохновления дежурных работников» главы 28) в рабочем порядке и в процессе улучшайте документацию, касающуюся инцидентов.

В. Пример документа о происшествиях

Shakespeare Sonnet++ Перегрузка: 2015-10-21

Информация об управлении сбоем: <http://incident-management-cheat-sheet>.

(Ответственный по связям поддерживает обновление сводки).

Краткая информация: поисковый сервис Shakespeare в состоянии каскадного отключения по причине нововыявленного сонета, не учтенного в поисковом индексе.

Статус: активный, сбой #465.

Командный (-ые) пункт (-ы): #shakespeare в IRC.

Иерархия управляющих (все ответственные):

- сейчас управляет инцидентом: jennifer;
- глава оперативных работников: docbrown;
- глава плановой группы: jennifer;
- ответственный по связям: jennifer;
- следующий управляющий инцидентом: *не определен.*

(Обновлять минимум каждые 4 часа и передавать исполняющему роль ответственного по связям.)

Подробный статус (в последний раз обновлялся 2015-10-21 в 15:28 UTC jennifer).

Критерии выхода:

- новый сонет добавлен к текстовой базе данных Shakespeare **ПЛАН**
- в пределах доступности (99,99 %) и периода ожидания (99 % < 100 мс) SLO для 30+ минут **ПЛАН**

Список ПЛАН и задокументированные ошибки:

- выполнить функцию MapReduce для переиндексирования текстовой базы данных Shakespeare **ГОТОВО**
- задействовать чрезвычайные резервы для привлечения дополнительной производительности **ГОТОВО**
- использовать потоковый накопитель для балансировки нагрузки между кластерами (баг 5554823) **ПЛАН**

Хронология происшествия (*последние события размещаются первыми: время отображается в формате UTC*)

- 2015-10-21 15:28 UTC jennifer:
 - увеличение производительности в два раза в глобальном масштабе.
- 2015-10-21 15:28 UTC jennifer:
 - направление всего трафика в неприоритетный кластер USA-2 и отвод трафика из других кластеров для их восстановления после каскадного отключения и одновременного развертывания большего количества задач;

- выполнение функции индекса MapReduce завершено, ожидание репликации Bigtable на все кластеры.

- 2015-10-21 15:10 UTC martym:

- добавление нового сонета к текстовой базе данных Shakespeare и запуск индекса MapReduce.

- 2015-10-21 15:04 UTC martym:

- получение текста обнаруженного нового сонета из списка рассылки **shakespeare-discuss@**.

- 2015-10-21 15:01 UTC docbrown:

- по причине каскадного отключения объявлен сбой.

- 2015-10-21 14:55 UTC docbrown:

- шквал оповещений, ManyHttp500s во всех кластерах.

Г. Пример постмортема

Shakespeare Sonnet++ Постмортем (сбой #465)

Дата: 2015-10-21.

Авторы: jennifer, martym, agoogler.

Статус: завершен, предпринимаются меры.

Краткая информация: поисковая система Shakespeare была недоступна на протяжении 66 минут в период очень высокого интереса к Shakespeare по причине открытия нового сонета.

Последствия[190](#): по предварительным подсчетам, потеряно 1,21 миллиарда запросов, доход не упал.

Исходные причины[191](#): каскадный отказ по причине сочетания исключительно высокой нагрузки и утечки ресурсов, когда поисковые запросы не обслуживались из-за отсутствия в текстовой базе данных Shakespeare. Новооткрытый сонет использовал слово, которое никогда не появлялось в работах Shakespeare и которое по совпадению оказалось поисковым запросом пользователей. В нормальных условиях доля отказа задач из-за утечки ресурсов является достаточно низкой, чтобы остаться незамеченной.

Запускающее событие: скрытая ошибка, активированная внезапным притоком трафика.

Решение: перенаправление трафика в неприоритетный кластер и добавление десятикратной производительности, чтобы устранить каскадное отключение. Развернут обновленный индекс, разрешая проблему взаимодействия со скрытой ошибкой. Поддержание дополнительной производительности, пока приток пользовательского интереса

к новому сонету не иссякнет. Утечка ресурсов определена, проблема решена.

Обнаружение: Borgmon обнаружил высокий уровень HTTP 500s и сообщил дежурным.

Перечень мер [192](#)

Мера	Тип	Ответственное лицо	Ошибка
Добавить в план инструкции для реагирования на каскадное отключение	Минимизация	jennifer	Н/д ГОТОВО
Задействовать потоковый накопитель для балансировки нагрузки между кластерами	Предотвращение	martyum	Ошибка 5554823 ПЛАН
Запланировать тест на каскадное отключение во время следующего процесса DiRT	Процесс	docbrown	Н/д ПЛАН
Непрерывно исследовать переменный указатель MR/fusion	Предотвращение	jennifer	Ошибка 5554824 ПЛАН
УстраниТЬ утечку из дескриптора файлов в поисковой подсистеме ранжирования	Предотвращение	agoogler	Ошибка 5554825 ГОТОВО
Добавить функционал сегментации нагрузки в поисковую систему Shakespeare	Предотвращение	agoogler	Ошибка 5554826 ПЛАН
Создать регрессионные тестирования для подтверждения нормального реагирования серверов на мертвые запросы	Предотвращение	clarac	Ошибка 5554827 ПЛАН
Добавить к продукту обновленную поисковую систему ранжирования	Предотвращение	jennifer	Ошибка н/д ГОТОВО
Остановить работы до 2015-11-20 из-за исчерпания бюджета ошибок или считать исключение возникшим ввиду гротескных, невероятных, странных и беспрецедентных обстоятельств	Другое	docbrown	Н/д ПЛАН

Полученные уроки

Что прошло хорошо

- Система контроля быстро оповестила нас о высокой доле (достигавшей 100 %) HTTP 500s.
- Быстрое обновление текстовой базы данных Shakespeare во всех кластерах.

Что прошло плохо

- У нас нет практики реагирования на каскадные отключения.
- Мы превысили разрешенный бюджет ошибок (на несколько порядков) из-за исключительного притока трафика, который в результате привел к отказам.

