УДК 62
ББК 32.972
Р64

Розенталь К., Джонс Н.
Р64 Хаос-инжиниринг / пер. с англ. В. С. Яценкова. – М.: ДМК Пресс, 2021. – 284 с.: ил.

ISBN 978-5-97060-796-1

Хаос-инжиниринг – относительно новое, однако уже широко востребованное направление в разработке ПО. Тысячи компаний разных размеров и разного уровня развития используют этот метод в качестве основного инструмента тестирования и контроля, чтобы сделать свои продукты и услуги более безопасными и надежными.

Эта книга охватывает историю рождения хаос-инжиниринга, фундаментальные теории, лежащие в его основе, определения и принципы, примеры реализации в масштабных вычислительных системах, примеры за пределами традиционного программного обеспечения, а также возможные перспективы развития подобных практик. Реальные истории от отраслевых экспертов из Google, Microsoft, Slack, LinkedIn и других компаний помогут читателю оценить преимущества хаос-инжиниринга во всей полноте.

Издание предназначено для разработчиков и инженеров по эксплуатации, стремящихся повысить устойчивость сложных корпоративных систем для достижения бизнес-целей.

УДК 62
ББК 32.972

Authorized Russian translation of the English edition of Chaos Engineering © 2021 by DMK Press. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-492-04386-7 (англ.)

ISBN 978-5-97060-796-1 (рус.)

© Casey Rosenthal, Nora Jones, Nathan Aschbacher, 2020
© Оформление, издание, перевод, ДМК Пресс, 2021

Мы посвящаем эту книгу Дэвиду Хассману.
Дейв был той искрой, которая превратила команду
Chaos Engineering Team в сообщество

Содержание

Предисловие

Революция хаос-инжиниринга свершилась! Тысячи компаний всех форм и размеров, достигшие разного уровня развития, используют хаос-инжиниринг в качестве основного инструмента тестирования и контроля, чтобы сделать свои продукты и услуги более безопасными и надежными. Существует множество ресурсов по этой теме, в частности выступления на конференциях, но ни один из них не дает полной картины.

Нора и Кейси задумали написать самую полную книгу о хаос-инжиниринге. Это весьма непростая задача, учитывая широту и разнообразие отрасли и развивающийся характер дисциплины. В этой книге мы попытаемся охватить историю рождения хаос-инжиниринга, фундаментальные теории, лежащие в его основе, определения и принципы, примеры реализации в масштабных вычислительных системах, примеры за пределами традиционного программного обеспечения и будущее, которое, на наш взгляд, ожидает подобные практики.

Соглашение о терминах в русском переводе книги

В этой книге мы используем уже устоявшееся в среде специалистов кириллическое написание термина *хаос-инжиниринг* для обозначения стратегического подхода к тестированию и идеологии *хаоса* в целом, а термины Chaos, Chaos Engineering Team, Chaos Monkey и другие слова в исходном английском написании с заглавной буквы обозначают названия рабочих групп, инструментов и технологий, а также зарегистрированные марки компаний и их продуктов.

Условные обозначения и соглашения, принятые в книге

В книге используются следующие типографские соглашения:

- *курсив* – для смыслового выделения важных положений, новых терминов, имен команд и утилит, а также имен и расширений файлов и каталогов;
- `моноширинный шрифт` – для листингов программ, а также в обычном тексте для обозначения имен переменных, функций, типов, объектов, баз данных, переменных среды, операторов, ключевых слов и других программных конструкций и элементов исходного кода.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры, для того чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите нам о ней главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и O'Reilly очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Благодарности

Мы можем назвать бесчисленное множество людей, которые вложили в подготовку этой книги время и силы, а также оказали эмоциональную поддержку авторам, редакторам и соавторам. Трудно переоценить объем помощи, полученной нами при создании сборника, включающего целых шестнадцать авторов (основные авторы Нора и Кейси, а также четырнадцать соавторов). Мы ценим все усилия соавторов, их терпение по отношению к нам, когда мы дорабатывали идеи и объем глав, а также помощь в процессе редактирования.

Нам повезло работать с замечательными редакторами и персоналом в O'Reilly. Амелия Блевинс (Amelia Blevins), Вирджиния Уилсон (Virginia Wilson), Джон Девинс (John Devins) и Никки Макдональд (Nikki McDonald) сыграли важную роль в создании этой книги. Во многих отношениях эта книга – произведение, созданное Амелией и Вирджинией так же, как и авторами. Спасибо за ваше терпение с нами и за многие, многие переносы сроков.

Мы ценим энтузиазм наших рецензентов: Уилла Гальего (Will Gallego), Райана Франца (Ryan Frantz), Эрика Доббса (Eric Dobbs), Лейна Десборо (Lane Desborough), Рэндала Хансена (Randal Hansen), Майкла Кехо (Michael Kehoe), Матиаса Лафельдта (Mathias Lafeldt), Барри О'Рейли (Barry O'Reilly), Синди Сридхарана (Cindy Sridharan) и Бенджамина Уилмса (Benjamin Wilms). Ваши комментарии, предложения и исправления значительно улучшили качество этой работы. Кроме того, ваши советы привели нас к дополнительным исследованиям, которые мы включили в книгу. По сути, эта книга стала результатом нашей совместной работы с вами.

Мы многим обязаны нашим соавторам, это: Джон Алспоу (John Allspaw), Питер Альваро (Peter Alvaro), Натан Ашбахер (Natan Aschbacher), Джейсон Кахун (Jason Cahoon), Раджи Чокайян (Raji Chockaiyan), Ричард Кроули (Richard Crowley), Боб Эдвардс (Bob Edwards), Энди Флинер (Andy Fleener), Расс Майлз (Russ Miles), Аарон Ринхарт (Aaron Rinhart), Логан Розен (Logan Rosen), Олег Сурмачев, Лю Тан (Lu Tang) и Хао Вэн (Hao Weng). Очевидно, что данной книги не было бы без вас. Каждый из вас внес необходимый и принципиально важный вклад в содержание. Мы ценим вас как единомышленников и друзей.

Мы хотим поблагодарить Дэвида Хассмана (David Hassman), Кента Бека (Kent Beck) и Джона Алспоу. Дэвид, которому посвящена эта книга, призвал нас пропагандировать идеологию хаос-инжиниринга за пределами нашего ограниченного профессионального сообщества в Кремниевой долине. Во многом благодаря его вовлеченности и поддержке хаос-инжиниринг стал осязаемой «вещью» – самостоятельной дисциплиной в широком мировом сообществе разработчиков программного обеспечения. В свою очередь, Кент Бек призвал нас воспринимать хаос-инжиниринг как инструмент, намного более серьезный, чем мы думали, способный изменить взгляды людей на создание, развертывание и эксплуатацию программного обеспечения. Джон Алспоу дал нам фундаментальные основы хаос-инжиниринга, подтолкнув нас заняться изучением человеческого фактора и систем безопасности в Лундском университете в Швеции. Он познакомил нас с областью Resilience Engineering (технологии отказоустойчивости), которая послужила фундаментом для хаос-инжиниринга и является тем объективом, через который мы рассматриваем проблемы надежности (включая доступность и безопасность), когда изучаем социально-технические системы, такие как широкомасштабное программное обеспечение. Все руководители и коллеги по программе в Лунде глубоко повлияли на наше мышление, особенно Йохан Бергстрем (Johan Bergstrom) и Энтони Смоукер (Antony Smoker).

Мы благодарим всех вас за влияние, которое вы оказали на нас, за то, что вдохнули в нас смелость продвигать идею хаос-инжиниринга в общество, и за то неизгладимое влияние, которое оказали на наше мировоззрение.

Введение. Рождение хаос-инжиниринга

Хаос-инжиниринг все еще остается относительно новым направлением в разработке программного обеспечения. В этом введении мы расскажем историю метода, начиная со скромного первого использования и заканчивая нынешней эпохой, когда все основные представители информационной отрасли переняли идеологию хаос-инжиниринга в той или иной форме. За последние три года вопрос «Должны ли мы заниматься хаос-инжинирингом?» превратился в вопрос «Каким образом лучше всего внедрить хаос-инжиниринг?».

История нашей зарождающейся дисциплины объясняет, как мы перешли от первого вопроса ко второму. Мы хотим не просто перечислить даты и события. Мы хотим рассказать живую историю о том, как это произошло, чтобы вы поняли, почему это произошло именно так, и какие уроки вы можете извлечь из этого пути, чтобы получить максимальную отдачу от практического применения.

История начинается в Netflix, где работали авторы этой книги Кейси Розенталь и Нора Джонс, когда команда Chaos Team создавала и внедряла технологию Chaos Engineering[1]. Netflix получил от этой технологии ощутимую выгоду, и как только другие компании увидели это, вокруг новой дисциплины возникло сообщество, которое распространило ее по всему миру.