В чем нам повезло[193](#)

- В списке рассылки приверженцев Shakespeare оказалась доступная копия нового сонета.
- В журналах сервера были отслеживания стека, указывающие на опустошение дескриптора файла как причину отказа.
- Мертвые запросы были устранены в результате загрузки нового индекса, содержащего популярный поисковой запрос.

Хронология[194](#)

2015-10-21 (*время везде указано в формате UTC*)

- 14:51 Появляются сообщения о том, что новый сонет Шекспира был обнаружен у Delorean.
- 14:53 Трафик по запросам о Шекспире возрастает в 88 раз после того, как пост в /r/shakespeare указывает на поисковую систему Shakespeare как место поиска нового сонета (вот только нового сонета у нас на тот момент не было).
- 14:54 **НАЧИНАЕТСЯ ОТКЛЮЧЕНИЕ** — поисковые серверы перегружаются.
- 14:55 docbrown получает шквал оповещений, ManyHttp500s из всех кластеров.
- 14:57 Сбой всего трафика по поисковому запросу Shakespeare: см. [http://monitor/shakespeare?
end_time=20151021T145700](http://monitor/shakespeare?end_time=20151021T145700).
- 14:58 docbrown начинает расследование, обнаруживает очень высокую долю отказов сервера.
- 15:01 **НАЧАЛО ИНЦИДЕНТА** — docbrown объявляет инцидент #465 по причине каскадного отключения, координация действий в #shakespeare, jennifer назначается управляющей инцидентом.
- 15:02 По совпадению кто-то отправляет письмо в **shakespeare-discuss@** с обнаруженным сонетом, который оказывается вверху списка входящих сообщений martym.
- 15:03 jennifer оповещает список рассылки **shakespeare-incidents@** об инциденте.

- 15:04 martym находит текст нового сонета и ищет документацию по обновлению текстовой базы данных.
- 15:06 docbrown обнаруживает, что симптомы отказа идентичны для всех задач во всех кластерах, начинает искать причину в журналах приложений.
- 15:07 martym находит документацию, начинает подготовительные работы по обновлению текстовой базы данных.
- 15:10 martym добавляет сонет к известным произведениям Шекспира, начинает работу по индексации.
- 15:12 docbrown связывается с clarac и agoogler (из команды разработчиков Shakespeare) и просит помочи с изучением базы исходного кода в поисках возможных причин.
- 15:18 clarac обнаруживает в журналах явный след, указывающий на опустошение дескриптора файлов, подтверждает, что утечка существует, так как поиск в текстовой базе данных не выполняется.
- 15:20 martym завершает работу с индексом MapReduce.
- 15:21 jennifer и docbrown решают увеличить количество экземпляров до показателя, при котором они будут способны работать под нагрузкой на приемлемом уровне до того, как их отключат и перезагрузят.
- 15:23 docbrown выравнивает нагрузку трафика в кластере USA-2, позволяя увеличить количество экземпляров в других кластерах без немедленного отключения сервиса.

- 15:25 martym начинает распространять новый индекс на все кластеры.
- 15:28 docbrown начинает увеличивать количество экземпляров в два раза.
- 15:32 jennifer меняет баланс нагрузки для увеличения трафика в приоритетных кластерах.
- 15:33 Начало отказа задач в приоритетных кластерах, симптомы остаются прежними.
- 15:34 В вычислениях обнаружено порядковое возрастание количества ошибок при увеличении числа экземпляров.
- 15:36 jennifer возвращает исходные показатели балансировки нагрузки для повторного использования неприоритетного кластера USA-2 в подготовке к дополнительному глобальному увеличению количества экземпляров в пять раз (в общей сумме в десятикратном размере от исходной производительности).
- 15:36 **ОТКЛЮЧЕНИЕ УМЕНЬШЕНО**, обновленный индекс распространен на все кластеры.
- 15:39 docbrown начинает второй этап увеличения количества экземпляров до десятикратного размера от начальной производительности.
- 15:41 jennifer восстанавливает балансировку нагрузки по всем кластерам для 1 % трафика.
- 15:43 Доли ошибки HTTP 500 в приоритетных кластерах не превышают допустимых пределов, случайные отказы задач — на низком уровне.

- 15:45 jennifer распределяет 10 % трафика по приоритетным кластерам.
- 15:47 Доля HTTP 500 в приоритетных кластерах остается в пределах SLO, отказов задач не наблюдается.
- 15:50 30 % трафика распределено по приоритетным кластерам.
- 15:55 50 % трафика распределено по приоритетным кластерам.
- 16:00 **ОТКЛЮЧЕНИЕ ЗАКОНЧЕНО**, весь трафик распределен по всем кластерам.
- 16:30 **ИНЦИДЕНТ ЗАКОНЧЕН**, достигнут выходной критерий 30 минут номинальной производительности.

Вспомогательная информация

Панель мониторинга, [http://monitor/shakespeare?
end_time=20151021T160000&duration=7200](http://monitor/shakespeare?end_time=20151021T160000&duration=7200).

[190](#) Под последствиями понимается воздействие на пользователей, изменение дохода и т.д.

[191](#) Пояснение обстоятельств, при которых произошел данный инцидент. Полезно использовать такие техники, как «Пять почему» [Ohno, 1988] для понимания сопутствующих факторов.

[192](#) Рефлекторные действия часто оказываются чрезесчур экстремальными для осуществления, и, возможно, их следует здраво пересмотреть в более широком контексте. Есть риск излишней оптимизации для конкретной проблемы, добавления специальных средств контроля и оповещения, в то время как надежные механизмы вроде тестирования модулей могут обнаружить проблемы на самых ранних стадиях процесса разработки.

[193](#) Этот раздел посвящен происшествиям без последствий.

194 Это «сценарий» инцидента. Используйте хронологию инцидента из документа управления происшествием для первоначального заполнения хронологии постмортема, а затем дополняйте другими релевантными записями.

Д. Список действий для координации запуска

Ниже приведен оригинальный список действий координации запуска от Google, датированный примерно 2005 годом, в немного сокращенном виде.

Архитектура

- Набросок архитектуры, типы серверов, типы запросов от клиентов.
- Программные запросы клиентов.

Машины и дата-центры

- Машины и загрузка сети, центры обработки данных, избыточность $N + 2$, качество обслуживания сети.
- Новые доменные имена, балансировка нагрузки DNS.

Оценки объема, пропускной способности и производительности

- Оценки трафика HTTP и загрузки сети, скачок запуска, смешанный трафик, полугодовой перерыв.
- Нагрузочный тест, комплексный тест, пропускная способность одного дата-центра и максимальное время ожидания.
- Влияние на остальные сервисы, о которых мы беспокоимся больше всего.

- Производительность хранилища.

Надежность системы и преодоление отказа

- Что происходит, если:
 - отказывает машина, отключаются стойка данных или кластер;
 - отключается сеть между двумя дата-центрами.
- Для каждого типа сервера, который обращается к другим серверам (его внутренним источникам):
 - как определить отказ сервера и что в этом случае делать;
 - как прекратить работу или перезапуститься, не затрагивая клиентов или пользователей.
 - балансировка нагрузки, ограничение скорости, время ожидания, повторы и поведение при устранении ошибок.
- Резервное копирование/восстановление данных, аварийное восстановление.

Система мониторинга и управление серверами

- Мониторинг внутреннего состояния, комплексный мониторинг поведения, управление оповещениями.
- Контроль мониторинга.
- Финансово важные оповещения и журналы.

- Советы по управлению серверами с кластерной средой.
- Не отключайте почтовые сервера, отправляя самому себе почтовые оповещения в собственном серверном коде.

Безопасность

- Обзор конструкции системы безопасности, аудит защитного кода, риски спама, опознавание, SSL.
- Предпусковая доступность/контроль доступа, различные типы черных списков.

Автоматические и ручные задачи

- Методы и контроль изменений для обновления серверов, данных и конфигураций.
- Процесс освобождения, повторяемые компоновки, канареечные тесты для реального трафика, поэтапные отправки кода.

Вопросы роста

- Дополнительная пропускная способность, десятикратный рост, увеличение оповещений.
- Ограничение расширяемости, линейное масштабирование, аппаратное масштабирование, необходимые изменения.
- Кэширование, фрагментация/дефрагментация данных.

Внешние зависимые объекты

- Сторонние системы, контроль, работа с сетью, объем трафика, скачки запуска.
- Постепенное отключение, способы избежать случайной перегрузки сторонних сервисов.
- Хорошие отношения с организационными партнерами, почтовыми системами, сервисами внутри Google.

График и планирование отправок

- Жесткие дедлайны, внешние события, понедельники или пятницы.
- Стандартные рабочие процедуры для данного сервиса и для других сервисов.

Е. Пример протокола рабочего совещания

Дата: 2015-10-23.

Присутствующие: agoogler, clarac, docbrown, jennifer, martym.

Повестка: крупное отключение (#465), превышение бюджета ошибок.

Анализ предыдущих пунктов действий

- Утвердить Goat Teleporter для использования (ошибка 1011101):

- нелинейные показатели ускорения массы теперь удается прогнозировать, планируем добиться точного нацеливания в ближайшие несколько дней.

Обзор отключения

- Новый сонет (отключение 465):
 - 1,21 миллиарда запросов потеряны из-за каскадного отключения после взаимодействия между скрытой ошибкой (утечка из дескриптора файлов в безрезультатные запросы) + отсутствие нового сонета в текстовой базе данных + беспрецедентный и неожиданный объем трафика;
 - ошибка утечки дескриптора файлов устранена (ошибка 5554825), продукт обновлен;
 - оценка возможности использования потокового накопителя для балансировки нагрузки (ошибка 5554823)

и применения фрагментации нагрузки (ошибка 5554823) для предотвращения повторения;

- исчерпанный бюджет ошибок; заставляет остановить работу продукта на один месяц, если только docbrown не получит разрешение на исключение на основании того, что событие было странным и непредвиденным (но в этом случае одобрение маловероятно).

Срочные события

- `AnnotationConsistencyTooEventual`: был объектом вызова пять раз за неделю, вероятно, из-за межрегиональной задержки репликации между Bigtables:
 - расследование еще в процессе, см. ошибку 4821600;
 - скорого решения не ожидается, решено поднять допустимый порог целостности, чтобы уменьшить количество оповещений, на которые пока невозможно ответить.

Несрочные события

- Нет.

Контроль изменений и/или периодов тишины

- `AnnotationConsistencyTooEventual`, допустимый порог задержки поднят от 60 до 180 секунд, см. ошибку 4821600; ПЛАН (`martym`).

Запланированные изменения в продукции

- Кластер USA-1 отключен для обслуживания в период между 2015-10-29 и 2015-11-02:

- реагирования не требуется, трафик автоматически перенаправится в другие кластеры региона.

Ресурсы

- Заемствованы ресурсы для реагирования на инцидент sonnet++, будут привлечены дополнительные экземпляры серверов, ресурсы будут возвращены на следующей неделе.
- Использование на 60 % ЦП, 75 % ОП, 44 % диска (увеличение с 40, 70, 40 % на прошлой неделе).

Параметры ключевых сервисов

- **OK** 99 % время ожидания $88 \text{ мс} < 100 \text{ мс}$ целевой показатель SLO (контроль 30 дней).
- **ПЛОХО** доступность: $86,95 \% < 99,99 \%$ целевой показатель SLO (контроль 30 дней).

Дискуссия/Обновление по проектам

- Запуск проекта Moliere в течение 2 недель.

Новые меры к принятию

- ПЛАН (martym): поднять порог AnnotationConsistencyTooEventual.

- ПЛАН (docbrown): вернуть количество экземпляров к обычному показателю и возвратить ресурсы.