Менеджмент в виде кода

Начиная с 2008 года Netflix приступил к масштабному переходу[2] с собственного центра обработки данных на внешний облачный сервис. В августе того же года из-за крупного сбоя базы данных в центре обработки данных Netflix не мог отправлять DVD-диски в течение трех дней. Это было до того, как потоковое видео стало вездесущим; рассылка DVD по почте составляла основу бизнеса компании.

В то время считалось, что центр обработки данных в силу своей архитектуры содержит несколько глобальных точек отказа, таких как большие

---

[1] Кейси Розенталь в течение трех лет создавал технологию и руководил командой разработчиков Chaos в Netflix. Нора Джонс присоединилась к команде разработчиков Chaos на раннем этапе в качестве инженера и технического лидера. Она отвечала за важные архитектурные решения в создаваемых инструментах, а также занималась внедрением.

[2] *Yury Izrailevsky, Stevan Vlaovic, Ruslan Meshenberg.* Completing the Netflix Cloud Migration // Netflix Media Center, Feb. 11, 2016, https://oreil.ly/c4YTI.

базы данных и вертикально масштабируемые компоненты. Переход к облаку означает использование горизонтально масштабируемых компонентов, что уменьшает количество и влияние точек отказа.

Увы, все пошло не по плану. Прежде всего потребовалось целых восемь лет, чтобы избавиться от центра обработки данных. Хотя теоретически облачные технологии больше отвечали интересам компании, переход к горизонтально масштабируемым облачным решениям не привел к желаемому повышению времени безотказной работы потокового сервиса[1].

Чтобы понять причину, мы должны вспомнить, что в 2008 году облачный сервис Amazon Web Services (AWS) был значительно менее зрелым, чем сейчас. Облачные вычисления еще не были товаром, а вариантов развертывания по умолчанию, которые мы имеем сегодня, просто не существовало. В то время облачный сервис действительно имел много особенностей, и одна из этих особенностей заключалась в том, что *экземпляры*[2] иногда прекращали свою работу без предупреждения. Такая форма сбоя считалась редкостью в центре обработки данных, где большие мощные машины тщательно обслуживались, а специфические особенности конкретных машин были хорошо известны. В облачной среде, где такую же вычислительную мощность обеспечивали за счет большого количества относительно слабых машин, работающих на обычном «железе», это, к сожалению, было обычным явлением.

Методы построения систем, устойчивых к такой форме отказа, были хорошо известны. Можно перечислить полдюжины распространенных методов, которые помогают системе автоматически компенсировать внезапный выход из строя одного из компонентов: избыточные узлы в кластере, ограничение области поражения путем увеличения количества узлов и уменьшения относительной мощности каждого узла, избыточное развертывание в разных регионах, автоматическое масштабирование и автоматизация обнаружения неисправностей и т. д. Конкретные средства, способные сделать систему нечувствительной к выходу экземпляров из строя, не имели значения. Они могли даже быть разными в зависимости от контекста системы. Важно было это сделать как можно скорее, потому что потоковый сервис столкнулся с ограничением доступности из-за высокой частоты отказов облачных экземпляров. В некотором смысле Netflix просто раздробил и умножил эффект единой точки отказа.

Однако Netflix отличался от других разработчиков масштабного программного обеспечения. Он активно продвигал культурные принципы, которые вытекают из уникальной философии управления компании и сильно повлияли на подходы к решению проблемы надежности, например:

- Netflix нанимал только старших разработчиков, которые имели опыт работы в той роли, для которой их наняли;
- Netflix предоставил всем разработчикам полную свободу действий, необходимых для решения проблемы, вкупе с ответственностью за любые последствия, связанные с этими действиями;

---

[1] В этой книге мы будем считать критерием доступности системы «время безотказной работы», или «аптайм» (uptime).

[2] В облачной технологии термин «экземпляр» соответствует виртуальной машине или серверу в предшествующем отраслевом языке.

- важно отметить, что Netflix полностью делегировал людям, делающим свою работу, право принятия решений в рамках этой работы;
- руководство не указывало отдельным исполнителям, что делать, а вместо этого позаботилось о том, чтобы исполнители поняли проблемы, которые необходимо решить. Затем исполнители рассказали руководству, как они планируют решить эти проблемы, и вместе приступили к работе над решением;
- по-настоящему продуктивные команды хорошо сбалансированы и мало связаны друг с другом. Если у всех одинаковое понимание цели, меньше усилий расходуется на процессы, формальное согласование или управление задачами.

Эти принципы являются частью производственной культуры Netflix и оказали любопытное влияние на зарождение идеи хаос-инжиниринга. Поскольку в задачу руководства не входили прямые указания исполнителям, в Netflix не нашлось какого-то одного человека, команды или группы, которые диктовали бы остальным разработчикам, как писать свой код. Хотя разработка полдюжины общих шаблонов для написания сервисов, устойчивых к выпадению экземпляров, – вполне решаемая задача, культура компании не позволяла приказать всем разработчикам строго следовать этим инструкциям.

Netflix пришлось искать другой путь.

Рождение Chaos Monkey

Разработчики перепробовали много разных подходов, но сработало как надо и осталось жить только одно решение – Chaos Monkey. Это очень простое приложение просматривает список кластеров, выбирает один случайный экземпляр из каждого кластера и внезапно отключает его в течение рабочего дня. И это происходит каждый рабочий день.

Звучит жестоко, но никто не ставил перед собой цель расстроить коллег. Разработчики знали, что этот тип отказа – выпадающие экземпляры – в любом случае произойдет с каждым кластером. Приложение Chaos Monkey дало им возможность проактивно проверить устойчивость каждого кластера и делать это в рабочее время, чтобы сотрудники могли реагировать на любые потенциальные последствия, когда у них есть для этого ресурсы, а не в 3 часа ночи, когда на телефонах обычно выключен звук. Увеличение частоты до одного раза в день действует наподобие регрессионного теста, гарантирующего, что инженеры не столкнутся с внезапным обвалом системы по причине такого отказа в процессе повседневных обновлений.

Опыт Netflix говорит, что это решение не сразу стало популярным. Был короткий период времени, когда инженеры по эксплуатации недовольно ворчали про Chaos Monkey. Но, похоже, идея сработала, и все больше и больше команд в конечном итоге стали брать метод на вооружение.

Вся суть этого простого приложения заключается в том, что оно взяло на себя ответственность за создание проблемы – исчезающие экземпляры влияют на доступность сервисов – и сделало это головной болью для каждого раз-

работчика. Как только проблема встала в полный рост, разработчики сделали то, что они обычно делают лучше всего: они решили проблему.

Действительно, когда Chaos Monkey каждый день неумолимо (хотя и управляемо) ронял какую-то службу, разработчики не могли спокойно работать, пока не решили эту проблему. И не важно, как они это сделали. Возможно, они добавили избыточность, может быть, автоматизировали масштабирование или пересмотрели шаблоны архитектуры. Это не имеет значения. На самом деле важно лишь то, что проблема была решена быстро и с немедленно ощутимыми результатами.

Достигнутый результат вырос из ключевого принципа культуры Netflix «высокая согласованность, слабая зависимость». Вредоносная обезьяна хаоса заставила всех быть в высшей степени ориентированными на достижение цели, максимально согласованно работать над решением проблемы, но не связывать себе руки в выборе решений.

Chaos Monkey – это прием менеджмента, реализованный в виде кода. Идея, стоящая за ним, выглядела необычно и слегка подозрительно, поэтому Netflix завел на эту тему отдельный блог. Довольно быстро Chaos Monkey стал популярным проектом с открытым исходным кодом и даже инструментом подбора персонала, благодаря которому Netflix предстал на рынке труда в новой роли – как идеолога творческой культуры разработки, а не просто развлекательной компании. Короче говоря, приложение Chaos Monkey стало признанным успехом и примером творческого подхода к риску как части культурной самобытности Netflix.

А теперь перемотаем время вперед до 24 декабря 2012 года, в канун Рождества[1]. В этот день в облаке AWS произошел перекос *эластичных балансировщиков нагрузки* (elastic load balancer, ELB). Эти компоненты анализируют поступающие запросы и распределяют трафик по вычислительным экземплярам, на которых развернуты службы. Когда балансировщики вышли из строя, прекратилась обработка новых запросов. Поскольку плоскость управления Netflix работала на AWS, клиенты не могли выбирать видео и запускать потоковую передачу.