Об авторах

Бетси Бейер работает составителем технической документации в компании Google в Нью-Йорке и специализируется на SRE. Ранее она создавала документацию для команд Google's Data Center и Hardware Operations в Маунтин-Вью, а также для распределенных данных центров. До переезда в Нью-Йорк Бетси читала лекции о разработке технической документации в Стэнфордском университете. В студенчестве она изучала международные отношения и английскую литературу в университетах Стэнфорда и Тулана.

Крис Джоунс работает SR-инженером для системы Google App Engine, облачного продукта «платформа как услуга», который обслуживает более 28 миллионов запросов в день. Живет в Сан-Франциско. Ранее отвечал за статистику рекламы, хранение данных и системы поддержки пользователей. Помимо этого, Крис работал в сфере академических информационных технологий, анализировал данные политических кампаний и немного участвовал в «допиливании» ядра BSD. Он имеет научные степени в области информатики, экономики и технической политики.

Дженнифер Петофф работает программным менеджером в команде SR-инженеров и проживает в Дублине, Ирландия. Она управляла крупными глобальными проектами во многих областях, включая научные исследования, инженерное дело, кадровую службу и рекламу. Дженифер присоединилась к компании Google после того, как 8 лет проработала в химической отрасли. У нее имеется степень доктора наук в области химии, полученная в Стэнфордском университете, а также степень бакалавра наук в области химии и бакалавра гуманитарных наук в области психологии, полученные в университете Рочестера.

Нейл Мёрфи руководит командой специалистов по обеспечению надежности рекламных сайтов в ирландском подразделении компании Google. Он более 20 лет занимается интернет-проектами и в настоящее время председательствует в INEX — центре обмена трафиком в Ирландии. Нейл внес свой вклад в написание множества технических документов и книг, включая *IPv6 Network Administration* издательства O'Reilly, и нескольких RFC. Сейчас он участвует в описании ирландского сегмента Интернета. Нейл имеет несколько ученых степеней в областях вычислительной техники, математики и поэзии (вот тут, несомненно, какая-то ошибка). Живет в Дублине с женой и двумя сыновьями.

Библиография

Adams Bram, Bellomo Stephany, Bird Christian, Marshall-Keim Tamara, Khomh Foutse, Moir Kim. The Practice and Future of Release Engineering: A Roundtable with Three Release Engineers (<http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=434819>). *IEEE Software*, vol. 32, no. 2 (March/April 2015), pp. 42–49.

Aguilera M.K. Stumbling over Consensus Research: Misunderstandings and Issues (<http://dl.acm.org/citation.cfm?id=2172342>). *Replication, Lecture Notes in Computer Science* 5959, 2010.

Allspaw J., Robbins J. *Web Operations: Keeping the Data on Time*: O'Reilly, 2010.

Allspaw J. Blameless PostMortems and a Just Culture (<https://codeascraft.com/2012/05/22/blameless-postmortems/>). Blog post, 2012.

Allspaw J. Trade-Offs Under Pressure: Heuristics and Observations of Teams Resolving Internet Service Outages (<http://lup.lub.lu.se/student-papers/record/8084520/file/8084521.pdf>). MSc thesis, Lund University, 2015.

Anantharaju S. Automating web application security testing (<https://googleonlinesecurity.blogspot.com/2007/07/automating-web-applicationsecurity.html>). Blog post, July 2007.

Ananatharayan R. et al. Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams (<https://research.google.com/pubs/pub41318.html>). *SIGMOD '13*, 2013.

Andrieux A., Czajkowski K., Dan A. et al. Web Services Agreement Specification (WS-Agreement)

(<http://www.opendaylight.org/documents/GFD.107.pdf>). September 2005.

Bailis P., Ghodsi A. Eventual Consistency Today: Limitations, Extensions, and Beyond (<http://dl.acm.org/citation.cfm?id=2462076>). *ACM Queue*, vol. 11, no. 3, 2013.

Bainbridge L. Ironies of Automation ([http://dx.doi.org/10.1016/0005-1098\(83\)90046-8](http://dx.doi.org/10.1016/0005-1098(83)90046-8)). *Automatica*, vol. 19, no. 6, November 1983.

Baker J. et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services (<https://research.google.com/pubs/pub36971.html>). *Proceedings of the Conference on Innovative Data System Research*, 2011.

Barroso L.A. Warehouse-Scale Computing: Entering the Teenage Decade (<http://dl.acm.org/citation.cfm?id=2019527>). Talk at 38th Annual Symposium on Computer Architecture, video available online, 2011.

Barroso L.A., Clidaras J., Holzle U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, Second Edition (<https://research.google.com/pubs/pub41606.html>). Morgan & Claypool, 2013.

Bennett C., Tseitlin A. Chaos Monkey Released Into The Wild (<http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html>). Blog post, July 2012.

Bland M. Goto Fail, Heartbleed, and Unit Testing Culture (<http://martinfowler.com/articles/testing-culture.html>). Blog post, June 2014.

Bock L. Work Rules! (<https://www.workrules.net>). Twelve Books, 2015.

Bolosky W.J., Bradshaw D., Haagens R.B., Kusters, Li P. Paxos Replicated State Machines as the Basis of a High-Performance Data Store

(https://www.usenix.org/legacy/event/nsdi11/tech/full_papers/Bolosky.pdf). *Proc. NSDI 2011*, 2011.

Boysen P.G. Just Culture: A Foundation for Balanced Accountability and Patient Safety (<http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3776518/>). *The Ochsner Journal*, Fall 2013.

Brasseur VM. Failure: Why it happens & How to benefit from it (<https://youtu.be/DLn4fZsZsKM?t=29m05s>). YAPC 2015.

Brewer E. Lessons From Giant-Scale Services (<http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=939450>). *IEEE Internet Computing*, vol. 5, no. 4, July/August 2001.

Brewer E. CAP Twelve Years Later: How the Rules Have Changed (<http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6133253>). *Computer*, vol. 45, no. 2, February 2012.

Brooker M. Exponential Backoff and Jitter (<http://www.awsarchitectureblog.com/2015/03/backoff.html>). *AWS Architecture Blog*, March 2015.