Беда приключилась в наихудшее время. В канун Рождества Netflix собирался занять центральное место за столом, ведь первые пользователи хотели показать своей семье и гостям, как легко смотреть настоящие фильмы через интернет. Вместо этого бедные члены семьи и родственники были вынуждены разговаривать друг с другом, не отвлекаясь на контентную библиотеку Netflix.

Netflix испытал глубокий болевой шок. Авария стала ударом по имиджу компании и чувству гордости за технологии. Вдобавок сотрудникам компании совсем не понравилось, когда их вытащили из-за рождественского стола, чтобы наблюдать, как AWS спотыкается и снова падает в процессе восстановления.

Приложение Chaos Monkey помогло успешно решить проблему выпадающих экземпляров. Однако это локальная проблема. Можно ли построить

---

[1] *Adrian Cockcroft.* A Closer Look at the Christmas Eve Outage // The Netflix Tech Blog, Dec. 31, 2012, https://oreil.ly/wCftX.

нечто подобное для решения проблемы выпадающих *регионов*? Будет ли это работать в очень, очень большом масштабе?

Настало время роста

Каждое взаимодействие устройства клиента с потоковой службой Netflix осуществляется через *плоскость управления*. Это функциональность, развернутая в AWS. Как только выбрано потоковое видео, данные для него начинают поступать из частной сети Netflix, которая на сегодняшний день является самой большой сетью доставки контента в мире.

Перебои в канун Рождества вновь привлекли внимание руководства и специалистов компании к поиску активного подхода по обслуживанию трафика для плоскости управления. Теоретически трафик для клиентов в Западном полушарии можно разделить между двумя регионами AWS, по одному на каждом побережье. Если в одном из регионов происходит сбой, необходимо оперативно смасштабировать инфраструктуру другого региона и перебросить туда все запросы.

Это решение коснулось каждого аспекта потокового сервиса. Существует задержка распространения данных между побережьями. Некоторым службам пришлось доработать свои технологии, чтобы обеспечить согласованность между побережьями, придумать новые стратегии совместного использования состояний и т. д. Конечно, это сложная техническая задача.

И опять же, в структуре Netflix нет органа разработки, выдающего всем программистам какое-то готовое централизованное решение, которое гарантированно справилось бы с региональными сбоями. Вместо этого команда хаос-инженеров, заручившись поддержкой высшего руководства, координировала усилия различных команд.

Чтобы убедиться, что все эти команды готовы к решению задачи, была создана процедура перевода региона в автономный режим. Разумеется, провайдер AWS не позволил бы Netflix перевести целый регион в автономный режим из-за наличия других клиентов в регионе, поэтому аварию пришлось моделировать программными средствами. Процедура получила название Chaos Kong.

Первые несколько запусков процедуры Chaos Kong были весьма увлекательным приключением, когда специалисты собирались в «боевом штабе» и следили за всеми показателями потокового сервиса, а восстановление продолжалось часами. Затем запуски Chaos Kong пришлось прекратить на несколько месяцев, так и не добившись полного переноса трафика в один регион, потому что удалось выявить множество проблем, которые передали владельцам сервисов для устранения. В конце концов работа по переброске трафика была полностью отлажена и формализована и превратилась в обычную работу команды инженеров по управлению трафиком. Тем не менее Chaos Kong продолжали регулярно запускать, чтобы убедиться, что у Netflix есть актуальный план действий на случай, если какой-либо регион рухнет.

И действительно, потом было много случаев, когда из-за проблем со стороны Netflix или из-за неполадок в AWS какой-то один регион страдал от значительных простоев. В этих случаях срабатывал протокол обработки отказа региона, используемый в Chaos Kong. Выгода от инвестиций в новую технологию была очевидна[1].

К сожалению, переброска трафика при отказе региона в лучшем случае занимала около 50 минут из-за сложности интерпретации данных и необходимости ручного вмешательства. Несколько увеличив частоту запусков Chaos Kong, что, в свою очередь, заставило поднять инженерные стандарты в отношении отказоустойчивости регионов, команда инженеров по управлению трафиком смогла отладить новый процесс и в конечном итоге сократила обработку отказа региона до шести минут[2].

Вот мы и подошли к 2015 году. В распоряжении Netflix на тот момент были Chaos Monkey и Chaos Kong. Первый работал в ограниченном масштабе с выпадающими экземплярами, а второй – в большом масштабе с выпадающими регионами. Оба инструмента появились благодаря технической культуре компании и внесли очевидный вклад в доступность сервиса на этом этапе.

Формализация дисциплины

Брюс Вонг создал команду разработчиков Chaos в Netflix в начале 2015 года и передал задачу по разработке устава и дорожной карты Кейси Розенталю. Не совсем уверенный, во что он ввязался (его первоначально наняли для управления командой по обслуживанию трафика, и какое-то время он параллельно работал в двух командах), Кейси обошел Netflix, спрашивая, что люди думают о хаос-инжиниринге.

Обычно ответом было что-то вроде «хаос-инжиниринг – это когда мы специально ломаем разные места в производстве». Теперь это звучит круто и может стать отличным дополнением к резюме в профиле LinkedIn, но все-таки это пустое определение. Чтобы сломать что-нибудь в производстве, много ума не надо, и это может сделать любой сотрудник Netflix, имеющий доступ к терминалу, только вряд ли это принесет выгоду компании.

Кейси усадил свою команду за стол, чтобы разработать формальное определение хаос-инжиниринга. Они искали ясные ответы на вопросы:

- Как звучит определение хаос-инжиниринга?
- В чем суть хаос-инжиниринга?
- Как я могу убедиться, что действительно занимаюсь именно этим?
- Как я могу улучшить свои навыки?

Примерно через месяц работы над своего рода манифестом они разработали «Принципы хаос-инжиниринга». Дисциплина получила формальную оболочку.

---

[1] *Ali Basiri, Lorin Hochstein, Abhijit Thosar, and Casey Rosenthal.* Chaos Engineering Upgraded // The Netflix Technology Blog, Sept. 25, 2015, https://oreil.ly/UJ5yM.

[2] *Luke Kosewski et al.* Project Nimble: Region Evacuation Reimagined // The Netflix Technology Blog, March 12, 2018, https://oreil.ly/7bafg.

Формальное определение выглядело так: «*Хаос-инжиниринг* – это дисциплина экспериментов с распределенной системой, призванных подтвердить способность системы противостоять турбулентным условиям рабочей среды». Отсюда следует, что это особая форма *экспериментов*, которая стоит отдельно от *тестирования*.

Цель хаос-инжиниринга в первую очередь заключается в укреплении доверия к системе. Это полезно помнить, так что если вам не нужна уверенность в системе, то это не для вас. Если у вас есть другие способы укрепления доверия, вы можете взвесить, какой метод наиболее эффективен.

В определении также упоминаются «турбулентные условия рабочей среды», чтобы подчеркнуть, что речь идет не о создании хаоса. Хаос-инжиниринг – это прием, который делает видимым хаос, изначально присущий системе.

Далее «Принципы» описывают базовый шаблон для экспериментов, который в значительной степени соответствует принципу фальсификации Карла Поппера. Поэтому хаос-инжиниринг больше похож на науку, чем на технологию.

И наконец, в «Принципах» перечислены пять ключевых шагов, которые устанавливают золотой стандарт хаос-инжиниринга:

- построить гипотезу о стабильном поведении;
- продумать различные события реального мира;
- провести эксперименты в производстве;
- автоматизировать эксперименты для непрерывного запуска;
- минимизировать радиус поражения.

Каждый из них обсуждается по очереди в следующих главах.

Команда Netflix установила свой флаг на новой территории. Теперь они знали, что такое хаос-инжиниринг, как он работает и какую ценность представляет для большой компании.

Рождение сообщества

Как мы уже говорили, Netflix нанимал только старших специалистов. Иными словами, если вы хотите нанять хаос-инженеров, вам нужна группа опытных людей в этой области, из которой можно выбирать кандидатов. Но если вы только что создали новую дисциплину, вам трудно найти готовых специалистов. Негде было взять старших хаос-инженеров, потому что не было младших, – ведь за пределами Netflix их вообще не было.

Чтобы решить эту проблему, Кейси Розенталь решил действовать с присущим Netflix размахом и создать сообщество специалистов с нуля. Он начал с того, что осенью 2015 года организовал закрытую конференцию под названием «День сообщества хаоса». Она проходила в офисе Uber в Сан-Франциско, и в ней приняли участие около 40 человек. Были представлены следующие компании: Netflix, Google, Amazon, Microsoft, Facebook, DropBox, WalmartLabs, Yahoo!, LinkedIn, Uber, UCSC, Visa, AT&T, NewRelic, HashiCorp, PagerDuty и Basho.