Brooks Jr. F.P. No Silver Bullet — Essence and Accidents of Software Engineering. *The Mythical Man-Month*, Boston: Addison-Wesley, 1995, pp. 180–186.

Brutlag J. Speed Matters (<http://googleresearch.blogspot.com/2009/06/speedmatters.html>). *Google Research Blog*, June 2009.

Bull G.M. *The Dartmouth Time-sharing System*. Ellis Horwood, 1980.

Burgess M. *Principles of Network and System Administration*. Wiley, 1999.

Burrows M. The Chubby Lock Service for Loosely-Coupled Distributed Systems (<https://research.google.com/archive/chubby.html>). OSDI'06:

Seventh Symposium on Operating System Design and Implementation, November 2006.

Burns B., Grant B., Oppenheimer D., Brewer E., Wilkes J. Borg, Omega, and Kubernetes (<http://dl.acm.org/citation.cfm?id=2898444>). *ACM Queue*, vol. 14, no. 1, 2016.

Castro M., Liskov B. Practical Byzantine Fault Tolerance (<http://www.pmg.lcs.mit.edu/papers/osdi99.pdf>). *Proc. OSDI 1999*, 1999.

Chambers C., Raniwala A., Perry F., Adams S., Henry R., Bradshaw R., Weizenbaum N. FlumeJava: Easy, Efficient Data-Parallel Pipelines (<http://research.google.com/pubs/pub35650.html>). *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

Chandra T.D., Toueg S. Unreliable Failure Detectors for Reliable Distributed Systems (<http://dl.acm.org/citation.cfm?id=226647>). *J. ACM*, 1996.

Chandra T., Griesemer R., Redstone J. Paxos Made Live — An Engineering Perspective (http://research.google.com/archive/paxos_made_live.html). *PODC '07: 26th ACM Symposium on Principles of Distributed Computing*, 2007.

Chang F. et al. Bigtable: A Distributed Storage System for Structured Data (<https://research.google.com/archive/bigtable.html>). *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, November 2006.

Chrousous G.P. Stress and Disorders of the Stress System (<http://www.ncbi.nlm.nih.gov/pubmed/19488073>). *Nature Reviews Endocrinology*, vol. 5, no. 7, 2009.

Clos C. A Study of Non-Blocking Switching Networks (<http://dx.doi.org/10.1002/j.1538-7305.1953.tb01433.x>). *Bell System*

Technical Journal, vol. 32, no. 2, 1953.

Contavalli C., Gaast W. van der, Lawrence D., Kumari W. Client Subnet in DNS Queries (<https://tools.ietf.org/html/draft-vandergaast-edns-clientsubnet>). *IETF Internet-Draft*, 2015.

Conway M.E. Design of a Separable Transition-Diagram Compiler (<http://dl.acm.org/citation.cfm?id=366704>). *Commun. ACM* 6, 7 (July 1963), 396–408.

Conway P. Preservation in the Digital World (<http://www.clir.org/pubs/reports/conway2/index.html>). Report published by the Council on Library and Information Resources, 1996.

Cook R.I. How Complex Systems Fail (<http://web.mit.edu/2.75/resources/random/How%20Complex%20Systems%20Fail.pdf>). *Web Operations*: O'Reilly, 2010.

Corbett J.C. et al. Spanner: Google's Globally-Distributed Database (<https://research.google.com/archive/spanner.html>). *OSDI '12: Tenth Symposium on Operating System Design and Implementation*, October 2012.

Cranmer J. Visualizing code coverage (<https://quetzalcoatal.blogspot.com/2010/03/visualizing-code-coverage.html>). Blog post, March 2010.

Dean J., Barroso L.A. The Tail at Scale (<http://research.google.com/pubs/pub40801.html>). *Communications of the ACM*, vol. 56, 2013.

Dean J., Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters (<https://research.google.com/archive/mapreduce.html>). *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, December 2004.

Dean J. Software Engineering Advice from Building Large-Scale Distributed Systems

(<https://static.googleusercontent.com/media/research.google.com/en/people/jeff/stanford-295-talk.pdf>). Stanford CS297 class lecture, Spring 2007.

Dekker S. Reconstructing human contributions to accidents: the new view on error and performance (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.411.4985&rep=rep1&type=pdf>). *Journal of Safety Research*, vol. 33, no. 3, 2002.

Dekker S. *The Field Guide to Understanding “Human Error”*, 3rd edition: Ashgate, 2014.

Dickson C. How Embracing Continuous Release Reduced Change Complexity (<http://usenix.org/conference/ures14west/summit-program/presentation/dickson>). Presentation at USENIX Release Engineering Summit West 2014, video available online.

Durmer J., Dinges D. Neurocognitive Consequences of Sleep Deprivation (<http://www.ncbi.nlm.nih.gov/pubmed/15798944>). *Seminars in Neurology*, vol. 25, no. 1, 2005.

Eisenbud D.E. et al. Maglev: A Fast and Reliable Software Network Load Balancer (<https://research.google.com/pubs/pub44824.html>). *NSDI’16: 13th USENIX Symposium on Networked Systems Design and Implementation*, March 2016.

Erenkrantz J.R. Release Management Within Open Source Projects (<http://www.erenkrantz.com/Geeks/Research/Publications/ReleaseManagement.pdf>). *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, Oregon, May 2003.

Fischer M.J., Lynch N.A., Paterson M.S. Impossibility of Distributed Consensus with One Faulty Process (<http://dl.acm.org/citation.cfm?id=214121>). *J. ACM*, 1985.

Fitzpatrick B.W., Collins-Sussman B. *Team Geek: A Software Developer’s Guide to Working Well with Others*: O'Reilly, 2012.

Floyd S., Jacobson V. The Synchronization of Periodic Routing Messages (<http://dl.acm.org/citation.cfm?id=187045>). *IEEE/ACM Transactions on Networking*, vol. 2, issue 2, April 1994, pp. 122–136.