Официальных тем для докладов не назначали, так что гости могли свободно говорить о проблемах, которые у них были, чтобы убедить руководство принять новую практику, а также обсуждать «провалы» и проблемы в неофициальном порядке. Организатор заранее выбрал докладчиков, способных рассказать о том, как они подошли к вопросам устойчивости, внедрения искусственных сбоев, тестирования аварийного восстановления и других методов, связанных с хаос-инжинирингом.

Учреждая «День сообщества хаоса», Netflix стремился подтолкнуть другие компании к созданию новой должности инженера по хаосу у них в штате. Это сработало. В следующем году «День сообщества хаоса» провели в Сиэтле в офисной башне Amazon Blackfoot. Менеджер из Amazon объявил, что после первого «Дня сообщества хаоса» они вернулись и убедили руководство создать команду хаос-инженеров в Amazon. Другие компании также ввели у себя должность хаос-инженера.

В том же 2016 году посещаемость выросла до 60 человек. На конференции были представлены такие компании, как Netflix, Amazon, Google, Microsoft, Visa, Uber, Dropbox, Pivotal, GitHub, UCSC, NCSU, Sandia National Labs, Thoughtworks, DevJam, ScyllaDB, C2, HERE, SendGrid, Cake Solutions, Cars.com, New Relic, Jet.com и O'Reilly.

По инициативе O'Reilly в следующем году команда Netflix опубликовала доклад на тему «Хаос-инжиниринг», который совпал с несколькими презентациями и семинаром на конференции Velocity в Сан-Хосе.

Также в 2017 году Кейси Розенталь и Нора Джонс организовали «День сообщества хаоса» в Сан-Франциско в офисе Autodesk на Market Street. (Кейси познакомился с Норой на предыдущей конференции, когда она работала в Jet.com. С тех пор она перешла в Netflix и присоединилась к команде Chaos Engineering Team.) Присутствовало более 150 человек, от завсегдатаев из крупных компаний Кремниевой долины до представителей стартапов, университетов и всего, что между ними. Это было в сентябре.

Пару месяцев спустя Нора выступила с докладом о хаос-инжиниринге на конференции AWS re:Invent в Лас-Вегасе, где 40 тысяч человек присутствовали лично и еще 20 тысяч смотрели потоковую трансляцию. Хаос-инжиниринг достиг большого успеха.

Стремительная эволюция

Как вы увидите на протяжении всей этой книги, концепция хаос-инжиниринга стремительно развивается. Это означает, что большая часть работы, проделанной в данной области, ушла далеко в сторону от первоначальной цели. Некоторые результаты могут даже показаться противоречивыми. Важно помнить, что хаос-инжиниринг – это прагматичный подход, впервые примененный в высокопроизводительной среде, столкнувшейся с уникальными масштабными проблемами. Этот прагматизм продолжает играть ключевую роль, несмотря на то что некоторые сильные стороны современного хаос-инжиниринга основаны на науке академического уровня.

Часть I

ОБЗОР ПОЛЯ ДЕЯТЕЛЬНОСТИ

На протяжении всей истории человечества сложные системы предоставляли их обладателям конкурентные преимущества. Военная наука, строительство, судоходство – люди, которым в то время приходилось работать с этими системами, сталкивались с таким количеством факторов, взаимодействующих непредсказуемым образом, что никогда не могли уверенно предсказать результат. Сегодня роль таких систем играют масштабные вычислительные системы.

Хаос-инжиниринг был задуман как дисциплина опережающего изучения и прогнозирования поведения сложных систем. В первой части этой книги представлены примеры сложных систем и обоснованы принципы хаос-инжиниринга в этом контексте. Содержание глав 1 и 2 изложено в естественном порядке, в котором опытные инженеры и архитекторы учатся управлять сложностью: размышляют, осваивают на деле, противостоят проблеме, приобретают опыт и, наконец, свободно ориентируются в этой области.

Глава 1 исследует свойства сложных систем, иллюстрируя эти свойства тремя примерами, взятыми из области вычислительных систем: «В случае сложных систем мы должны признать, что невозможно уместить все аспекты в голове одного человека». В главе 2 мы обращаем внимание на системный подход к управлению сложностью: «Целостный, системный подход хаос-инжиниринга – это одно из ключевых отличий от других практик». В главе представлены две модели для работы со сложностью: динамическая модель безопасности и экономическая модель сложности.

Глава 3 рассказывает об исследовании сложных систем, представленных в предыдущих главах, и формулирует основной принцип и определение хаос-инжиниринга: «Проведение экспериментов для выявления системных недостатков». В этой главе описана эволюция теории до настоящего момента и определены темы для последующих глав, посвященных реализации и развитию техники хаоса. «Принципы для того и разработаны, чтобы, занимаясь хаос-инжинирингом, мы знали, как это делать и как делать это хорошо».

Глава 1

Знакомьтесь: сложные системы

В первой части этой главы мы рассмотрим проблемы, возникающие при работе со сложными системами. Хаос-инжиниринг родился по необходимости в сложной распределенной вычислительной системе. В нем учтены особенности эксплуатации сложных систем, в частности присущая таким системам нелинейность, что делает их непредсказуемыми и, в свою очередь, приводит к нежелательным результатам. Это раздражает нас как инженеров, потому что нам нравится думать, что мы способны справиться с неопределенностью. Мы часто испытываем искушение винить в неполадках людей, которые строят и эксплуатируют системы, но на самом деле неприятные неожиданности являются естественным свойством сложных систем. Далее в этой главе мы спрашиваем, можем ли мы устранить сложность системы и тем самым устранить возможное нежелательное поведение. (Спойлер: нет, не можем.)

1.1. Размышления о сложности

Прежде чем вы сможете решить, имеет ли смысл хаос-инжиниринг для вашей системы, вам необходимо понять, где провести черту между простым и сложным. Один из способов охарактеризовать систему состоит в описании того, каким образом изменения во входных данных системы соответствуют изменениям в выходных данных. Простые системы часто называют линейными. Изменение на входе линейной системы производит пропорциональное изменение на выходе системы. Многие природные явления представляют собой линейные системы. Чем сильнее вы бросаете мяч, тем дальше он летит.

Нелинейные системы имеют выход, который может сильно меняться в зависимости от изменений в составных частях. Эффект кнута – пример системного явления[1], которое наглядно иллюстрирует нелинейность: легкое движение запястья (небольшое изменение на входе системы) приводит к тому, что дальний конец кнута разгоняется быстрее скорости звука и издает характерный громкий хлопок, которым известны кнуты (большое изменение на выходе системы).

---

[1] См.: *Peter Senge.* The Fifth Discipline. New York: Doubleday, 2006.

Нелинейные эффекты могут принимать различные формы: изменения частей системы способны вызывать экспоненциальные изменения на выходе; например, социальные сети растут быстрее, когда они большие, а не маленькие; или же воздействие может вызвать квантовый переход состояния системы, например если приложить нарастающую силу к жесткому стержню, то ничего не происходит до тех пор, пока стержень внезапно не сломается; или они могут выдать, казалось бы, случайный результат, например энергичная песня, которая вдохновит кого-то во время тренировки сегодня, но утомит на следующий день.

Линейные системы, очевидно, легче прогнозировать, чем нелинейные. Как правило, нам легко понять работу линейной системы, особенно после взаимодействия с одной из частей и получения пропорционального отклика. Поэтому мы можем сказать, что линейные системы являются простыми системами. Напротив, нелинейные системы демонстрируют непредсказуемое поведение, особенно когда взаимодействуют несколько нелинейных компонентов. Сочетание нелинейных реакций компонентов может привести к росту производительности системы до некоторого порога, а затем к внезапному изменению курса и к столь же внезапному обвалу. Мы говорим, что такие нелинейные системы являются сложными.

Мы можем дать менее техническое, но интуитивно более понятное определение сложности. Простая система – это система, в которой человек может понять все части, то, как они работают и как вносят вклад в результат. Сложная система, напротив, имеет так много связанных частей, или части меняются так быстро, что ни один человек не способен держать ее в уме. Взгляните на табл. 1.1.

Таблица 1.1 Простые и сложные системы

Простые системы	**Сложные системы**
Линейные	Нелинейные
Предсказуемый выход	Непредсказуемое поведение
Понятные	Невозможно построить полную ментальную модель

Глядя на характеристики сложных систем, легко понять, почему традиционные методы исследования безопасности систем неадекватны. Нелинейное поведение трудно симулировать или точно моделировать. Во всяком случае, люди не могут мысленно моделировать сложные системы у себя в голове.

В мире программного обеспечения нет ничего необычного в том, чтобы работать со сложными системами, которые обладают такими свойствами. Фактически следствием *закона необходимого разнообразия*[1] является то, что любая система управления должна иметь как минимум такую же сложность, как и система, которой она управляет. Поскольку сложность инструментов разработки постоянно растет, то и основная масса программного обеспече-

[1] См. комментарий к W. Ross Ashby's «Law of Requisite Variety» в сборнике: *W. Ross Ashby*. Requisite Variety and Its Implications for the Control of Complex Systems. Cybernetica 1:2 (1958), p. 83–99. Проще говоря, система A, которая полностью контролирует систему B, должна быть по меньшей мере такой же сложной, как система B.

ния со временем становится все сложнее. Если вы работаете в области информационных технологий и пока не сталкивались со сложными системами, то велика вероятность, что скоро вы с ними встретитесь.

Одним из следствий роста числа сложных систем является то, что традиционная роль разработчика программного обеспечения со временем становится менее актуальной. В простых системах один человек, обычно опытный разработчик, может управлять работой нескольких рядовых сотрудников. По совместительству он играет роль архитектора, потому что этот человек может мысленно смоделировать всю систему и знает, как ее части взаимодействуют между собой. Он может одновременно руководить текущей работой и планировать, как в продукте будут появляться новые функциональные возможности и технологии с течением времени.

Надо признать, что один человек не может удержать в своей голове все аспекты сложной системы. Это означает, что разработчики программного обеспечения должны более активно участвовать в разработке системы. Исторически инженерная деятельность – это бюрократическая профессия: одни люди решают, какую работу необходимо выполнить, другие решают, как и когда она будет выполняться, а третьи делают реальную работу. В сложных системах такое разделение труда является контрпродуктивным, потому что больше всего погружены в контекст именно те, кто решают реальные задачи. Работа архитекторов и связанной с ними бюрократии становится менее продуктивной. Сложные системы побуждают создавать *не*бюрократические организационные структуры, способные к эффективному прямому взаимодействию на всех уровнях.

1.2. Столкновение со сложностью

Непредсказуемая, непостижимая природа сложных систем ставит новые задачи. Дальше мы рассмотрим три примера сбоев, вызванных сложными сочетаниями. В каждом из этих случаев нельзя было ожидать, что команда разработчиков сможет предвидеть нежелательные эффекты.

1.2.1. Несоответствие между бизнес-логикой и логикой приложения

Рассмотрим архитектуру микросервиса, изображенную на рис. 1.1. В этой системе у нас есть четыре компонента.

Служба P

Хранит персонализированную информацию. Идентификатор представляет человека и некоторые метаданные, связанные с этим человеком. Для простоты будем считать, что хранимые метаданные никогда не бывают очень большими, и люди никогда не удаляются из системы. P передает данные в Q для сохранения.

Служба Q
Универсальная служба хранения данных, используемая несколькими вышестоящими службами. Она хранит данные в постоянной базе данных для обеспечения отказоустойчивости и восстановления, а также в кешированной базе данных на основе памяти для быстрого доступа.

Служба S
Постоянная база данных, возможно, столбчатая система хранения, такая как Cassandra или DynamoDB.

Служба T
Кеш в памяти, возможно, что-то вроде Redis или Memcached.

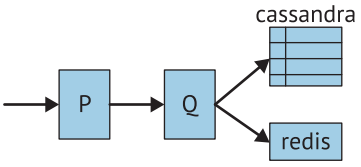

Рис. 1.1 ❖ Диаграмма компонентов микросервиса, показывающая поток запросов, поступающих в P и проходящих через хранилище

Команды, ответственные за каждый компонент, предвидят сбои, поэтому добавляют в систему некоторые разумные резервы. Служба Q будет записывать данные в обе службы: S и T. При извлечении данных сначала будет отправлен запрос к службе T, поскольку она работает быстрее. Если по какой-то причине произошел сбой кеша, данные будут прочитаны из службы S. Если не работают обе службы T и S, то служба Q отправит ответ по умолчанию о недоступности данных.

Аналогично, у службы P есть разумные резервы. Если служба Q просрочила ответ или вернула ошибку, то P может существенно снизить качество сервиса, возвращая ответ по умолчанию. Например, P может возвращать неперсонализированные метаданные для определенного человека, если Q возвращает ошибку.

Однажды служба T выходит из строя (рис. 1.2). Поиск в P начинает замедляться, потому что Q обнаруживает, что T больше не отвечает, и поэтому перебрасывает все запросы на чтение из S. К сожалению, подобным системам с большими кешами обычно присущи высокие требования к скорости чтения. В этом случае служба T достаточно хорошо справлялась с нагрузкой, поскольку чтение непосредственно из оперативной памяти происходит быстро, но S не может справиться с внезапным увеличением рабочей нагрузки. Работа службы S замедляется и в конечном итоге полностью прекращается. Она начинает возвращать ошибку истечения времени запроса.

К счастью, служба Q готова к такому развитию событий и возвращает ответ по умолчанию. Ответ по умолчанию для конкретной версии Cassandra при поиске объекта данных, когда все три реплики недоступны, – это код ошибки `404[Not Found]`, поэтому Q отправляет код `404` в P.

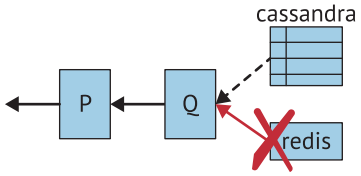

Рис. 1.2 ❖ В оперативном кеше T происходит сбой, в результате чего служба Q полагается на ответы из базы данных постоянного хранения S

Служба P знает, что человек, которого она ищет, точно существует, потому что у нее есть ID. Люди никогда не удаляются из службы. Поэтому ответ `404[Not Found]`, который P получает от Q, является неприемлемым с точки зрения бизнес-логики (рис. 1.3). Служба P могла бы обработать какую-то другую ошибку Q или даже полное отсутствие ответа, однако она не знает, как реагировать на этот невозможный ответ. Служба P падает, увлекая за собой всю систему (рис. 1.4).

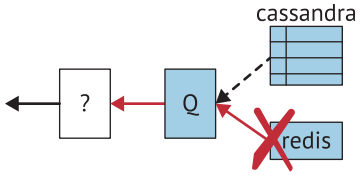

Рис. 1.3 ❖ Если T не отвечает, а S не в состоянии справиться с нагрузкой, требующей высокой скорости чтения, Q возвращает ответ по умолчанию на P

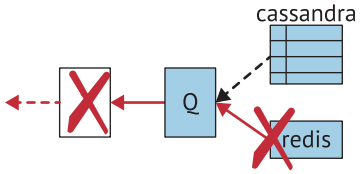

Рис. 1.4 ❖ Ответ по умолчанию `404[Not Found]` от Q выглядит логически невозможным для P, что приводит к катастрофическому сбою службы и обвалу системы

В чем проблема этого сценария? Выход всей системы из строя – это явно нежелательное поведение. Допустим, это сложная система, когда ни один человек не может учесть все аспекты ее работы. Каждая из команд, отвечающих за работу P, Q, S и T, приняла разумные инженерные решения. Они даже сделали опережающие шаги, чтобы реагировать на сбои и умеренно снижать качество сервиса. Так кто из них виноват?

На самом деле здесь некого винить. Это правильно скомпонованная система. Не стоит надеяться, что локальные команды смогут предвидеть *системный* сбой, поскольку взаимодействие компонентов превышает способность любого человека удерживать все факторы в своей голове и неизбежно при-

водит к необоснованным предположениям, что другие члены команды (или другие команды) лучше знают ситуацию. Неожиданная поломка этой сложной системы является выбросом, который вызван нелинейным сочетанием сопутствующих факторов.

Давайте рассмотрим другой пример.

1.2.2. Лавина повторных запросов пользователей

Рассмотрим следующий фрагмент распределенной системы из службы потокового видео (рис. 1.5). В этой системе у нас есть две основные подсистемы.

Система R
: Хранит персонализированный пользовательский интерфейс. При наличии идентификатора пользователя она возвращает интерфейс, оформленный в соответствии с настройками просмотра этого человека. R обращается к S для получения дополнительной информации о каждом человеке.

Система S
: Хранит различную информацию о пользователях, например есть ли у них действующая учетная запись и что им разрешено просматривать. Это слишком большой объем данных для размещения на одном экземпляре или виртуальной машине, поэтому S разделяет доступ, чтение и запись на две подсистемы:

  S-L
  : Балансировщик нагрузки, который использует алгоритм консистентного (согласованного) хеширования для распределения большой нагрузки чтения между компонентами S-D.

  S-D
  : Единица хранения с небольшим фрагментом полного набора данных. Например, один экземпляр S-D может хранить информацию обо всех пользователях, чьи имена начинаются с буквы «m», тогда как другой может хранить информацию о тех, чьи имена начинаются с буквы «p»[1].