Ford D. et al. Availability in Globally Distributed Storage Systems (<http://research.google.com/pubs/pub36737.html>). *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

Fox A., Brewer E.A. Harvest, Yield, and Scalable Tolerant Systems (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=798396). *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, March 1999.

Fowler M. GUI Architectures (<http://martinfowler.com/eaaDev/uiArchs.html>). Blog post, 2006.

Gall J. *SYSTEMANTICS: How Systems Really Work and How They Fail*, 1st ed. Pocket, 1977.

Gall J. *The Systems Bible: The Beginner's Guide to Systems Large and Small*, 3rd ed. General Systemantics Press/Liberty, 2003.

Gawande A. *The Checklist Manifesto: How to Get Things Right*. Henry Holt and Company, 2009.

Ghemawat S., Gobioff H., Leung S-T. The Google File System (<https://research.google.com/archive/gfs.html>). *19th ACM Symposium on Operating Systems Principles*, October 2003.

Gilbert S., Lynch N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services (<http://dl.acm.org/citation.cfm?id=564601>). *ACM SIGACT News*, vol. 33, no. 2, 2002.

Glass R. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2002.

Golab W. et al. Eventually Consistent: Not What You Were Expecting? (<http://dl.acm.org/citation.cfm?id=2582994>). *ACM Queue*, vol. 12, no. 1, 2014.

Graham P. Maker's Schedule, Manager's Schedule (<http://paulgraham.com/makersschedule.html>). Blog post, July 2009.

Gupta A., Shute J. High-Availability at Massive Scale: Building Google's Data Infrastructure for Ads (<https://research.google.com/pubs/pub44686.html>). *Workshop on Business Intelligence for the Real Time Enterprise*, 2015.

Hamilton J. On Designing and Deploying Internet-Scale Services (<https://www.usenix.org/legacy/event/lisa07/tech/hamilton.html>). *Proceedings of the 21st Large Installation System Administration Conference*, November 2007.

Hanks S., Li T., Farinacci D., Traina P. Generic Routing Encapsulation over IPv4 networks (<https://tools.ietf.org/html/rfc1702>). *IETF Informational RFC*, 1994.

Hickins M. Tape Rescues Google in Lost Email Scare (<http://blogs.wsj.com/digits/2011/03/01/tape-rescues-google-in-lost-email-scare/>). *Digits, Wall Street Journal*, 1 March 2011.

Hixson D. Capacity Planning (<https://www.usenix.org/publications/login/feb15/capacity-planning>). ;login:, vol. 40, no. 1, February 2015.

Hixson D. The Systems Engineering Side of Site Reliability Engineering (<https://www.usenix.org/publications/login/june15/hixson>). ;login: vol. 40, no. 3, June 2015.

Hodges J. Notes on Distributed Systems for Young Bloods (<https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-youngbloods/>). Blog post, 14 January 2013.

Holmwood L. Applying Cardiac Alarm Management Techniques to Your On-Call (<http://fractio.nl/2014/08/26/cardiac-alarms-and-ops/>). Blog post, 26 August 2014.

Humble J., Read C., North D. The Deployment Production Line. *Proceedings of the IEEE Agile Conference*, July 2006.

Humble J., Farley D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

Hunt P., Konar M., Junqueira F.P., Reed B. ZooKeeper: Wait-free coordination for Internet-scale systems (https://www.usenix.org/legacy/events/atc10/tech/full_papers/Hunt.pdf). USENIX ATC, 2010.

International Atomic Energy Agency, Safety of Nuclear Power Plants: Design, SSR-2/1 (http://www-pub.iaea.org/MTCD/publications/PDF/Pub1534_web.pdf), 2012.

Jain S. et al. B4: Experience with a Globally-Deployed Software Defined WAN (<https://research.google.com/pubs/pub41761.html>). *SIGCOMM'13*.

Jones C., Underwood T., Nukala S. Hiring Site Reliability Engineers (<https://www.usenix.org/publications/login/june15/hiring-site-reliability-engineers>). ;login:, vol. 40, no. 3, June 2015.

Junqueira F., Mao Y., Marzullo K. Classic Paxos vs. Fast Paxos: Caveat Emptor (<http://dl.acm.org/citation.cfm?id=1323158>). *Proc. HotDep '07*, 2007.

Junqueira F.P., Reid B.C., Serafini M. Zab: High-performance broadcast for primary-backup systems. (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5958223&tag=1). *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference* on 27 Jun 2011: 245–256.

Kahneman D. *Thinking, Fast and Slow*. Farrar, Straus and Giroux, 2011.

Karger D. et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web (<http://dl.acm.org/citation.cfm?id=258660>). *Proc. STOC '97*, 29th annual ACM symposium on theory of computing, 1997.

Kemper C. Build in the Cloud: How the Build System Works (<https://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html>). *Google Engineering Tools* blog post, August 2011.

Kendrick S. What Takes Us Down? (<http://usenix.org/publications/login/october-2012-volume-37-number-5/what-takes-us-down>). ;login:, vol. 37, no. 5, October 2012.

Kincaid Jason. T-Mobile Sidekick Disaster: Danger's Servers Crashed, And They Don't Have A Backup. *Techcrunch*. n.p., 10 Oct. 2009. Web. 20 Jan. 2015, <http://techcrunch.com/2009/10/10/t-mobile-sidekick-disaster-microsofts-serverscrashed-and-they-dont-have-a-backup>.

Kingsbury K. The trouble with timestamps (<http://www.aphyr.com/posts/299-the-trouble-with-timestamps>). Blog post, 2013.

Kirsch J., Amir Y. Paxos for System Builders: An Overview (<http://dl.acm.org/citation.cfm?id=1529979>). *Proc. LADIS '08*, 2008.

Klau R. How Google Sets Goals: OKRs (<https://library.gv.com/how-google-sets-goals-okrs-a1f69b0b72c7>). Blog post, October 2012.

Klein D.V. A Forensic Analysis of a Distributed Two-Stage Web-Based Spam Attack (https://www.usenix.org/legacy/event/lisa06/tech/klein_klein_html/index.html). *Proceedings of the 20th Large Installation System Administration Conference*, December 2006.

Klein D.V., Betser D.M., Monroe M.G. Making Push On Green a Reality (<https://www.usenix.org/publications/login/october-2014->

[vol-39-no-5/makingpush-green-reality](#)). ;login:, vol. 39, no. 5, October 2014.

Krattenmaker T. Make Every Meeting Matter (<https://hbr.org/2008/02/make-every-meeting-matter>). *Harvard Business Review*, February 27, 2008.

Kreps J. Getting Real About Distributed System Reliability (<http://blog.empathybox.com/post/19574936361/getting-real-about-distributed-systemreliability>). Blog post, 19 March 2012.

Krishan K. Weathering The Unexpected (<http://dl.acm.org/citation.cfm?id=2366332>). *Communications of the ACM*, vol. 55, no. 11, November 2012.

Kumar A. et al. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing (<https://research.google.com/pubs/pub43838.html>). *SIGCOMM '15*.

Lamport L. The Part-Time Parliament (<http://research.microsoft.com/enus/um/people/lamport/pubs/lamport-paxos.pdf>). *ACM Transactions on Computer Systems* 16, 2, May 1998.

Lamport L. Paxos Made Simple (<http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf>). *ACM SIGACT News* 121, December 2001.

Lamport L. Fast Paxos (<http://research.microsoft.com/pubs/64624/tr-2005-112.pdf>). *Distributed Computing* 19.2, October 2006.

Limoncelli T.A., Chalup S.R., Hogan C.J. *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2*. Addison-Wesley, 2014.

Loomis J. How to Make Failure Beautiful: The Art and Science of Postmortems. *Web Operations*: O'Reilly, 2010.

Lu H. et al. Existential Consistency: Measuring and Understanding Consistency at Facebook (<http://sigops.org/sosp/sosp15/current/2015-Monterey/printable/240-lu.pdf>). *SOSP '15*, 2015.

Mao Y., Junqueira F.P., Marzullo K. Mencius: Building Efficient Replicated State Machines for WANs (https://www.usenix.org/legacy/events/osdi08/tech/full_papers/mao/mao.pdf). *OSDI '08*, 2008.

Maslow A.H. A Theory of Human Motivation. *Psychological Review* 50 (4), 1943.

Maurer B. Fail at Scale (<http://dl.acm.org/citation.cfm?id=2839461>). *ACM Queue*, vol. 13, no. 12, 2015.

Mayer M. This site may harm your computer on every search result?!?! (<https://googleblog.blogspot.com/2009/01/this-site-may-harm-your-computeron.html>). Blog post, January 2009.

McIlroy M.D. A Research Unix Reader: Annotated Excerpts from the Programmer's Manual, 1971–1986 (<http://www.cs.dartmouth.edu/~doug/reader.pdf>).

McNutt D. Maintaining Consistency in a Massively Parallel Environment (<https://www.usenix.org/conference/ucms13/summit-program/presentation/mcnutt>). Presentation at USENIX Configuration Management Summit 2013, video available online.

McNutt D. Accelerating the Path from Dev to DevOps (https://www.usenix.org/system/files/login/articles/05_mcnutt.pdf). *login:*, vol. 39, no. 2, April 2014.

McNutt D. The 10 Commandments of Release Engineering (https://www.youtube.com/watch?v=RNMjYV_UsQ8). Presentation at 2nd International Workshop on Release Engineering 2014, April 2014.

McNutt D. Distributing Software in a Massively Parallel Environment (<https://www.use-->

nix.org/conference/lisa14/conference-program/presentation/mcnutt). Presentation at USENIX LISA 2014, video available online.

Microsoft TechNet. What is SNMP? Last modified March 28, 2003, <https://technet.microsoft.com/en-us/library/cc776379%28v=ws.10%29.aspx>.

Meadows D. *Thinking in Systems*. Chelsea Green, 2008.

Menage P. *Adding Generic Process Containers to the Linux Kernel* (<https://www.kernel.org/doc/ols/2007/ols2007v2-pages-45-58.pdf>). *Proc. Of Ottawa Linux Symposium*, 2007.

Merchant N. Culture Trumps Strategy, Every Time (<https://hbr.org/2011/03/culture-trumps-strategy-every>). *Harvard Business Review*, March 22, 2011.

Mockapetris P. Domain Names — Implementation and Specification (<https://tools.ietf.org/html/rfc1035>). *IETF Internet Standard*, 1987.

Moler C. Matrix Computation on Distributed Memory Multiprocessors. *Hypercube Multiprocessors 1986*, 1987.

Moraru I., Andersen D.G., Kaminsky M. Egalitarian Paxos (<http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-12-108.pdf>). *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-12-108*, 2012.

Moraru I., Andersen D.G., Kaminsky M. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes (<http://dl.acm.org/citation.cfm?id=2671001>). *Proc. SOCC '14*, 2014.

Morgenthaler J.D., Gridnev M., Sauciuc R., Bhansali S. Searching for Build Debt: Experiences Managing Technical Debt at Google (<https://research.google.com/pubs/pub37755.html>). *Proceedings of the 3rd Int'l Workshop on Managing Technical Debt*, 2012.

Narla C., Salas D. Hermetic Servers (<http://googletesting.blogspot.com/2012/10/hermetic->

[servers.html](#)). Blog post, 2012.

Nelson B. The Data on Diversity (<http://dl.acm.org/citation.cfm?id=2684442.2597886>). *Communications of the ACM*, vol. 57, 2014.

Nichols K., Jacobson V. Controlling Queue Delay (<http://dl.acm.org/citation.cfm?id=2209336>). *ACM Queue*, vol. 10, no. 5, 2012.

O'Connor P., Kleyner A. *Practical Reliability Engineering*, 5th edition. Wiley, 2012.

Ohno T. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1988.

Ongaro D., Ousterhout J. In Search of an Understandable Consensus Algorithm (Extended Version) (<https://ramcloud.stanford.edu/raft.pdf>).