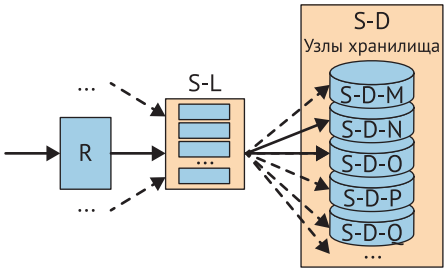

Рис. 1.5 ❖ Путь запроса данных о пользователе по имени Луи от R к S-D-N через S-L

[1] Механизм работает не совсем так, потому что алгоритм консистентного хеширования распределяет объекты данных псевдослучайно по всем экземплярам S-D.

Команда, которая поддерживает эту структуру, имеет опыт работы с распределенными системами и знакома с отраслевыми нормами облачного развертывания. Нормы включают в себя такие меры, как наличие рациональных резервов. Если R не может получить информацию о человеке из S, на этот случай есть пользовательский интерфейс по умолчанию. Обе системы также заботятся о стоимости, поэтому у них есть политика масштабирования, которая позволяет кластерам поддерживать необходимый размер. Например, если дисковый ввод-вывод S-D падает ниже определенного порогового значения, подсистема убирает данные с наименее занятого узла и отключает этот узел, а S-L перераспределяет рабочую нагрузку на оставшиеся узлы. Данные S-D хранятся в избыточном локальном кеше, поэтому если дисковое хранилище по какой-то причине работает медленно, из кеша может быть возвращен слегка устаревший результат. В системе предусмотрены оповещения о повышенных коэффициентах ошибок; детектор аномального поведения перезапускает экземпляры, ведущие себя странно, и т. д.

Однажды пользователь по имени Луи смотрит потоковое видео с этого сервиса в неоптимальных условиях. В частности, Луи получает доступ к системе через веб-браузер на своем ноутбуке в поезде. В какой-то момент с видео происходит странная вещь, которая удивляет Луи. Он роняет свой ноутбук на пол, случайно нажимая некоторые клавиши, и когда он снова ставит ноутбук на колени, чтобы продолжить просмотр, видео останавливается.

Наш Луи делает то, что сделал бы любой разумный пользователь в этой ситуации, – хаотично нажимает кнопку обновления 100 раз подряд. Запросы встают в очередь веб-браузера, но в этот момент поезд находится между вышками сотовой связи, и временный разрыв канала предотвращает доставку запросов. Как только канал связи восстанавливается, все 100 запросов доставляются одновременно.

На серверной стороне R получает все 100 запросов и инициирует 100 равноправных запросов к S-L, которая использует консистентный хеш идентификатора Луи для пересылки всех этих запросов конкретному узлу в S-D, который мы назовем S-D-N. Получение 100 запросов одновременно – это значительное увеличение нагрузки, поскольку S-D-N используется для обслуживания не более 50 запросов в секунду, и эти запросы уже поступают от других пользователей. Благодаря Луи происходит трехкратное увеличение нагрузки по сравнению с базовым уровнем, но, к счастью, у нас есть рациональные резервы и возможность снижения качества.

S-D-N не успевает обслуживать 150 запросов в секунду (базовый уровень плюс запросы Луи), поэтому он начинает отвечать на *все* запросы данными из кеша. Это значительно быстрее. В результате как дисковый ввод-вывод, так и загрузка ЦП резко снижаются. На этом этапе срабатывают политики масштабирования, чтобы стоимость эксплуатации системы соответствовала нагрузке. Поскольку дисковый ввод-вывод и загрузка ЦП так резко упали, S-D решает выключить S-D-N и передать его рабочую нагрузку равноправному узлу. Или, возможно, детектор аномалий отключил этот узел; иногда это трудно предсказать в сложных системах (рис. 1.6).

S-L возвращает ответы на 99 запросов Луи, все они извлечены из кеша S-D-N, но сотый ответ теряется из-за изменения конфигурации кластера,

когда S-D-N выключается и начинает переброску данных. В этом последнем ответе, поскольку R получает ошибку тайм-аута от S-L, система возвращает пользовательский интерфейс по умолчанию, а не персонализированный пользовательский интерфейс для Луи.

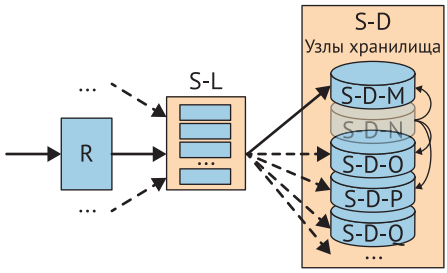

Рис. 1.6 ❖ Путь запроса от R к S-L к S-D-M для данных пользователя Луи после того, как S-D-N выключится и перебросит данные

Ответы приходят на ноутбук, где веб-браузер Луи успешно игнорирует 99 правильных ответов и отображает сотый ответ, который является пользовательским интерфейсом по умолчанию. По мнению Луи, это еще одна ошибка, так как это не его личный пользовательский интерфейс, к которому он привык.

Луи делает то, что любой разумный пользователь сделает в этой ситуации, и нажимает кнопку обновления еще 100 раз. На этот раз процесс повторяется, но S-L перенаправляет запросы в узел S-D-M, который взял на себя нагрузку S-D-N. К сожалению, передача данных еще не завершена, поэтому диск S-D-M быстро перегружается.

S-D-M переключается на обслуживание запросов из кеша. Как и в случае S-D-N, это значительно ускоряет запросы. Нагрузка на дисковый ввод-вывод и процессор резко снижается. Снова срабатывает политика масштабирования, и S-D решает завершить работу S-D-M и передать рабочую нагрузку одноранговому узлу (рис. 1.7).

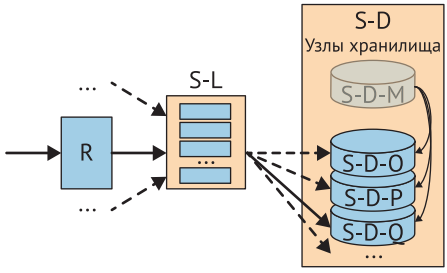

Рис. 1.7 ❖ Путь запроса от R к S-L и далее к S-D для пользовательских данных Луи после того, как S-D-M и S-D-N одновременно отключились и передали данные

Теперь подсистема S-D столкнулась с ситуацией, когда два узла отключились, но еще не успели перебросить свои данные. Эти узлы отвечают не

только за пользователя Луи, но и за некоторую часть остальных пользователей. Система R получает все больше ошибок тайм-аута от S-L для этой части пользователей, поэтому возвращает им пользовательский интерфейс по умолчанию, а не персонализированный пользовательский интерфейс.

Теперь и другие пользователи видят в своих браузерах то же, что и Луи. Многие из них считают это ошибкой, поскольку это не тот интерфейс, к которому они привыкли. Они также делают то, что любой разумный пользователь сделал бы в этой ситуации, и нажимают кнопку обновления 100 раз.

Теперь у нас есть лавина повторных запросов.

Цикл ускоряется. Все больше узлов подсистемы S-D переходят на быстрый кеш и отключаются. Резко возрастает задержка выдачи информации из хранилища, поскольку все больше узлов перегружено передачей данных соседям. Подсистема S-L изо всех сил пытается удовлетворить запросы. Но поскольку частота запросов от клиентских устройств резко возрастает, балансировщик вынужден увеличивать допустимое время ожидания ответа. В конечном итоге система R, сохраняющая все эти запросы к S-L открытыми, получает перегрузку пула потоков, что приводит к сбою виртуальной машины. Весь сервис падает (рис. 1.8).

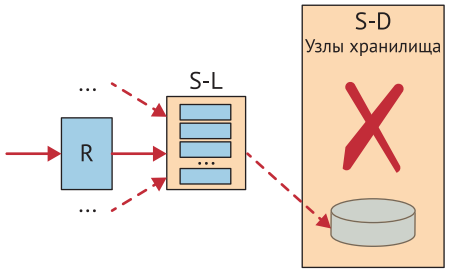

Рис. 1.8 ❖ Ситуация, когда подсистема S-D остановилась, а система R перегружена лавиной повторных запросов

Что еще хуже, сбой приводит к дальнейшему увеличению числа повторных запросов, инициированных пользователями, и это еще больше затрудняет решение проблемы и перевод службы обратно в стабильное состояние.

И снова можно спросить: кто виноват в этом сценарии? Какой компонент был построен неправильно? В сложной системе ни один человек не может держать в уме все рабочие компоненты. Каждая из команд, создавших R, S-L и S-D, приняла разумные инженерные решения. Они даже постарались предвидеть сбои, отслеживать эти случаи и разумно снижать нагрузку. Так кто же виноват?