Peng D., Dabek F. Large-scale Incremental Processing Using Distributed Transactions and Notifications (<https://research.google.com/pubs/pub36726.html>). *Proc. of the 9th USENIX Symposium on Operating System Design and Implementation*, November 2010.

Perrow C. *Normal Accidents: Living with High-Risk Technologies*. Princeton University Press, 1999.

Perry A.R. Engineering Reliability into Web Sites: Google SRE (<https://research.google.com/pubs/pub32583.html>). *Proc. of LinuxWorld 2007*, 2007.

Pike R., Dorward S., Griesemer R., Quinlan S. Interpreting the Data: Parallel Analysis with Sawzall (<https://research.google.com/archive/sawzall.html>). *Scientific Programming Journal* vol. 13, no. 4, 2005.

Potvin R., Levenberg J. The Motivation for a Monolithic Codebase: Why Google stores billions of lines of code in a single repository. *Communications of the ACM*, forthcoming July 2016.

Video available on YouTube (<https://www.youtube.com/watch?v=W71BTkUbdqE>).

Rooney J.J., Vanden Heuvel L.N. Root Cause Analysis for Beginners (<http://asq.org/quality-progress/2004/07/quality-tools/root-cause-analysis-forbeginners.html>). *Quality Progress*, July 2004.

Saint Exupery A. de. Hommes Terre des. Paris: Le Livre de Poche, 1939, in translation by Lewis Galantiere as *Wind, Sand and Stars*.

Sambasivan R.R., Fonseca R., Shafer I., Ganger G.R. So, You Want To Trace Your Distributed System? Key Design Insights from Years of Practical Experience (http://pdl.cmu.edu/PDL-FTP/SelfStar/CMU-PDL-14-102_abs.shtml). Carnegie Mellon University Parallel Data Lab Technical Report CMUPDL-14-102, 2014.

Santos N., Schiper A. Tuning Paxos for High-Throughput with Batching and Pipelining (http://rd.springer.com/chapter/10.1007%2F978-3-642-25959-3_11). *13th Int'l Conf. on Distributed Computing and Networking*, 2012.

Sarter N.B., Woods D.D., Billings C.E. Automation Surprises. *Handbook of Human Factors & Ergonomics*, 2nd edition, G. Salvendy (ed.), Wiley, 1997.

Schmidt E., Rosenberg J., Eagle A. How Google Works (<http://www.howgoogleworks.net>): Grand Central Publishing, 2014.

Schwartz B. The Factors That Impact Availability, Visualized (<https://www.vividcortex.com/blog/the-factors-that-impact-availability-visualized>). Blog post, 21 December 2015.

Schneider F.B. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial (<http://dl.acm.org/citation.cfm?id=98167>). *ACM Computing Surveys*, vol. 22, no. 4, 1990.

Securities and Exchange Commission, Order In the Matter of Knight Capital Americas LLC (<https://www.sec.gov/litigation/admin/2013/34-70694.pdf>). File 3-15570, 2013.

Shao G., Berman F., Wolski R. Master/Slave Computing on the Grid (<http://www.cs.ucsb.edu/~rich/publications/shao-hcw.pdf>). *Heterogeneous Computing Workshop*, 2000.

Shute J. et al. F1: A Distributed SQL Database That Scales (<https://research.google.com/pubs/pub41344.html>). *Proc. VLDB 2013*, 2013.

Sigelman B.H. et al. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure (<https://research.google.com/pubs/pub36356.html>). Google Technical Report, 2010.

Singh A. et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network (<https://research.google.com/pubs/pub43837.html>). *SIGCOMM '15*.

Skelton M. Operability can Improve if Developers Write a Draft Run Book (<http://blog.soft-wareoperability.com/2013/10/16/operability-can-improve-if-developers-write-a-draft-run-book/>). Blog post, 16 October 2013.

Treynor Sloss B. Gmail back soon for everyone (<http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html>). Blog post, 28 February 2011.

Tatham S. How to Report Bugs Effectively (<http://www.chiark.greenend.org.uk/~sgtatham/bugs.html>), 1999.

Verma A., Pedrosa L., Korupolu M.R., Oppenheimer D., Tune E., Wilkes J. Large-scale cluster management at Google with Borg (<https://research.google.com/pubs/pub43438.html>). *Proceedings of the European Conference on Computer Systems*, 2015.

Wallace D.R., Fujii R.U. Software Verification and Validation: An Overview (http://www-usr.inf.ufsm.br/~ceretta/papers/fujii89_software_vv.pdf). *IEEE Software*, vol. 6, no. 3 (May 1989), pp. 10, 17.

Ward R., Beyer B. BeyondCorp: A New Approach to Enterprise Security (<https://www.usenix.org/publications/login/dec14/ward>). *login:*, vol. 39, no. 6, December 2014.

Whittaker J.A., Arbon J., Carollo J. *How Google Tests Software*: Addison-Wesley, 2012.

Wood A. Predicting Software Reliability (<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=544240>). *Computer*, vol. 29, no. 11, 1996.

Wright H.K. Release Engineering Processes, Their Faults and Failures (<http://www.hyrumwright.org/papers/dissertation.pdf>), (section 7.2.2.2) PhD Thesis, University of Texas at Austin, 2012.

Wright H.K., Perry D.E. Release Engineering Practices and Pitfalls (<http://www.hyrumwright.org/papers/icse2012.pdf>), in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. (IEEE, 2012), pp. 1281–1284.

Wright H.K., Jasper D., Klimek M., Carruth C., Wan Z. Large-Scale Automated Refactoring Using ClangMR (<http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/41342.pdf>). *Proceedings of the 29th International Conference on Software Maintenance (ICSM '13)*, (IEEE, 2013), pp. 548–551.

ZooKeeper Project (Apache Foundation), ZooKeeper Recipes and Solutions (<http://zookeeper.apache.org/doc/trunk/recipes.html>). ZooKeeper 3.4 documentation, 2014.