Как и в предыдущем примере, здесь некого винить. Конечно, все мы сильны задним умом и *теперь* можем улучшить систему, чтобы предотвратить повторение только что описанного сценария. Тем не менее не следует думать, что разработчики должны были предвидеть этот сбой. В данном случае наложение разных факторов сформировало нелинейный отклик, который обрушил систему.

Давайте рассмотрим еще один пример.

1.2.3. Замораживание кода на праздники

Рассмотрим схему инфраструктуры (рис. 1.9) крупного розничного интернет-магазина.

Компонент E
: Балансировщик нагрузки, который просто перенаправляет запросы, аналогично эластичному балансировщику нагрузки в облачной службе AWS.

Компонент F
: Шлюз API. Он анализирует информацию из заголовков, файлов cookie и пути, а потом использует эту информацию для сопоставления с шаблоном политики дополнения; например, добавляет дополнительные заголовки, указывающие, к каким функциям пользователь имеет доступ. Затем шаблон приводится в соответствие с форматом серверной части (бэкенд) и отправляется на выполнение.

Компонент G
: Огромная свалка бэкенд-приложений, работающих с различными уровнями критичности на разных платформах и обслуживающих бесчисленное множество функций, адресованных неопределенному кругу пользователей.

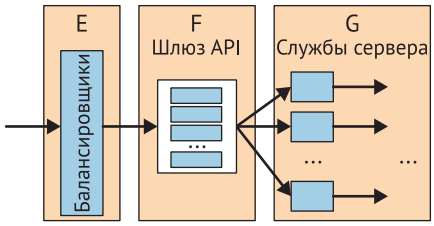

Рис. 1.9 ❖ Путь пользовательского запроса в крупном розничном интернет-магазине

Команда, поддерживающая компонент F, столкнулась с несколькими своеобразными препятствиями. У них нет контроля над стеком или другими рабочими свойствами G. Их интерфейс должен быть гибким, чтобы обрабатывать шаблоны различных форм в соответствии с заголовками запросов, файлами cookie и путями и пересылать запросы в правильное место. Функциональность G распределена по широкому спектру: от откликов с малой задержкой при небольших полезных нагрузках до поддержки длительных соединений при передаче больших файлов. Ни один из этих факторов не может быть точно запланирован заранее, потому что компоненты в составе G и за его пределами сами являются сложными системами с динамически изменяющимися свойствами.

Компонент F очень гибок и обслуживает разнообразные рабочие нагрузки. Новые функции добавляются и развертываются в F примерно один раз в день – это необходимо для реализации новых вариантов использования G. Чтобы поддерживать такой функционально нагруженный компонент, коман-

да с течением времени вертикально масштабирует решение по мере увеличения числа вариантов использования G. Они ставят все больше и больше серверных стоек, что позволяет им выделять больше памяти, хотя и требует больше времени для запуска. Постоянно растущий перечень шаблонов как для дополнения, так и для маршрутизации приводит к появлению гигантского набора правил, которые необходимо конвертировать в конечный автомат и загрузить в память для более быстрого доступа. Это тоже требует времени. В конечном итоге запуск такой громоздкой виртуальной машины, на которой работает F, занимает около 40 минут от момента старта конвейера инициализации до момента окончания загрузки кеша, когда экземпляр начинает работать с базовой производительностью или близко к ней.

Поскольку компонент F находится на критически важном пути любого запроса к G, команда, работающая с ним, понимает, что это потенциальная единая точка отказа. Они не просто используют один экземпляр; они развертывают кластер. Количество экземпляров в любой момент времени определено таким образом, чтобы весь кластер имел дополнительную емкость 50 %. В любой момент времени треть экземпляров может внезапно исчезнуть, и все должно продолжать работать.

Вертикальное масштабирование, горизонтальное масштабирование и сверхрезервирование; F – это дорогой компонент.

Чтобы сделать все возможное для обеспечения доступности, команда принимает несколько дополнительных мер предосторожности. Конвейер CI выполняет тщательный набор тестов модулей и целостности перед запуском образа виртуальной машины. Автоматизированные канареечные релизы проверяют любое новое изменение кода на небольшом объеме трафика, прежде чем перейти к развертыванию по протоколу типа Blue-Green, который параллельно запускает изменения на некоторой части кластера, прежде чем полностью перейти на новую версию. Все запросы на изменение рабочего кода на F проходят проверку двух рецензентов, и рецензентом не может быть кто-то, кто работает над изменяемой функцией. Это условие, благодаря которому вся команда может быть хорошо информирована обо всех аспектах рабочего процесса.

Наконец, вся структура замораживается на период с начала ноября до января. В течение этого времени не допускаются никакие изменения, за исключением случаев, когда это абсолютно необходимо для безопасности системы, поскольку праздники между Черной пятницей и Новым годом являются сезонами пикового трафика для компании. В этот период времени внесение ошибки ради побочной функции может иметь катастрофические последствия, поэтому лучший способ избежать неприятностей – вообще не менять систему. Поскольку многие сотрудники берут отпуск примерно в это время года, запрет на развертывание кода также обоснован с точки зрения надзора.

Затем, через год, происходит интересное событие. В конце второй недели ноября, после двухнедельного замораживания кода, команда обнаруживает внезапное увеличение количества ошибок на одном из экземпляров. Нет проблем: этот экземпляр выключен, а другой загружен. В течение следующих 40 минут, прежде чем новый экземпляр станет полностью работоспособным,

на некоторых других машинах также наблюдается аналогичное увеличение количества ошибок. Пока загружаются резервные экземпляры, остальная часть кластера испытывает ту же проблему.

В течение нескольких часов весь кластер заменяется новыми экземплярами, выполняющими один и тот же код. Даже при резерве в 50 % значительное количество запросов не обрабатывается в течение периода, пока весь кластер перезагружается за такой короткий интервал. Этот частичный сбой колеблется по серьезности в течение нескольких часов, прежде чем завершается весь процесс инициализации, и стабилизируется новый кластер.

Перед командой стоит дилемма: для устранения проблемы необходимо развернуть новую версию, содержащую средства наблюдения, сфокусированные на подозрительной области кода. Но система находится в состоянии заморозки, и вновь запущенный кластер по всем показателям выглядит стабильным. На следующей неделе команда решает все-таки развернуть небольшое количество экземпляров с новыми средствами мониторинга.

Две недели проходят без происшествий, и вдруг такая же проблема возникает снова. Сначала несколько, а в конечном итоге все экземпляры испытывают внезапное увеличение количества ошибок. Точнее, все экземпляры, кроме тех, которые были оснащены новыми средствами мониторинга…

Как и в предыдущем случае, весь кластер перезагружается в течение нескольких часов и, по-видимому, стабилизируется. На этот раз перебои в работе более серьезны, поскольку сейчас компания находится на пике сезонного спроса.

Несколько дней спустя экземпляры, оснащенные новыми средствами мониторинга, начинают испытывать тот же всплеск ошибок. Из собранных показателей видно, что одна из библиотек от стороннего разработчика вызывает постоянную утечку памяти, которая линейно зависит от количества обслуживаемых запросов. Поскольку экземпляры работают на очень мощном железе, утечка длится около двух недель, пока не займет достаточно памяти, чтобы вызвать нехватку ресурсов и неполадки в работе других библиотек.

Эта ошибка проникла в рабочий код почти девятью месяцами ранее. Подобное поведение никогда не наблюдалось, потому что ни один экземпляр в кластере никогда не работал непрерывно более четырех дней. Рабочие обновления приводили к регулярной перезагрузке экземпляров, и ошибка просто не успевала проявить себя. По иронии судьбы, к появлению системного сбоя привела именно процедура, предназначенная для повышения безопасности, – замораживание кода на праздники.

И снова вопрос: кто виноват в этом сценарии? Да, мы нашли ошибку в импортированной библиотеке, но даже если мы укажем пальцем на постороннего разработчика, который понятия не имеет о нашем проекте, – какой нам от этого прок? Каждый из членов команды, работавших над компонентом F, принимал разумные инженерные решения. Они даже постарались предвидеть сбои, внедрили этап развертывания новых функций, сверхрезервирование и «были осторожными» настолько, насколько это вообще возможно. Так кто же виноват?

Как и в двух предыдущих примерах, здесь *никто* не виноват. Было бы глупо ожидать, что разработчики способны предвидеть подобный сбой. Не-

предсказуемое и нелинейное сочетание факторов привело к неожиданному и дорогостоящему сбою в этой сложной системе.

1.3. Противодействие сложности

Три предыдущих примера иллюстрируют случаи, когда от людей, задействованных в производственном цикле, никто не может ожидать, что они предскажут факторы, сочетание которых в конечном итоге создаст проблемы. Люди продолжат писать программное обеспечение в обозримом будущем, поэтому вывести их из цикла – не вариант решения. Что нам остается делать, чтобы уменьшить системные сбои, подобные упомянутым выше?

Одна из популярных идей – уменьшить или устранить сложность. Уберите сложность из сложной системы, и у нас больше не будет проблем со сложной системой…

Возможно, если бы мы могли сократить эти системы до более простых, линейных, мы бы даже смогли определить, кто виноват, если что-то пойдет не так. В этом простом воображаемом мире мы можем представить себе сверхэффективного безликого менеджера, который способен предотвратить все ошибки, просто избавившись от плохих парней, которые их совершают.

Чтобы оценить это возможное решение, полезно усвоить несколько дополнительных характеристик сложности. Грубо говоря, сложность можно разделить на две группы: *случайную* и *намеренную* – это классификация, предложенная Фредериком Бруксом в 1980-х гг.[1]

1.3.1. Случайная сложность

Случайная сложность является следствием написания программного обеспечения в условиях ограниченных ресурсов, то есть в нашей реальной Вселенной. В повседневной работе всегда есть конкурирующие приоритеты. Для разработчиков программного обеспечения явными приоритетами могут быть скорость работы, охват тестирования или универсальность кода. Неявными приоритетами могут быть экономика, нагрузка и безопасность. Ни у кого нет бесконечного времени и ресурсов, поэтому попытка следовать нескольким приоритетам неизбежно приводит к компромиссу.

Код, который мы пишем, пронизан нашими намерениями, предположениями и приоритетами в определенный момент времени. Он по определению не может быть безупречным, потому что мир все равно изменится, и наши ожидания от кода изменятся вместе с ним.

Компромисс в программном обеспечении может проявляться как слегка неоптимальный фрагмент кода, неясное намерение в соглашении, двусмысленное имя переменной, расчет на последующую доработку кода и т. д. Эти

[1] *Frederick Brooks*. No Silver Bullet – Essence and Accident in Software Engineering // Proceedings of the IFIP Tenth World Computing Conference, H.-J. Kugler ed., Elsevier Science BV, Amsterdam, 1986.

фрагменты постепенно копятся в системе, словно грязь на полу. Никто специально не приносит грязь домой и не кладет ее на пол; это просто происходит как побочное следствие жизни в доме. Аналогичным образом неоптимальный код просто является побочным продуктом разработки. В какой-то момент эти накопленные неоптимальности превышают способность человека интуитивно понимать их, и в этот момент у нас возникает сложность, точнее случайная сложность.

Интересное свойство случайной сложности заключается в том, что не существует известного, *надежного* метода ее уменьшения. Вы можете уменьшить случайную сложность в какой-то момент времени, прекратив работу над новыми функциями, чтобы провести рефакторинг ранее написанного программного обеспечения. Это может сработать, но есть нюансы.

Например, нет оснований предполагать, что компромиссы, которые были сделаны во время написания кода, были хуже, чем те, на которые придется пойти при рефакторинге. Мир меняется, как и наше ожидание того, как программное обеспечение должно вести себя. Часто бывает так, что написание нового программного обеспечения для уменьшения случайной сложности просто создает новые формы случайной сложности. Эти новые формы могут быть более приемлемыми, чем предыдущие, но эта приемлемость исчезает примерно с той же скоростью, что и раньше.

Масштабные рефакторинги часто страдают от так называемого *эффекта второй системы* (second-system effect) – термин, также введенный Фредериком Бруксом, – когда ожидается, что последующий проект должен быть лучше оригинала из-за понимания, полученного во время разработки первой версии. Однако вместо этого вторые системы в конечном итоге становятся больше и сложнее из-за непреднамеренных компромиссов, вдохновленных успехом написания оригинала.

Независимо от подхода, принятого для уменьшения случайной сложности, ни один из этих методов не является надежным. Все они требуют отвлечения ограниченных ресурсов, таких как время и внимание, от разработки новых функций. В любой организации, целью которой является достижение прогресса, эти отклонения противоречат другим приоритетам. Следовательно, на них нельзя полагаться.

Таким образом, случайная сложность всегда нарастает как побочный продукт написания кода.

1.3.2. Намеренная сложность

Если мы не можем надежно уменьшить случайную сложность, то, возможно, получится уменьшить другой вид сложности? Источником намеренной сложности в программном обеспечении является написанный нами код, который добавляет проблемы просто потому, что такова наша работа. Как разработчики программного обеспечения мы пишем новые функции, а новые функции усложняют ситуацию.

Рассмотрим следующий пример: у вас есть самая простая база данных, которую вы можете себе представить. Это хранилище пар данных ключ/зна-

чение, как показано на рис. 1.10: дайте ей ключ и значение, и база данных сохранит значение. Дайте ей только ключ, и она вернет значение. Чтобы сделать ситуацию до абсурда простой, представьте, что код работает в памяти вашего ноутбука.

Теперь представьте, что вам дано задание сделать хранилище более надежным. Вы можете поместить хранилище в облако. Когда вы закрываете крышку ноутбука, данные сохраняются в облаке. Вы можете добавить несколько узлов для избыточности. Вы можете поместить пространство ключей в консистентный хеш и распределить данные по нескольким узлам. Вы можете сохранять данные этих узлов на диске, чтобы их можно было включать и отключать для восстановления или передачи данных. Вы можете реплицировать кластер в другой регион, и, если один регион или центр обработки данных станет недоступным, пользователи все равно смогут получить доступ к другому кластеру.

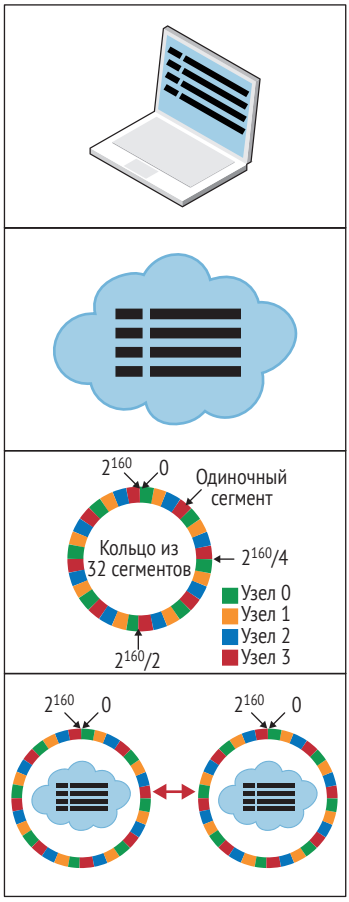

Рис. 1.10 ❖ Переход от простой базы данных ключ/значение к системе с высокой доступностью

Как видите, в одном абзаце можно описать множество известных принципов проектирования, чтобы сделать базу данных более доступной.

Теперь давайте вернемся к нашему простому хранилищу данных ключ/значение, работающему в памяти вашего ноутбука (рис. 1.11). Представьте, что вам дано задание сделать его более доступным и простым одновременно. Не тратьте слишком много времени, пытаясь решить задачу: это невозможно сделать в рамках здравого смысла.

Рис. 1.11 ❖ Возврат к простой базе данных ключ/значение

Добавление новых функций к программному обеспечению (или свойств безопасности, таких как доступность и безопасность) требует дополнительной сложности.

В целом перспектива борьбы со сложными системами в надежде получить простые системы не внушает оптимизма. Случайная сложность всегда будет являться побочным продуктом работы, а намеренная сложность будет зависеть от новых функций. Возрастание намеренной сложности является неизбежным следствием прогресса программного обеспечения.

1.4. Принятие сложности

Если сложность доставляет неприятности и мы не можем устранить сложность, то что нам делать? Решение состоит из двух этапов.

Первый этап – принять сложность, а не избегать ее. Большинство свойств, которые мы добавляем или оптимизируем в нашем программном обеспечении, подразумевают увеличение сложности. Попытка оптимизировать систему в сторону упрощения устанавливает неправильный приоритет и обычно приводит к разочарованию. Перед лицом неизбежной сложности мы иногда слышим: «Не добавляйте ненужной сложности». Конечно, это верно, но то же самое можно сказать и о чем угодно: «Не добавляйте ничего лишнего». Смиритесь с тем, что сложность неизбежно увеличивается, даже если программное обеспечение совершенствуется, и это неплохо.

Второй шаг, который является темой главы 2, заключается в том, чтобы научиться ориентироваться в сложности. Найдите инструменты для быстрого и уверенного движения вперед. Изучите методы добавления новых функций, не подвергая свою систему повышенному риску нежелательного поведения. Вместо того чтобы захлебнуться сложностью и утонуть в отчаянии, скользите по ним, как по волнам. Для вас как для разработчика хаос-инжиниринг может стать наиболее доступным и эффективным способом управления сложностью вашей системы.